Spring 5-2009

Atomic Transfer for Distributed Systems

Haraldur Darri Thorvaldsson
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Kenneth J. Goldman, Chair
Christopher D. Gill
Rudolf B. Husar
Robert E. Morley
William D. Smart
Jonathan S. Turner

ATOMIC TRANSFER FOR DISTRIBUTED SYSTEMS

by

Haraldur Darri Thorvaldsson

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

May 2009
Saint Louis, Missouri

ABSTRACT OF THE DISSERTATION

Atomic Transfer for Distributed Systems

by

Haraldur Darri Thorvaldsson

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2009

Research Advisor: Dr. Kenneth J. Goldman

Building applications and information systems increasingly means dealing with concurrency and faults stemming from distribution of system components. Atomic transactions are a well-known method for transferring the responsibility for handling concurrency and faults from developers to the software's execution environment, but incur considerable execution overhead. This dissertation investigates methods that shift some of the burden of concurrency control into the network layer, to reduce response times and increase throughput. It anticipates future programmable network devices, enabling customized high-performance network protocols.

We propose Atomic Transfer (AT), a distributed algorithm to prevent race conditions due to messages crossing on a path of network switches. Switches check request messages for conflicts with response messages traveling in the opposite direction. Conflicting requests are dropped, obviating the request's receiving host from detecting and handling the conflict. AT is designed to perform well under high data contention, as concurrency control effort is balanced across a network instead of being handled by the contended endpoint hosts themselves.

We use AT as the basis for a new optimistic transactional cache consistency algorithm, supporting execution of atomic applications caching shared data. We then present a scalable refinement, allowing hierarchical consistent caches with predictable performance despite high data update rates.

We give detailed I/O Automata models of our algorithms along with correctness proofs. We begin with a simplified model, assuming static network paths and no message loss, and then refine it to support dynamic network paths and safe handling of message loss.

We present a trie-based data structure for accelerating conflict-checking on switches, with benchmarks suggesting the feasibility of our approach from a performance standpoint.

Acknowledgments

First of all I want to thank my thesis advisor, Ken Goldman, for his inexhaustible patience with me throughout my doctoral journey. I have learned much from his calm and careful thinking, and aspire to add some of that style to my own.

I would also like to thank the other members of my dissertation committee, Chris Gill, Rudy Husar, Bob Morley, Bill Smart and Jon Turner, for the time and valuable insights they contributed.

I want to thank the guys of Lunch Club, for all our great times together. Octav Chipara and Paul Gross for good friendship, and Paul especially for our hours spent in wise conversation and inane banter. Sajeeva Pallemulle, the oldest soul of us all, for his friendship and enjoyable cooperation on all things Byzantine. Rohan Sen with his boundless knowledge of all things WashU and beyond. Greg Hackmann for his ready assistance with any problem. Also, Mart, Liang, Jamie, Radu, it was a pleasure knowing you all.

My thanks also to the peerless administrative staff at the department: Myrna, Madeline, Andrea and Sharon, I've greatly enjoyed your company and benefited from your helpfulness.

Last but not least I thank my amazing wife, Anna Margrét, and my wonderful children Halldór Alexander, Jökull Ari and Hugrún Eva. I've had to ask much of you, especially during the last miserable months without you. From now on, I'm all yours.

Haraldur Darri Thorvaldsson

Washington University in Saint Louis
May 2009

Dedicated to my family.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Contributions

Humanity's data, business processes and social interactions are rapidly moving online. Before long, the pervasiveness of the Internet will be complete in the developed world, with everyone permanently connected at work, home or on the go through a wireless device. In the developing world, mobile phone technology is already making a positive impact on people's lives. But in general, the expectation is growing that any information about anything should be accessible at any time from anywhere in the world. These kinds of trends will likely continue, and thinking them through to their logical conclusion we arrive at the following:

1. We want unfettered access to applications of arbitrary complexity from anywhere, with a user interface experience matching or surpassing that of applications installed on local workstations today.

2. We want to view fresh data, updated in our user interfaces not within minutes or seconds but within milliseconds of changing. The intuitive illusion of every data item having one global consistent copy should be maintained at all times.

3. We want to modify data in real time in collaboration with other people, regardless of physical location. The default mode of working with information should be to observe it and interact with it, not distribute it or download it.

4. We want it to work! We should never lose time, work or data due to component failures in our computing infrastructure.

Systems meeting these requirements will inevitably be distributed, due to the need for global accessibility and scalability. Developing such systems is highly challenging though, due chiefly to the concurrent execution and partial failures inherent in distributed systems. Attempts by multiple applications to modify the same data must be reconciled and the transient or permanent failure of some of the tens or hundreds of components that may partake in an application's execution must be tolerated. The futility of handling these issues on an ad-hoc basis has long been recognized by the designers of large, mission-critical data processing systems. A widely adopted solution for handling them is the *atomic transaction* [1, 2].

An atomic transaction is a collection of operations that must appear to execute together as a whole or not at all. In the first case a transaction is *committed* and its effects on data are permanently recorded, while in the latter case it is *aborted* and no data is affected. Furthermore, transactions are prevented from interfering, so a transaction never sees the intermediate effects of any concurrently executing or aborted transactions but only the consistent state left behind by committed transactions. This is achieved through the use of *concurrency control algorithms*, that delay, re-order, abort or otherwise manipulate transactions to ensure that the result of an execution could have been produced by some total order of the transactions.

The performance overhead of concurrency control can be significant. Enterprise-critical database systems and applications have long chosen to pay this price, as the price of system failures is higher still. A failed fund transfer operation in the computing system of a bank or financial institution, for example, could easily lead to the loss of the transfer's money and the firm's reputation. But while Internet applications commonly rely on a transactional database or data store behind the scenes, transactions have yet to be adopted as the basis for software development in general. We propose an architecture for network-centric concurrency control designed to facilitate such a transition. The key architectural ingredients of our approach are:

1. An execution model that makes progress by executing atomic transactions.
2. Pervasive caching of persistent global data, with atomic cache consistency.
3. Publish/subscribe multicasting and forwarding of data as name/value pairs.

This dissertation makes the following primary contributions.

1. We propose an algorithm for distributed concurrency control with network switch participation, enabling an aggressively speculative atomic request / response protocol and reducing the load on servers imposed by concurrency control.

2. We develop a scalable atomic cache consistency protocol that relies on the guarantees provided by the network for safety.

We make certain forward-looking assumptions about the network. In particular, we assume that we can customize the network's routing and forwarding protocols and functionality. While we do not advance the art of multicast routing or subscription processing per se, our approach depends heavily on these elements.

We take atomic transactions and persistent data in a global namespace as the basis for our approach. We take an *optimistic* approach to concurrency control, where applications first execute transactions on locally cached data before sending the transactions to a server for commitment. Existing work suggests that this approach leads to good performance and scalability [3]. This dissertation investigates ways to further improve the performance of such systems. We provide a detailed model of our algorithms and prove them to be correct.

We do not present a complete implementation, but we do evaluate the performance of a data structure for the performance-critical processing that would take place on network switches in an implementation of our approach.

1.2 Overview of Approach

Our novelty stems from recruiting the network layer into making distributed atomic execution and cache consistency more efficient. Data networks have a graph-like structure, whose main components are switches[1] connected via data links. This structure

[1]We use the term somewhat inaccurately to refer to both switches, routers and related devices.

is usually opaque to applications, which treat the network as a “cloud” that accepts packets tagged with destination network addresses and delivers them to addressees, sometimes losing, rearranging or duplicating packets in the process. We are mainly concerned with packets containing operation requests and responses. Requests are sent from application hosts to server hosts to invoke operations while responses are sent from servers to report back operation results. We observe that the forwarding of packets is an active computation process, performed by the switches along network paths. Most importantly, packets navigating a network path between a pair of hosts pass through the same switches.

This sets the stage for Atomic Transfer. We ask: what if requests could “collide” with responses at the switch where they meet in the network, such that a request rendered invalid by a recent response would be dropped and never reach its destination server? For example, if a request to toggle the status of a bit from 0 to 1 encountered a response noting that the bit has just been set to 1, the request would no longer be applicable. With traditional concurrency control, the server would detect this situation and resolve it, for example by rejecting the latter request. But while the effort a server expends on concurrency control is usually manageable, it can increase considerably in the presence of heavy data contention, where many applications concurrently attempt to access the same data in incompatible ways. By contrast, if most such conflicts were resolved in the network, they would not consume any server resources whatsoever.

While today’s switches perform fixed data routing and forwarding according to standard protocols, there is nothing that fundamentally precludes them from performing other types of processing as well, such as checking packets for concurrency conflicts. However, conflict checking can only add a very slight overhead to switch processing before its gains are outweighed by increased normal-case forwarding costs. Our results indicate that conflict checking can be sustained at multi-gigabit data rates, although we’ve yet to verify this in a real implementation.

Atomic Transfer requires that application hosts receive the uninterrupted sequence containing all responses that may conflict with their requests, to ensure that conflicting requests and responses meet in the network. For this reason, we assume multicasting capability in the network, for efficient transmission of responses to a large number of receiving applications. Although not widely deployed, multicasting has been the focus of much research. Since we already assume that we can customize switches and network protocols, assuming multicast does not seem contentious.

We make an important observation, that the threat of concurrency control overwhelming a server is especially acute with the family of algorithms known as Optimistic Concurrency Control (OCC). At the same time, OCC works particularly well in conjunction with application-side caching of data, while readily admitting the type of speculative execution that can mask network latency to a large extent. We perceive a beneficial synergy here: the multicast architecture required for Atomic Transfer enables efficient maintenance of application caches, while Atomic Transfer may mitigate cache-friendly OCC's principal weakness, i.e. its failure to cope with heavy contention. We believe that an architecture utilizing Atomic Transfer-based cache consistency could meet our high-level goal: scalable and low-latency access to shared data and applications.

In our proposed model, application hosts are essentially stateless, but cache arbitrary amounts of data from the servers tasked with persistent data storage. Application hosts non-deterministically issue requests and apply server responses to their cached data. We use the term "application" in a broad sense. For example, an application host could correspond to a user device, with non-determinism stemming from user input. As a different example, an application could correspond to a front-end host, with non-determinism stemming from incoming legacy RPC calls [4, 5]. As a final example, it could correspond to a proactive monitoring agent or workflow processing controller, with non-determinism stemming from timing and ordering of responses and scheduling decisions. In each case, the execution model presented to software developers[2] is very simple: all data is globally scoped and is modified only by transactions, that seemingly move the system from one global state to another.

To enable rapid conflict checking of requests and responses we require that data be organized as global name / value pairs, such that requests invoke operations on data names and responses report the results of operations on data names. In an implementation, names would be encoded into packets in a standardized format such as the one described in Chapter 8, enabling switches to perform conflict detection with

[2]And ultimately: application end-users and data owners.

limited knowledge of the semantics of requests and responses. In fact, our models go a step further and assume that switches forward packets based on data names, not host or network identifiers. While this is not strictly required for our approach, it makes our protocols simpler, more self-contained and more highly abstracted.

This dissertation is organized as follows. Chapter 2 discusses related work at a relatively high level. Additional related work is discussed in context, at the end of each technical chapter. Chapter 3 models a network of switches that perform Atomic Transfer of requests and responses and shows that they function correctly, assuming certain well-formedness conditions are met by application and server hosts. Subsequent chapters present a series of refinements of the switches, applications and servers comprising an atomic network. Chapter 4 adds servers and data-caching applications to the network, and shows that application executions are atomic, or serializable, given the appropriate operation conflict relation. The model in these chapters assumes a reliable network and a static association between applications and servers. The system of Chapter 5 relaxes these assumptions, permitting applications to dynamically subscribe to servers and subscribe to only subsets of a server's state. Furthermore, it tolerates message loss, although with loss of performance. Chapter 6 improves on the cache system of Chapter 4 for dynamic/partial subscriptions. Chapter 7 further improves the scalability of the system of Chapter 6 by permitting hierarchical caches, allowing applications to fetch data into their caches from the caches of other applications. Chapter 8 presents a concise, trie-based data structure that could serve as the basis for high-performance conflict detection on switches, along with synthetic micro-benchmarks for the data structure. Chapter 9 discusses future extension of our work to multi-server transactions, as well as suggesting a few optimizations.

Chapter 2

Background and Related Work

This chapter gives an account of related work, organized around a discussion of our architectural ingredients. It gives a relatively broad overview of relevant fields, while the technical chapters present more detailed discussions of related work in the context of the material presented in each chapter.

2.1 Transactions and Concurrency Control

A variety of formal and semi-formal definitions of transactions have been proposed, but their essence is that an atomic transaction is a set of operations seemingly executing together as an indivisible whole, never executing partially nor interleaving with operations from other transactions.

While the ideas underlying transaction processing originate with the development of mainframe database systems in the late sixties [6], the terms and concepts began to take shape in the early seventies [7]. The classic transaction papers [1, 2] describe the notion of atomic transactions and serializability [8], as well as rules and implementation techniques for locking and distributed commit of multi-server transactions. Concurrency control is a rich, mature field [9] and we do not attempt a complete survey.

The defining properties of transactions are often known by the acronym ACID: Atomic, Consistent, Isolated and Durable. Atomic, because partial results of transactions that fail to commit are never visible to outside observers, Consistent because a (correctly programmed) transaction moves a system from one consistent state to another, Isolated because a transaction never sees intermediate results from other concurrently executing transactions, and Durable because the effects of a committed transaction survive transient system failures, such as crashes that lose the contents of volatile memory.

The purpose of concurrency control is to permit multiple independent applications to access shared data while preventing anomalies resulting from interfering data updates. For example, if two fund transfer programs interfere in such a way that one deposit is lost, that would be considered an anomaly. In general, it may be less clear what constitutes an anomaly-free, correct execution. Various formal execution correctness standards have therefore been developed, giving precise conditions for which interleavings of operations in a concurrent execution are permitted. All are based on the notion of a concurrent execution being *consistent* with some sequential, non-concurrent execution of the same operations. Intuitively, an execution is correct if none of the participants in it can distinguish it from a sequential execution. The most widely used standard in transactional systems is that of *serializability* [1], which says that a concurrent execution is correct if its operations could be re-ordered such that 1) each operation of each transaction appears before or after those of other transactions, and 2) the re-ordered sequence is a correct sequential (serial) execution of the transactions. There are many subtle aspects to such definitions [9], with important consequences for the consistency of state observed by transactions, the rollback of aborted transactions (transaction *recoverability*) and the achievable levels of concurrency and performance. Our algorithms are based on conflict serializability, the variant most commonly used in practice, where only those operations defined as non-conflicting may be re-ordered for correspondence to a sequential execution.

A notable alternative to serializability is *linearizability* [10], mainly used to reason about correctness of concurrent data structures and algorithms in shared-memory concurrent systems without recoverability. Another standard, originally defined in terms of shared-memory multiprocessors, is that of *sequential consistency* [11].

A concurrency control algorithm constrains the execution of transactions at runtime as to preserve an execution's consistency. Concurrency control algorithms fall into three main classes: algorithms based on (pessimistic) locking, timestamps and (optimistic) validation. The first class is dominant in current practice, but many hybrid schemes exist, combining elements from more than one class.

- In locking schemes, transactions obtain exclusive (write) or shared (read) locks on data items before using them. A transaction attempting to obtain a lock incompatible with an existing lock on an item becomes blocked and must wait until the lock is released again. Alternatively, either transaction may be aborted. Locking is a relatively straightforward, well-understood technique and handles contention well, granted that locks are held for short durations of time. It suffers performance loss from blocking, which increases rapidly with increased transaction waiting times and contention [12]. This makes it less attractive for distributed systems, where network latency may contribute to waiting times. It also suffers the well-known drawbacks of locking, such as deadlocks and convoying.

- In timestamp schemes [13] the ordering of a transaction is decided as it is created, by assigning to it a unique timestamp from a totally ordered domain. The concurrency control algorithm then decides whether operations are permitted to complete based on transaction timestamps. For example, if a transaction with a lower (earlier) timestamp attempts to read a value written by a transaction with a higher (later) timestamp, either one of the transactions must be aborted. Pure timestamp schemes have been found to perform worse than locking or optimistic methods, owing to their conservativeness in permitting concurrency and the scheduling rigidness resulting from choosing transaction ordering in advance. They are more commonly used as a part of other concurrency control schemes [14], for example in the validation phase of optimistic schemes.

- In optimistic schemes [15] a transaction is executed in isolation, with changes made to a local buffer invisible to other transactions. Once ready to commit, the transaction enters a *validation phase* that checks whether the transaction is still serializable, that is: whether it has been invalidated by another transaction in the meantime (*backward validation*) or alternatively: whether it would invalidate some other as of yet uncommitted transaction (*forward validation*). If not, it is committed and its changes are atomically propagated to the globally visible state. In a distributed system, a requesting host must obtain the data values read by a transaction and / or cache a copy of those values. If hosts cache data across transactions, the concurrency control problem effectively becomes that of ensuring *cache consistency* [16]: that the values read from cache could have come from the original state.

Our algorithm falls under the rubric of optimistic transactional cache consistency algorithms, as requesting applications execute transactions locally using cached data and then send the request (and possibly its response) to the original server for validation. This results either in an update to the server's global state or rejection and rollback in the requester's cache. An important difference between pessimistic and optimistic algorithms is that pessimistic transactions are ready to commit at any time, since they've "stopped the world" to preserve their state assumptions. Optimistic transactions execute blithely based on cached state, but can only commit following a successful validation phase verifying that their assumptions still hold.

We state our models in terms of operation invocations on abstract data types [17, 18], rather than low-level data reads and writes. Aside from generality, abstract data types can support enhanced levels of concurrency, by supporting a wider range of non-conflicting operations that can proceed in parallel [19].

As an important note, the models in this dissertation do not permit a transaction to be distributed across more than one server, that is: all the operations of a transaction must be executed at the same server. Multi-server transactions need additional processing steps to ensure that a transaction commits at all the servers involved or none of them, and that transactions can be serialized in compatible orders at all servers. While multi-server transactions are a fundamental requirement for scalable transactional systems, postponing their consideration simplifies our discussion and keeps it focused on the novel contributions of this dissertation, which is Atomic Transfer. However, we have begun work on extending our protocols to handle multi-server transactions, using the network to accelerate distributed atomic commit, as discussed in Chapter 9 on future work.

Transactions are today nearly synonymous with database systems and enterprise applications. Innumerable database systems have been built, academic and commercial.

Pioneering implementations such as IBM's IMS [20] and System R [21] ran in mainframe environments. In the eighties, research on scalable "database machines" shifted the focus from custom hardware solutions [22] to shared-nothing architectures [23, 24] and eventually to commodity hosts and interconnects, the platform of choice for today's commercial database systems.

Databases gain considerable implementation flexibility from the relational data model [25], allowing automated partitioning of data and parallelization of (declarative) queries, for example. However, despite many attempts at addressing it, an "impedance mismatch" remains between databases and general-purpose programming languages based on memory objects and pointers. Object-Oriented (OO) Databases [26, 27] seek to meld OO programming into databases, while work on orthogonally persistent programming languages [28, 29] seeks to meld databases into languages. While OO databases have had some commercial success, they have had a limited impact on general software development practice.

An alternative approach is to provide atomicity at a lower level, in language-agnostic transactional data stores [30, 31, 32, 33], file systems [34, 35] or virtual memory [36]. Such systems suggest a more layered approach, as apposed to the fully integrated approach of databases, and could provide a foundation for AT-base server implementations. There have been calls for more modular architectures for database systems [37, 38].

Active Databases [39, 40] are yet another approach to bridging the gap, embedding atomic application actions within a database's execution environment. Actions are executed according to Event-Condition-Action (ECA) rules, specifying the changes (events) and a predicate (condition) for when an action may be scheduled to run, similar to guarded commands [41]. These ideas have influenced mainstream databases, most of which support ECA-like "triggers" for stored procedures executing within the database. Our model would be suitable for ECA-structured applications, with condition evaluation performed by application hosts in response to cache updates.

An interesting recent area of research is *transactional memory* (TM) [42, 43, 44], attempting to provide transactional semantics for batches of memory location loads and stores, using extensions to memory cache coherency logic (hardware TM), software techniques (software TM) or both. TM is motivated by the ongoing shift to multi-processors that require application-level concurrency for full utilization. TM has reignited some interest in atomic language constructs [45, 46]. Our work is conceptually related to TM, if viewed as an extension of TM to clusters of hosts, treating the network like a scalable cache coherency controller. Yet, performance of software TM prototypes has been disappointing so far [47]. We conjecture that providing atomicity at the low level of memory loads and stores suffers similar overhead characteristics as do distributed memory systems (see Section 2.3).

ACID transaction have been criticized for being overly restrictive and unsuitable to emerging types of applications, such as long-running work-flow processing systems and intermittently connected mobile applications, e.g. This has spurred considerable research into relaxed notions of atomicity [48, 49, 50]. A common technique is to use *compensating actions* to undo the effect of transactions that were "prematurely" committed in order to unblock other activities. Although new models are being proposed to this day, they have seen limited use as the basis for system implementation. In any case, most such extensions rely on an underlying notion of atomicity of operations. Similarly, systems that preserve availability during transient partitions[3] by offering weaker *eventual consistency* "guarantees" [52, 53, 54, 55], and attempt to reconcile conflicting operations upon reconnection [56], also assume atomic actions as an underlying framework. Many of the world's most highly scalable data stores offer very selective atomic guarantees, for example on a per-row or per object basis [57, 58, 59]. These systems relax consistency guarantees to achieve higher availability and lower response times.

Our view is that full, "hard-nosed" atomic actions should always be available as a basic system primitive, since their performance will suffice for all but the most demanding of applications and any execution overhead will be paid back through significant decreases in development and operational costs accruing from simplified programming models and robust fault handling. When atomic actions are too expensive, one can always relax the degree of consistency they provide by breaking large transactions into smaller ones and handling any resulting scheduling, progress and consistency issues on an ad-hoc basis or through some application-specific methodologies.

[3] A fundamentally unavoidable trade-off [51]

2.2 Atomic Actions

We have described transactions as an aggregation of individual operations. It is also instructive to view them as a mechanism for matching the atomicity of an application's execution steps to the application's structure and semantics, instead of matching it to the structure of the computing hardware. A transaction turns a group of operations into an *atomic action* [60, 61, 17], that captures the underlying structure of a program as a set of discrete state transitions [41]. For example, writing a word of memory may be atomic at the machine instruction level, but it might suit an application's semantics better to update some two particular words atomically together (or not at all). Similarly, it better suits a banking application to have the debiting and crediting of accounts involved in a fund transfer occur as a single indivisible step, instead of the multiple separate steps of sending and receiving requests over a network, computing balances and reading and writing data to non-volatile storage, etc.

This is the enduring attraction of atomic actions: the software developer implements operations at an abstraction level and granularity that suits the task at hand, while the execution environment ensures they appear to execute atomically. This largely frees the developer from having to anticipate and handle failures and concurrent interference, allowing her to pretend that actions runs sequentially in a failure-free world.

The early eighties saw a flurry of transaction research, shoring up the formal foundations of atomic execution while extending its definition in many ways, most notably to hierarchically nested transactions [62, 63] that allow transactions to recursively contain other, concurrently executing transactions. ARGUS [64, 65] was a pioneering distributed transactional programming system, adding atomic execution constructs to the CLU programming language. The Camelot [66] system had full support for nested transactions. It was based on C and the C++ derived Avalon language. While these systems were highly influential, they suffered from lackluster performance. The follow-up to ARGUS, Thor [67], improved performance with new optimistic cache consistency algorithms [14], among other things. Another notable system is Quicksilver [35], a UNIX variant with atomic execution semantics for processes and a transactional file system. Performance overhead was moderate and the researchers reported positively on their experience with the use of transactions. Our work is directly inspired by these pioneering efforts, although we provide only a well-defined execution model and refrain from dictating programming languages.

The advantages of atomic actions cannot be overstated. Anticipating every failure and writing the code to roll back partial state changes is an error-prone and arduous task. Anticipating and preventing undesirable interleavings of data accesses from concurrently executing programs or threads is harder still [68], and even thorough testing will likely leave some latent bugs undiscovered. Co-mingling failure-case and normal-case code tends to obfuscate the latter, making maintenance hard and error-prone. But in addition to their manifold benefits for developer productivity and system robustness, atomic actions directly reify the notion of application state transitions and states. This significantly aids the implementation of replication[69], as two executions can be kept synchronized by ensuring they execute the same sequence of actions[4]. It also aids state migration and (dynamic) system reconfiguration [70, 71], as executions can be readily halted in a well-defined state and restarted on a different host, after a copy of the state has been installed on it.

A final and possibly underrated benefit of well-defined, relatively coarse-grained atomic actions is their potential for turning any execution into a shared, multi-user experience. An application based on atomic actions can be readily shared by several hosts, each caching the application's state and sending locally initiated operation requests to the application's server. Since the server's concurrency control decides (or restricts) the "official" order in which operations execute, the execution appears as if all hosts execute operations sequentially on a single copy of the application, less aborted actions. The sharing might take place at the level of the application's data structures, with the entire application cached at each host. Alternatively, it might take place at the level of its user interface elements, with the elements translating local user input into operation requests for the application's server. The latter is an interesting option, as it might allow a relatively low-powered devices to access remotely executing computationally demanding and / or data-rich applications. While this dissertation does not further explore these scenarios, we believe our model, combining distributed atomic actions with multicast-based cache synchronization, would be a good fit.

[4]Assuming circumscription of non-determinism.

Why the world is not yet sold on programming with atomic actions may owe something to the fragmented and awkward architecture of contemporary information systems, with non-atomic, non-recoverable programs written in general-purpose programming languages trading queries and messages with transactional databases and/or atomic program snippets written in highly specialized and proprietary languages executing within the database domain. A tortuous mapping between the language's object model and the relational model is often involved. A common protocol for atomic data operations might help break this logjam. But another important reason may be the performance overhead of (distributed) transactions, perceived and real. If our protocols succeed in lowering that overhead, this would strengthen the case for atomic programming.

2.3 Atomic Cache Consistency

Data caching is indispensable for building high performance computing systems. Caching plays a crucial role at every level of a computing system's architecture, from the memory caches built into modern CPUs and caching of disk data in memory to the caching of remote data on proxy hosts and middle-tier database hosts. Indeed the World Wide Web might not work quite as well as it does were it not for the world-wide caching of pages in the "content delivery networks" of Akamai [72] and similar providers.

Our models are based on a *master/slave* configuration, where each data item is hosted on a particular server at any one time and all other copies of it are considered caches. This model is simpler than replicated *multi-master* models, but can lead to the master server becoming a bottleneck. Our contributions to this problem are twofold: a network-based concurrency control algorithm that preserves the goodput of servers under data contention and a multicast-based scalable caching architecture that unburdens the server from serving read requests while maintaining consistency despite heavy data update rates. It goes without saying that any scalable system must eventually re-balance workloads to prevent server overload. However, executing atomic actions across multiple independently failing and autonomous hosts is inherently more costly than executing them within a single host (see Section 9.2) so it is prudent to restrict executions to individual hosts as much as possible.

Caching can simultaneously lower data access latency and increase data access bandwidth, while preserving capacity on original data host(s) and their channel(s) [73]. Furthermore, it can do so transparently, maintaining the illusion of homogeneously accessible data with relatively good performance even as data is stored in a heterogeneous and possibly distributed fashion. This bears similarity to the way virtual memory in modern operating systems provides the illusion of uniform ranges of bytes in memory, while memory pages may in fact be swapped in and out from disk. In summary, caching can significantly increase data access performance in a manner transparent to those accessing the data.

A large body of work exists on distributed caching for file systems and the Internet [74, 75, 76]. For end-user browsing of web pages, temporary inconsistencies or staleness of data are rarely an issue. To ensure atomic and serializable execution, however, the illusion must be perfect; a transaction must never observe behavior that is inconsistent with serial execution. This is the challenge of *cache consistency.*

These issues were explored in research on *Distributed Shared Memory* (DSM) systems [77], seeking to extend virtual memory across a network of hosts, presenting them with a shared global address space. But the overhead of ensuring uniformly sequential consistency for all memory reads and writes proved too great, exacerbated by the problem of *false sharing* resulting from unrelated data co-residing on large-granularity data pages. Logically simple operations such as inserting an entry into a map, can become expensive to share at the raw memory level, when arrays must be reallocated or hash tables rehashed, for example.

Later systems [78, 79, 80] progressively relaxed consistency guarantees, requiring programmers to explicitly synchronize shared accesses with locks, semaphores or barriers, e.g. They also shifted from memory-based coherency to finer-granularity, higher-level type and object-based coherency [81, 82], blurring the distinction between DSM and distributed object systems such as Thor. Some early systems (and, in a category of their own, shared-memory "supercomputers") used specialized hardware and operating systems to accelerate communication and coherency operations, while later systems are based on networked clusters of commodity "shared-nothing" hosts.

Most DSM systems are geared towards high performance for large, scientific computations, treating fault-tolerance and availability as a secondary issue. They differ along many dimensions, such as whether it is predominantly data or computation that is shipped around, whether coherency is enforced through centralized or distributed means and whether all replicated copies of data are equal or whether there is a master copy to which other copies are subservient. The last point is important, because a thin line divides coherent caching from more general replication.

We find it desirable, in general, for master copies to be stored in secure data centers rather than end-user access devices. However, moving all computation from "dumb" access devices into remote servers may create a barrier for the performance and responsiveness of applications. Network latency and bandwidth will continue to be bound by geographic distances and the laws of physics. We believe that global, atomic consistency caching strikes a good balance between restrictive, total centralization and free-wheeling complete distribution. We compare our approach to existing atomic consistency caching algorithms at the end of Chapter 6.

In the "message-passing vs. shared-memory" argument, we side tend to with the latter. While the low-level optimization opportunities afforded by message passing may give it a performance edge in most cases, we believe that the intuitive abstraction provided by globally shared data portends its adoption for general applications and information systems. While global-scale systems can and are being built using highly asynchronous message-passing programming styles, the difficulties are significant and successes may be hard to replicate.

2.4 Name-based Routing and Multicast

Atomic Transfer is premised on switches rapidly detecting conflicts between requests and responses. While Chapters 3 and 4 treat requests and responses as abstract entities, subsequent chapters refine the model so a server's state is partitioned by data/variable names and conflicts between requests and responses imply that they have a name in common.

Our models assume that servers can multicast [83] responses to applications, that subscribe to the server. We could define our models in terms of end-to-end connections between servers and application hosts, but assuming multicast leads to more elegant models and allows more of the concurrency control burden to be shifted to the network, our original goal. Furthermore, multicast directly supports our cache architecture, as multiple caches can efficiently receive streams of responses updating their contents. However, we must make the relatively strong assumption that the multicast subsystem is reliable, delivering each response to all subscribers in the order sent, with no response omitted. Scaling reliable multicast is non-trivial, but numerous approaches have been proposed [84, 85, 86]. We leave integration of Atomic Transfer with these solutions to future work. We note that while Chapter 5 presents a refinement allowing safe dropping of response messages, it does so at a cost to performance and availability.

We go further still and assume that messages are forwarded based on the names they contain, not host network identifiers. Application hosts address transactions to the data names on which operations are to be invoked and each response is multicast to all receivers subscribed to names it affects. In effect, each name in the global data name space is a multicast topic, and applications subscribe to the names involved in their requests, ensuring that requests will meet any conflicting response in the network. By extension, data servers expose state as name / value pairs[5]. While implementations could instead resolve names to hosts before sending messages, this presentation makes our models simpler and allows us to focus on the properties of Atomic Transfer rather than the details of multicast routing.

Routing and forwarding based on application-level naming is not a new idea. Indeed, tuple-spaces [87, 88] may represent the ultimate in communication abstraction, with data transferred between processes based on predicate matching on tuple fields. Most forms of network communications, though, involve some form of *name resolution*, where a human-readable resource name is used to look up the network address(es) of the host(s) providing access to that resource. The primary method for name resolution in the Internet is the Domain Name System (DNS) [89], which maps hierarchical names such as `www.wustl.edu` to Internet IP addresses. There has been some interest in pushing resource discovery and name resolution deeper into the network

[5]More specifically, a name corresponds to an Abstract Data Type, providing a set of operations.

[90, 91, 92, 93, 94], to increase flexibility while raising the abstraction level away from network identifiers and towards identifiers meaningful to applications. This work explores the spectrum for the timing of name resolution, from late binding at message forwarding time to early binding, for example before connections are established. It also investigates the best division of labor between network elements and end-hosts. Many proposed systems use consistent hashing [95] for routing towards a name's home node, in *distributed hash tables* (DHTs) [96, 97, 98, 99].

Van Jacobson has recently espoused the view that "Content-Centric Networking" is the future of networking [100], arguing that globally named data should be the focus, not the networks used to transmit that data.

Our formal models abstract from routing and the internal structure of names and resource discovery is outside our scope. Chapter 8 does suggest an implementation where names are variable-length bit sequences and parts of names are hierarchical, with similar name suffixes indicating higher probability of resource co-location or proximity in the network. We find it desirable for hosts to be able to construct and send requests[6] without having to consult name resolution services. Also we find it important that names and their operations be abstract, to allow system components freedom in choosing their internal representation details. Encoding pointers or other machine-dependent data directly, as for example in DSM systems, reduces implementation flexibility and interoperability.

More generally, an abstract, name-based request/response protocol can serve as a useful abstraction barrier, enabling diverse, evolving programming languages, network architectures and host platforms to safely and efficiently share data with atomic guarantees. The success of protocols such as IP, TCP and HTTP suggests that conceptually clear network protocols are more likely than complex software libraries to be adopted as a common ground for diverse distributed systems. In an age where everything is networked, from powerful database servers to wireless earbuds, the only thing devices will generally have in common is a network protocol.

There has been considerable interest in publish/subscribe data dissemination systems [101] over the past decade. Earlier systems supported only topic-based subscriptions

[6]Possibly in user space, with minimal Operating System involvement.

[102, 103] but later systems allow *content-based subscriptions* [104, 105, 106] using predicates over the data contained in event notifications. A multitude of academic and commercial systems have been created. In our model, the only "event" of note is a change in the state of a data server, and the only notifications are responses generated by operation executions. We believe that programmers and system users are better served by state-centric abstractions than event-centric ones, as the latter precipitate the asynchronous, event-driven programming style. However, one can easily imagine state-update responses feeding into dissemination networks.

Total order multicast algorithms [107, 108, 109] seek to ease the development of distributed systems by providing powerful communication primitives guaranteeing causal or total ordering (and sometimes atomicity) of messages exchanged between a group of closely cooperating programs. This contrasts with state-centric models, where ordering guarantees and atomicity are primarily enforced on an end-to-end basis [110, 111]. Our approach is firmly in the second camp, with hosts interacting only through shared state and servers acting as the roots of individual multicast trees. In our model, state exists on its own beyond the scope of any program using it, and concurrency control may have to mediate accesses from arbitrary programs. Furthermore, whereas ordered multicast groups use rounds of message exchanges to agree on delivery order, we basically expect the network to preserve the order of messages multicast.

Expecting more than standard Internet functionality from global networks may become more realistic in the near future. Our approach is partly inspired by the promise of programmable switches and routers. While high-performance routers have traditionally used hardwired logic in Application Specific Integrated Circuits (ASICs) for their most demanding forwarding functions, the trend is towards performing them in software, using general-purpose specialized multiprocessors such as the (now defunct) Intel IXP line [112] or the proprietary 40-core Cisco Quantum Flow processor [113]. While currently found only in academia [114], open programmable router platforms may eventually become widely available. This could transpire, for example, if general-purpose or embedded multi-core processors acquire the features needed for high-performance router implementation, such as asynchronous memory access and high-performance I/O interfaces coping with multi-gigabit data rates. In the meantime, research can be carried out using commodity PC routers [115, 116] and overlay network testbeds [117].

Deployment of radically new network protocols in the Internet is currently a near-impossibility [118, 119]. It is significantly easier to deploy novel protocols within the confines of data centers, that have centralized control of network equipment and configuration. At the time of this writing, the notion of renting host and network resources from data centers has garnered much interest, under the moniker of *cloud computing*. Our algorithms are highly applicable to data center settings and fit well with the hierarchical network architectures they commonly employ. Such network topologies facilitate content-based forwarding and multicasting as well as flexibility in choosing "choke points" for concurrency conflict checking. The gateway switch into a subnetwork, for example, is well placed to detect conflicts between operation requests from (possibly high-latency) external requesters flowing into that sub-network. To summarize, data centers built from commodity programmable switches and networks would be a suitable foundation for scalable, atomic cloud computing infrastructures based on our approach.

2.5 Background: I/O Automata

This section gives a short review of the I/O Automata [120, 121] formalism, which we use to model and reason about our algorithms. Our summary below is adapted from [122].

An I/O automaton is an (infinite) state machine whose state transitions are *actions*. An I/O automaton *signature* S consists of a set of actions, denoted $acts(S)$, partitioned into *input actions*, *output actions* and *internal actions*, denoted $in(S)$, $out(S)$ and $int(S)$, respectively. Let $ext(S) = in(S) \cup out(S)$ be the *external actions* of S. An automaton a is a tuple (sig, $states$, $start$, $trans$, $tasks$), with sig an automaton signature, $states$ a (potentially infinite) set of states, $start$ a non-empty subset of $states$, $trans$ a *state-transition relation*, with $trans \subseteq states \times acts(sig) \times states$ and $tasks$ an equivalence relation on $ext(S)$. We abbreviate $acts(sig(a))$ as $acts(a)$, and similarly for in, out and so forth.

An *execution fragment* of a is a finite sequence s_0, π_1, s_1, π_2, $\ldots$, π_r, s_r or an infinite sequence s_0, π_1, s_1, π_2, $\ldots$ of alternating states and actions such that $(s_k, \pi_{k+1}, s_{k+1}) \in$ *trans*(a) for every $k \geq 0$. An *execution* is an execution fragment beginning in a start state. An execution α is *fair* if for each task partition C, α is finite and all actions in C are disabled in α's final state or α is infinite and there are either infinitely many occurrences of actions from C in α or infinitely many occurrences of states in which all actions in C are disabled. Let *execs*(a) and *fairexecs*(a) be the set of all executions and fair executions of a, respectively.

The *trace* of an execution α of a, denoted *trace*(α), is the subsequence of α consisting of all the occurrences of actions from *ext*(a). Any two finite execution fragments α, α' of a where α' begins with the last state of α may be concatenated (less the last state of α) to yield another execution fragment of a, denoted $\alpha \cdot \alpha'$. The occurrence of an action π in an execution or trace is called a π *event*.

An action $\pi \in$ *int*$(a) \cup$ *out*(a) is *enabled* in state $s \in$ *states* if there exists transition $(s, \pi, s') \in$ *trans*, for some state $s' \in$ *states*. Input actions are always enabled by definition, so for every $\pi \in$ *in*(a) and state $s \in$ *states* there is a tuple (s, π, s') for some $s' \in$ *states*. The actions in *in*$(a) \cup$ *out*(a) are called the *local* actions of a, and a is said to be *quiescent* in state s if none of its local actions are enabled in s.

A collection $\{a_i\}_{i \in I}$ of automata may be *composed* to form a new automaton a if the signatures of each pair $a_i \neq a_j$ are compatible, meaning that each internal or output action is under the control of a single automaton. Formally, a collection $\{S_i\}_{i \in I}$ of signatures (indexed by some countable set I) is *compatible* if for each pair S_i and S_j with $i \neq j$ we have $\mathit{int}(S_i) \cap \mathit{acts}(S_j) = \emptyset$, $\mathit{out}(S_i) \cap \mathit{out}(S_j) = \emptyset$ and each action is contained in finitely many sets $\mathit{acts}(S_i)$. The signature of the composed automaton a has $\mathit{out}(a) = \bigcup_{i \in I} \mathit{out}(a_i)$, $\mathit{int}(a) = \bigcup_{i \in I} \mathit{int}(a_i)$ and $\mathit{in}(a) = \bigcup_{i \in I} \mathit{in}(a_i) - \bigcup_{i \in I} \mathit{out}(a_i)$. The states of automaton a are defined as the Cartesian product of the states of its component automata, that is $\mathit{states}(a) = \prod_{i \in I} \mathit{states}(a_i)$. Similarly, $\mathit{start}(a) = \prod_{i \in I} \mathit{start}(a_i)$. *trans*$(a)$ is the set of triples (s, π, s') such that for all $i \in I$, if $\pi \in \mathit{int}(a_i)$ then $(s_i, \pi, s'_i) \in \mathit{trans}(a_i)$ otherwise $s_i = s'_i$, with s_i denoting the part of state s "belonging" to a_i. The task equivalence classes of the component automata become the equivalence classes of a, that is: $\bigcup_{i \in I} \mathit{tasks}(a_i)$.

Given an execution fragment α and some set of actions A we define the *projection* of α on A, denoted $\alpha|A$, as the subsequence of α comprised of all adjacent states and transitions π_r, s_r where $\pi_r \in A$. Similarly, for a trace β we define $\beta|A$ as the subsequence of β comprised of all actions in A. The *projection* $\alpha|a_i$ of an execution α of a composition automata a on one of its component automata a_i is defined as $\alpha|\mathit{acts}(a_i)$, with each state s_r replaced by the state of a_i in s_r. Similarly, the projection $\beta|a_i$ of a trace β of a is defined as $\beta|\mathit{ext}(a_i)$. It can be shown that executions and traces of a yield executions and traces of a_i when projected on a_i, for each $i \in I$. Conversely, given an execution α_i for each $i \in I$ and a sequence β of actions in $\mathit{ext}(a)$ such that $\beta|a_i = \mathit{trace}(\alpha_i)$ for each $i \in I$, there is an execution α of a such that $\mathit{trace}(\alpha) = \beta$ and $\alpha|a_i = \alpha_i$ for each $i \in I$. Furthermore, if β is a sequence of actions in $\mathit{ext}(a)$ such that $\beta|a_i \in \mathit{traces}(a_i)$ for each $i \in I$, then $\beta \in \mathit{traces}(a)$. These theorems enable modular reasoning about executions and traces of composite automata.

We describe the state transition relations of our automata using a mixture of pseudocode and formal expressions. Each output or internal action has a predicate characterizing the states when the action is enabled. The effects of an action on state are described as a collection of assignments to state components (*fields*) that occur together, atomically. We use the convention that v denotes the "old" or current value of a field v while v' denotes its "new" value, which takes effect in the automaton's next state. If a field is not mentioned in a action it is assumed to retain its previous value.

Chapter 3

Atomic Transfer

This Chapter introduces Atomic Transfer (AT), a new primitive in the network layer that can be used to prevent race conditions due a pair of messages crossing on a path of network switches. AT can be used, for example, to prevent an operation request message from crossing "on the wire" with an update notification message that renders the request invalid. This chapter defines the switches and channels comprising an atomic network but leaves end-hosts undefined, except for well-formedness conditions that they must uphold to ensure end-to-end atomic transfer.

In our system of discourse, host machines send *requests* to remote server host machines and receive *responses* in return, over a network of Atomic Transfer switches. A request can cause a state change on the server executing it, in which case the request's response notifies potential requesters about its effects. For example, a request may ask for the value of a variable to be changed and the corresponding response would notify of the variable's new value. The switches in an atomic network provide guarantees about request and response atomicity that are not provided by traditional network switches.

We present I/O Automata defining these switches and the atomicity guarantees they provide. We describe requests and response messages quite abstractly, leaving their semantics undefined. The only thing we need is a relation containing all pairs of requests and responses that *conflict*, in some abstract sense. Subsequent chapters present more concrete refinements, where a request invokes operations on a set of named variables and conflicts with responses notifying of changes to one or more of those variables. For example, an operation may specify that it reads a certain variable v. This puts it in conflict with any response notifying about a new value for v, caused by the execution of some earlier request. An *atomic switch* can detect this conflict as the request and response "meet" at the switch, by computing the intersection of request and response variable name sets. The switch can then take actions such as dropping the request, obviating the receiving end-host from detecting and handling the conflict. This preserves the goodput of server hosts, particularly during periods of heavy data contention, when many hosts concurrently send conflicting requests to the server. But to simplify the discussion and focus on the essential properties of Atomic Transfer, we use a completely abstract conflict relation in this Chapter.

Recall that this dissertation is limited in scope to single-server transactions, so each transaction executes independently on a particular server. Furthermore, we implicitly equate requests and response messages with network packets, that is: each request and response fits within a single network transmission unit. Section 9.2 in Chapter 9 on future work outlines an approach for removing these restrictions.

3.1 Atomic Switches and Atomic Transfer

We model the system as an undirected graph $S = (\mathit{NODES}, \mathit{CHANNELS})$, where *NODES* represent network nodes and *CHANNELS* represent bi-directional communication channels between them. The set of nodes is partitioned into the set *HOSTS* of computer hosts and the set *SWITCHES* of atomic switches. Each host is incident to exactly one node, which is a switch that we term the host's *home switch.* Observe that any pair of nodes have at most a single channel in common.

We lay out our basic definitions for networks and messages. Let M be the domain of all messages and let Q and R be disjoint subsets of M corresponding to *request messages* and *response messages*, respectively. A host can send a message $q \in Q$ containing an operation request to a destination host, which may send back a *response* $r \in R$ containing a response to the operation request. As a simple concrete example, request set Q could be the set of messages "$v := x$" requesting the value of some variable v be set to some value x, while response set R could be the set of messages "$v =$ x" notifying that a variable v has received the new value x. We will call a host a issuing a request q *the requester of* q and the host b receiving request q and generating a response r the *responder of* q / *responder of* r. We also say that host a *calls* b.

Incident nodes can exchange data messages directly through their common channel, while non-incident nodes can exchange messages over a path of channels and switches. As in any store-and-forward network, a switch must forward each message it receives on a path leading towards its destination. For each switch i let $CHANNELS_i$ denote the subset of channels in $CHANNELS$ that are incident to i and let $qHop_i$ denote a function $Q \rightarrow CHANNELS_i$, mapping any request q to a channel that leads to the destination host of q. Intuitively, $qHop_i$ corresponds to the forwarding table that switch i uses to move requests towards their destinations. We will restrict our discussion in this dissertation to requests that are executed by a single destination host, so let $\mathit{destination}$ denote a function $Q \rightarrow HOSTS$, mapping any request to its destination host. In Chapter 9 on future work, we discuss the relaxation of $\mathit{destination}$ to a general relation, allowing *multi-server* request to be forwarded to multiple destination hosts.

Let $fp(q)$ denote the *forwarding path* of a request q sent from a host a to a destination host b, namely: the sequence of switches s_1, s_2, ..., s_k such that s_1 and s_k are the home switches of a and b, respectively, and for each $i \in \{1 \,..\, k$ - $1\}$ we have $qHop_i(q) = (s_i,\ s_{i+1})$. For notational clarity, we will often abbreviate a channel $(s_i,\ s_{i+1})$ to $(i,\ i{+}1)$ when it is clear from context that the channel connects two switches on a particular path. Note that a forwarding path, if it exists, is uniquely determined for a particular request and switch, since a request's next hop is a function of the request at each switch. We will assume, for convenience, that there exists a total function sender: $M \rightarrow HOSTS,$ mapping each message m to the host a that originally sent it (the host a with the send$(m)_{c,a}$ event causing each receive of m, in the model below).

Requests are unicast from a requester to its responder, while responses are multicast to all *subscribed hosts*, including the requester. For each switch i let $rHops_i$ be a relation $R \times CHANNELS_i$, relating a response r to the channel(s) that lead toward the host(s) that should receive r. We assume that the graph induced by $rHops_i$ relations is acyclic. Intuitively, the $rHops$ relations correspond to multicast subscriptions to state updates in hosts that occur in response to request execution. We will sometimes refer to the hosts subscribed to a response r as the *subscribers of* r. Formally:

Definition 3.1 *A host a is subscribed to a response r if there exists a sequence of switches $(s_1,\ s_2,\ ...,\ s_k)$ such that s_1 and s_k are the home switches of a and* $\mathit{sender}(r)$, *respectively,* $(r,(a,1)) \in$ *rHops*$_1$ *and for each* $i \in \{2\ ..\ k\}$ *we have* $(r,(i-1,i)) \in$ *rHops*$_i$.

An *Atomic Transfer* (AT) is the transfer of a response message along a path of switches and channels guaranteeing that if a request traveling the path in the opposite direction *conflicts* with the response, then this will be detected and can be handled. A request q could, for example, specify that the current values of some variables v_1 and v_2 were assumed as q was issued. If q encountered a response r on its way to *destination*(q) notifying of a change in the value of v_1, then q is in conflict with r and cannot be executed, since it has been invalidated by r. The handling of conflicts may vary, but can include such actions as dropping the request, modifying it or rerouting it. In our initial model, conflicting requests are simply dropped.

Note the asymmetric handling of requests and responses. Conflicting request messages may get dropped, in cases when they cannot be executed by their destination responders anyway. Responses are reliably transported to their subscribers, as they reflect a completed operation and actual state change in the system.

Atomic Transfer is implemented in switches using an acknowledgement scheme. Let A be a sub-domain of M disjoint with both Q and R, corresponding to acknowledgement messages (ACKs). Let *ack* be a total function $M \rightarrow A$ mapping a message to its unique ACK. Conversely, let *message* be a function $A \rightarrow M$ mapping an ACK to the unique message it acknowledges, so for any $m \in M$ we have *message*(*ack*$(m)) = m$.

A request $q \in$ Q and response $r \in R$ *conflict* if they are related by a *conflict relation*, which we denote by $(q,r) \in$ *conflicts* or *conflicts*$(q,\ r)$. We place the following restriction on the conflict relation and forwarding relations:

Definition 3.2 *Conflict Locality: for any pair of messages* $q \in Q$ *and* $r \in R$ *where conflicts*(q,r) *and any pair of switches* $i,\ j \in$ *NODES: qHop*$_i(q) = (i,j) \Rightarrow (r,(i,j)) \in$ *rHops*$_j$.

In other words: if a switch i forwards a request q on to channel (i,j), then switch j will forward each responses r that conflicts with q back on channel (i,j). This ensures that conflicting requests and responses “meet” somewhere in the network so the conflict is detected. The restriction implies that a host must be subscribed to all possible responses to a request q as a precondition for sending q into the network.

The definition does not require all requests from a particular host to follow the same path to a particular destination; by conflict locality, any conflicting response will be sent back along *all* these paths. However, it is convenient for our discussion to restrict all requests (and thus responses) between a pair of hosts to some unique path. This allows us to unambiguously refer to *the forwarding path* fp_{ab} from any node a to another node b, which is either uniquely determined or does not exist.

Assumption 3.1 *Unique forwarding path: for any pair of requests $q_1, q_2 \in Q$ and any node $i \in NODES$: $destination(q_1) = destination(q_2) \Rightarrow qHop_i(q_1) = qHop_i(q_2)$.*

3.2 FIFOChannel$_i$, Channel$_c$ and AtomicSwitch$_i$

We provide I/O Automata for network channels and atomic switches below. We only provide the signature (external events) of end-hosts, but will later specify a set of well-formedness conditions for their behaviors. We use the dot operator $\cdot$ to denote concatenation to the end of a sequence or queue. Given a queue or sequence Q, we use $head(Q)$ to denote the first element of Q and $tail(Q)$ to denote the queue or sequence resulting from removing $head(Q)$ from Q. We abuse notation and use $Q \setminus C$ to denote the queue or sequence resulting from removing from queue or sequence Q all elements that are members of set C. Also, we take $x \in s$ to mean that x appears at least once in queue or sequence s. Similarly, let $x \notin s$ mean x does not appear in queue or sequence s.

We leave the precise action that a switch takes upon detecting a conflict unspecified for now. A typical implementation might discard the conflicting request, possibly sending an exception notification back to the sender.

FIFOChannel$_{i,j}$

Models a unidirectional, reliable FIFO channel, connecting nodes i and j.

State:

$outQueue_{i,j}$: a FIFO queue of messages from i to j currently in transit on the channel.

Input actions:

send$(m \in M)_{c,i}$
Effect:
    outQueue$'_{i,j}$ = outQueue$_{i,j} \cdot m$

Output actions:

receive$(m \in M)_{c,j}$
Precondition:
    head(outQueue$_{i,j}$) $= m$
Effect:
    queue$'_{i,j}$ = *tail*(outQueue$_{i,j}$)

Channel$_c$

Models a bidirectional, reliable FIFO channel for edge $c = (i, j)$ in *CHANNELS*, connecting nodes i and j. We define it as the I/O Automata composition of $FIFOChannel_{i,j}$ and $FIFOChannel_{j,i}$.

Host$_i$

The signature of host i, that sends and receives messages, including requests and responses.

Input actions:

receive$(m \in M)_{c,i}$

Output actions:

send$(m \in M)_{c,i}$

AtomicSwitch$_i$

Models the behavior of an Atomic Switch.

State:

```
for each $c \in CHANNELS_i$
    $outQueue_c$: queue of messages outbound on $c$, initially empty
for each $c \in CHANNELS_i$
    $responses_c$: set of non-acknowledged responses sent or outbound on $c$, initially empty
```

Input actions:

```
receive$(q \in Q)_{c,i}$
Effect:
    // if the request conflicts with a response we're buffering
    if $\exists r \in responses_c$ such that $conflicts(q, r)$
        // do not enqueue it, but handle it somehow
        $handleConflict()$
    // else: enqueue the request on the appropriate output channel
    else let $d = qHop_i(q)$ in
        $outQueue'_d = outQueue_d \cdot q$
```

```
receive$(r \in R)_{c,i}$
Effect:
    // enqueue an ACK back
    $outQueue'_c = outQueue_c \cdot ack(r)$
    for each $d \in CHANNELS_i$ such that $(r, d) \in rHops_i$
        // forward response
        $outQueue'_d = outQueue_d \cdot r$
        // remember it, at outbound port
        $responses'_d = responses_d \cup \{r\}$
```

```
receive$(a \in A)_{c,i}$
Effect:
    // the acking node is now responsible
    $responses'_c = responses_c \;\; message(a) \}$
```

Output actions:

```
send$(m \in M)_{c,i}$
Precondition:
    $head(outQueue_c) = m$
Effect:
    $outQueue'_c = tail(outQueue_c)$
```

The receive$(q \in Q)_{c,i}$ action receives a request from channel c and enqueues it for forwarding, but only if it doesn't conflict with a response already enqueued for forwarding on c. This behavior is the foundation for Atomic Transfer.

The receive$(r \in R)_{c,i}$ action receives a response and enqueues it for forwarding on one or more outbound channels, as per $rHops_i$. It also remembers the response at the outbound channels, so that conflicts with future requests arriving on those channels can be detected. It enqueues an ACK back to the sender, to signal that the switch has now assumed responsibility for detecting conflicts with the response.

The send($m \in M)_{c,i}$ action sends the next message queued for channel c and removes it from the outbound queue of c. It does *not* remove a sent response from the $\mathit{responses}_c$ set; removing a response at this point would allow the reception and forwarding of a conflicting request currently in transit on channel c, for example. The response cannot be removed from $\mathit{responses}_c$ until an ACK for its reception has been received from the other node incident to c, signaling that the node has taken responsibility for detecting requests that conflict with the response.

The receive($a \in A)_{c,i}$ action receives a response acknowledgement from channel c and removes the corresponding response from the response_c set. Note that these ACKs are for atomic transfer alone, not reliability; Channels are assumed to reliably transfer all messages, including requests. An implementation using unreliable channels could use a unified acknowledgement scheme for atomicity, reliability and possibly flow-control, but our model abstracts from these considerations. Similarly, an implementation could use message sequence numbers to obviate the need for ordered delivery by the network.

3.3 Properties and Proofs

The Atomic Transfer theorem says, approximately, that once a responder has transmitted a response, it is impossible for a requester to successfully transmit requests that conflict with the response before receiving it. More precisely, we show that once a response r has been injected into an atomic network, it will only deliver a conflicting request q traveling in the opposite direction if the requester of q has acknowledged r.

For any pair of event occurrences e_1 and e_2 in an execution or trace E, we use the notation $e_1 <_E e_2$ (read "e_1 occurs ahead of e_2 in E") to denote that the occurrence of e_1 in E is before the occurrence of e_2 in E. We also say $e_1 <_E e_2$ if e_1 occurs in E but there is no occurrence of e_2 in E. The intuition for the latter case is that e_2 would have to occur after e_1 in any sequence extending E. Note that $<_E$ is irreflexive and transitive and that $\neg(e_1 <_E e_2)$ implies $e_2 <_E e_1$. We use $e_1 <_E e_2 <_E e_3 ... <_E e_n$ as shorthand for $e_1 <_E e_2 \wedge e_2 <_E e_3 \wedge \ldots e_{n-1} <_E e_n$.

Our first theorem shows that a network forwarding path composed of AtomicSwitch and Channel automata provides atomic transfer. Our second theorem shows atomic transfer end-to-end between requester and responder hosts, given certain well-formedness conditions on the hosts. We begin with two simple lemmas, regarding the behavior of individual switches and channels. They essentially say that switches do not invent new requests and that channels do not invent new messages.

Lemma 3.1 *(No spontaneous switch requests): In any execution or trace E, for every send$(q \in Q)_{d,i}$ in E where $i \in SWITCHES$ there is a distinct receive$(q)_{c,i}$ earlier in E, for some channel $c = (i, j)$ incident to i. We say that the earlier event e_1 causes the later event e_2, denoted by $e_1 \rightarrow_E e_2$.*

Proof: From the preconditions of AtomicSwitch$_i$ we see that a message m must be on queue $outQueue_d$ of i before send$(m \in M)_{d,i}$ can occur. The only way for a request or response message $m \in Q \cup R$ to be added to $outQueue_d$ is via a receive$(m)_{c,i}$ event for some channel $c \in CHANNELS_i$, so such an event must occur earlier in the trace. Since the send$(m)_{d,i}$ removes m from $outQueue_d$, each receive$(m)_{c,i}$ event can cause at most one send$(m)_{d,i}$, so there must be at least as many receive$(m)_{c,i}$ events as there are send$(m)_{d,i}$ events □

Lemma 3.2 *(no spontaneous channel sends): In any execution or trace E, for every receive$(m \in M)_{c,j}$ in E where $c \in CHANNELS$ there is a distinct send$(m)_{c,i}$ event earlier in E. We say that the send event causes the receive event, denoted by send$(m)_{c,i} \rightarrow_E$ receive$(m)_{c,j}$.*

Proof: From the preconditions of $Channel_c$ we see that message m must be on $outQueue_{i,j}$ in c before receive$(m)_{c,j}$ can occur. The only way for m to be added to $outQueue_{j,i}$ is via a send$(m)_{c,i}$ event. Since the receive$(m)_{c,j}$ removes m from $outQueue_{i,j}$, each send$(m)_{c,i}$ event can cause at most one receive$(m)_{c,j}$, so there must be at least as many send$(m)_{c,i}$ events as there are receive$(m)_{c,j}$ events □

We extend $\rightarrow_E$ to its reflexive, transitive closure. Hence, $e \rightarrow_E e$ holds for any send or receive event e and if there exists a subsequence of send and receive events $\alpha = s_1$, r_1, s_2, r_2, ..., s_n, r_n in some trace E such that $s_j \rightarrow_E r_j$ for $j \in \{ 1, 2, ..., n \}$ and

$r_j \to_E s_{j+1}$ for $j \in \{ 1, 2, ..., n\text{-}1 \}$, we say that $e_1 \to_E e_2$ for any pair of events e_1, e_2 $\in \alpha$ where $e_1 <_E e_2$. Contrariwise, if $\neg(e_1 \to_E e_2)$, so no such subsequence exists, we say that e_1 *does not cause* e_2, denoted by $e_1 \not\to_E e_2$. The following Lemma highlights the relationship between $\to_E$ and $<_E$:

Lemma 3.3 *(Time ordering and causality): If e_1 and e_2 are distinct events in an execution or trace then $e_1 \to_E e_2 \Rightarrow e_1 <_E e_2$.*

Proof: if e_1 and e_2 are related non-transitively by $\to_E$, then the definition of $\to_E$ implies that e_1 occurs earlier in E than e_2, so $e_1 <_E e_2$. If they are transitively related by $\to_E$, then for each pair of events (e_i, e_j) in the sequence of events S leading from e_1 to e_2 occurs we have $e_i <_E e_j$, so $e_1 <_E e_2$, by transitivity of $<_E$ $\square$

Note that while $\to_E$ may resemble Lamport's happens-before relation [123], it is different. It relates a send (receive) event precisely to the receive (send) events that cause it, but not to unrelated events occurring on the same switch. Also note that while $\to_E$ is currently defined for request messages only, we extend the relation to other message types in later chapters as needed.

Switches along a network path share the responsibility for detecting conflicts along the path. We define the concept of *responsibility intervals*, corresponding to the period in an execution during which a switch buffers a response and conflict-detects incoming requests against it.

Definition 3.3 *A state s of AtomicSwitch$_j$ is in a responsibility interval of r in j with respect to a channel $d \in$ rHops$_j(r)$, denoted $s \in$ resp-interval$(r)_{d,j}$, exactly if $r \in$ j.responses$_d$ in state s. We say for an event e in X that $e \in$ resp-interval$(X,\ r)_{d,j}$ if $s_e \in$ resp-interval$(r)_{d,j}$, where s_e is the state preceding e in X. It is clear from the definition of AtomicSwitch$_i$ that receive$(r)_{c,j} <_X e <_X$ receive$(\text{ack}(r))_{d,j}$, where c is the channel by which r is received at j.*

From the definition of the receive$(q \in Q)_{c,i}$ action of AtomicSwitch$_j$, we immediately have the following:

Corollary 3.1 *(Conflicting receives in responsibility intervals never cause sends): In any execution X in which AtomicSwitch$_j$ appears, if receive$(q \in Q)_{d,j} \in$ resp-interval$(X, r)_{d,j}$ for some $r \in R$ and conflicts$(q,\ r)$ then receive$(q)_{d,j} \not\rightarrow_X$ send$(q)_{c,j}$, for any channel $c \in CHANNELS_j$.*

The first Atomic Transfer Theorem is stated for a path of switches and channels, with no reference to the behavior of hosts; this is deferred to Theorem 3.2.

Let AS Let be the I/O Automaton composed of an AtomicSwitch$_i$ automaton for each switch $s_i \in SWITCHES$ and a Channel$_c$ automaton for each channel $c = (i, j) \in CHANNELS$. Let s_1, s_2, ..., s_n denote the switches of fp_{ab}, the forwarding path from host $a \in HOSTS$ to some other host $b \in HOSTS$. We will use s_k to denote the k-th switch on the path, for any $1 \leq k \leq n$. Note that requests travel from lower-indexed switches to higher-indexed switches, while responses travel from higher-indexed switches to lower-indexed switches. We have a straightforward Lemma relating the responsibility intervals of adjacent switches in fp_{ab}.

Lemma 3.4 *(Overlap of responsibility intervals): In any $X \in$ execs(AS), events send$(r)_{(k,k+1),k+1}$, receive$(r)_{(k,k+1),k}$ and send$(ack(r))_{(k,k+1),k}$ are all in resp-interval$(X, r)_{(k,k+1),k+1}$.*

Proof: the first event is caused by the receive$(r)_{(k+1,k+2),k+1}$ event defining the beginning of *resp-interval*$(X,\ r)_{(k,k+1),k+1}$, the first causes the second, the second causes the third and the third causes the receive$(ack(r))_{(k,k+1),k+1}$ that defines the end of *resp-interval*$(X, r)_{(k,k+1),k+1}\square$

The lemma makes it explicit that the responsibility intervals on adjacent switches for a response r overlap in any execution X, since receive$(r)_{(k,k+1),k} <_X$ receive$(ack(r))_{(k,k+1),k+1}$.

Theorem 3.1 says that if a switch i on path fp_{ab} receives request $q \in Q$ via fp_{ab} in an interval after a switch j further down fp_{ab} receives a conflicting response r but before i receives the ACK for r, then request q is *dropped* and never delivered by j. While the proof could appeal to Conflict Locality directly, a slightly weaker version of it suffices, referring only to switches on path fp_{ab}. It is clear from our definition that general Conflict Locality implies Path Conflict Locality for all forwarding paths. Formally:

Definition 3.4 *For any $q \in Q$, fp(q) has Path Conflict Locality for q if for any $r \in R$ where conflicts(q, r) and any pair of switches s_k and s_{k+1} on fp$_{ab}$: qHop$_k(q) = (k, k+1) \Rightarrow (r, (k, k+1)) \in$ rHops$_{k+1}$.*

Theorem 3.1 *(Path Atomic Transfer): For all $X \in$ execs(AS), all $i, j \in [1, n]$ where $i \leq j$ and all $r \in R$ and $q \in Q$ where fp(q) has Path Conflict Locality: receive$(r)_{(j,j+1),j} <_X$ receive$(q)_{(i-1,i),i} <_X$ receive$(ack(r))_{(i-1,i),i} \wedge$ conflicts$(q, r) \Rightarrow$ receive$(q)_{(i-1,i),i} \not\rightarrow_X$ send$(q)_{(j,j+1),j}$.*

Proof: Assume for contradiction that there exists some $X \in$ *execs*(AS) and some $r \in R$, $q \in Q$ where *conflicts*(q, r) such that receive$(r)_{(j,j+1),j} <_X$ receive$(q)_{(i-1,i),i} <_X$ receive$(ack(r))_{(i-1,i),i}$ and receive$(q)_{(i-1,i),i} \rightarrow_X$ send$(q)_{(j,j+1),j}$, where $i, j \in [1, n]$ and $i \leq j$. Let I be the execution interval of X beginning after receive$(r)_{(j,j+1),j}$ and ending after the receive$(ack(r))_{(i-1,i),i}$.

Since receive$(q)_{(i-1,i),i} \rightarrow_X$ send$(q)_{(j,j+1),j}$ then for each switch s_k on *fp*$_{ab}$: (q, (k, k+1)) $\in$ *qHop*$_k$. Similarly, since *conflicts*(q, r) then by the Path Conflict Locality of *fp*$_{ab}$ for q, (r, (k-1, k)) $\in$ *rHops*$_k$, that is: s_k forwards r along *fp*$_{ab}$ in the direction opposite to q. Also, by the definition of $\rightarrow$ and the construction of AS there must be a receive$(q)_{(k-1,k),k}$ event and a send$(q)_{(k,k+1),k}$ event in I for every $i \leq k \leq j$, where receive$(q)_{(k-1,k),k} \rightarrow_I$ send$(q)_{(k,k+1),k}$ and send$(q)_{(k-1,k),k-1} \rightarrow_I$ receive$(q)_{(k-1,k),k}$. We show that one of these events is not in I, which contradicts receive$(q)_{(i-1,i),i} \rightarrow_I$ send$(q)_{(j,j+1),j}$.

We define *q-switch*(q, X) as the index of the latest switch on *fp*(q) to receive a request $q \in Q$ in any execution interval X, or precisely: the largest k such that receive$(q)_{(k-1,k),k} \in X$, or i-1 if there is no such event in X. We similarly define *r-switch*(r, X) as the index of the latest switch on *fp*(q) to receive response $r \in R$, or precisely: the lowest k such that receive$(r)_{(k,k+1),k} \in X$.

We argue that the request q received on s_i and the response r received by s_j must meet somewhere along path *fp*$_{ij}$. Since each receive$(q)_{(k,k+1),k+1}$ is caused by an earlier send$(q)_{(k,k+1),k}$ event, a receive(q) event increases the value of *q-switch* by at most 1. By similar reasoning, a receive(r) event can decrease the value of *r-switch* by at most 1. Furthermore, since Q and R are disjoint, no single event alters both *q-switch* and *r-switch*. Since *q-switch*(q, λ) = i-1 but *q-switch*(q, I)= j, *q-switch* takes on every value in $[i, j]$ during I. Since *r-switch*(r, I') $\in [i, j]$ for any prefix I' of I and is lowered by at most 1 by any one event, there must be some prefix I_e of I ending with event e after which *q-switch*(q, I_e) = *r-switch*(r, I_e) = k, for some $k \in [i, j]$. Let I'_e be the prefix of I up to but not including e. There are two possible cases for event e (see figure 3.1):

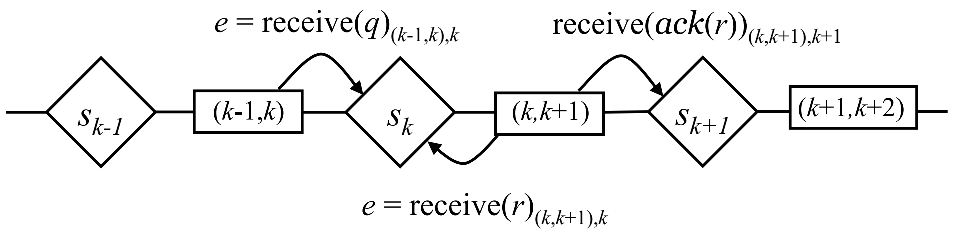

e = receive$(q)_{(k\text{-}1,k),k}$
receive$(ack(r))_{(k,k+1),k+1}$
$s_{k\text{-}1}$
$(k\text{-}1,k)$
s_k
$(k,k+1)$
s_{k+1}
$(k+1,k+2)$
e = receive$(r)_{(k,k+1),k}$

Figure 3.1: The meeting of a request and response

1. e = receive$(q)_{(k-1,k),k}$, so e increased *q-switch* from k-1 to k. Since *r-switch* = k, receive$(r)_{(k,k+1),k} \in I'_e$ but receive$(r)_{(k-1,k),k-1} \notin I'_e$. By Lemma 3.4, $e \in$ *resp-interval*$(I,\ r)_{(k-1,k),k}$, so by Corollary 1, receive$(q)_{(k-1,k),k} \not\rightarrow_I$ send$(q)_{(k,k+1),k}$, which is a contradiction.

2. e = receive$(r)_{(k,k+1),k}$, so e decreases *r-switch* from k+1 to k. Observe that this case only occurs for $k < j$. By Lemma 3.4, $e \in$ *resp-interval*$(I,\ r)_{(k,k+1),k+1}$. Since *q-switch* = k, receive$(q)_{(k,k+1),k} \in I'$ and the $ack(r)$ enqueued by receive$(r)_{(k,k+1),k}$ is behind q in $s_k.outQueue_{(k,k+1)}$. By the FIFO property of $s_k.outQueue_{(k,k+1)}$ and channel $(k, k+1)$, receive$(q)_{(k,k+1),k+1} <_I$ receive$(ack(r))_{(k,k+1),k+1}$, so receive$(q)_{(k,k+1),k+1} \in$ *resp-interval*$(I,\ r)_{(k,k+1),k+1}$. By Corollary 1, receive$(q)_{(k,k+1),k+1} \not\rightarrow_I$ send$(q)_{(k+1,k+2),k+1}$, which is a contradiction □

Reversing the implication of Theorem 3.1, we obtain: For all $i,\ j \in [1,\ n]$ where $i \leq j$, $X \in$ *execs*(AS) and $r \in R$, $q \in Q$: receive$(q)_{(i-1,i),i} \rightarrow_X$ send$(q)_{(j,j+1),j} \Rightarrow$ $\neg$*conflicts*$(q,\ r)$ $\vee$ receive$(ack(r))_{(i-1,i),i} <_X$ receive$(q)_{(i-1,i),i}$ $\vee$ receive$(q)_{(i-1,i),i} <_X$ receive$(r)_{(j,j+1),j}$.

The first two clauses in the disjunction give us what we want: for any request and response, either the first switch receives an ACK for the response ahead of the request or else the request and response do not conflict. But if neither of these cases applies, the third clause reminds us that a request received at the first switch before a conflicting response is received at the last switch can in fact be forwarded along whole path and sent by the last switch. Hence, a responder host can receive a request conflicting with a response the responder has created (and possibly sent) but which has not yet been received by the responder's home switch. The next section closes this loophole.

Note that the proof makes no mention of requests other than q and only considers a response r if it conflicts with q. In our model, other requests and responses that do not conflict with q have no effect on whether q is delivered or not. This highlights the fact that Atomic Transfer allows all concurrency permitted by the conflict relation to take place; non-conflicting requests and responses do not interact. We state this insight as a corollary.

Corollary 3.2 *(Independence of non-conflicting requests and responses): let X_1, $X_2 \in \text{execs}(AS)$ be two executions such that no response in X_1 conflicts with any request in X_2, and no response in X_2 conflicts with any request in X_1. Then any interleaving of executions X_1 and X_2 is in* $\text{execs}(AS)$.

3.4 End-to-End Atomicity

Theorem 3.1 is stated in terms of the forwarding path of requests and responses along switches and message channels. We now state two fairly unrestrictive well-formedness conditions on requesters and responders and show that these ensure end-to-end Atomic Transfer. For requesters, we require that they only acknowledge responses they've actually received.

Definition 3.5 *ACK Well-Formedness (no spontaneous ACKs) An execution or trace E is ACK Well-Formed for a requester $a \in HOSTS$ if for every $send(\text{ack}(r \in R))_{d,a}$ in E there is a distinct $receive(r)_{d,a}$ event earlier in E, where d is the channel incident* *to a. We include these events in the $\rightarrow_E$ causes relation, and say receive$(r)_{d,a} \rightarrow_E$ send$(ack(r))_{d,a}$.*

We will furthermore assume that for any execution or trace E there exists a partial function from the requests received at a responder $b \in HOSTS$ to the responses sent by b, defining the pairs of requests and responses (q, r) such that q *is the response to* r. This function captures how requester implementations recognize the responses to their own requests, using requester identifiers and request sequence numbers, for example. If the function maps q to r then the receive of q at $b = destination(q)$ must precede the sending of r in E. We include events receive$(q)_{(n,n+1),b}$ and send$(r)_{(n,n+1),b}$ in the causes relation and say receive$(q)_{(n,b),b} \rightarrow_E$ send$(r)_{(n,b),b}$. The function is necessarily partial, as some requests are dropped and never cause a response. We say that a request that causes a response is *successful*.

The well-formedness safety condition for responders says that a responder must check incoming requests for conflicts with the responder's recently created responses. More precisely: the responder must drop any request q received after the event e_r enqueuing a response r that conflicts with q but before the responder receives the ACK for r from its home switch. To model implementations that execute requests sequentially in the order received, we can use the event receiving a request as event e_r, assuming that event executes the request and enqueues its response. However, framing the discussion more generally in terms of a (possibly internal) enqueueing event e_r permits implementations to choose the order in which they execute their set of received but not yet executed requests. Furthermore, it permits modeling of responders that generate responses spontaneously, without external requests. This can be used to model sources of original data, such as sensors or user input devices, as well as active computation processes in responders.

A downside of using e_r is that it refers to the "enqueuing" of a response, a somewhat vague notion when discussing an unknown responder implementation. But all we need to know about the e_r is that it irrevocably commits to sending r, ahead of the response to any request received after e_r. Formally:

Definition 3.6 *Event e_r is an event that enqueues a response $r \in R$ at a responder b in an execution or trace E if e_r follows a send$(r)_{(n,b),b}$ event in E and for every*

$q \in Q$: $e_r <_X receive(q)_{(n,b),b} \Rightarrow send(r)_{(n,b),b} <_X send(r_q)_{(n,b),b}$ where r_q is a response such that $receive(q)_{(n,b),b} \rightarrow_E send(r_q)_{(n,b),b}$ and n is the home switch of b.

Responder Well-Formedness defines the responsibility of a responder for conflict-checking requests against responses recently created by the responder. Formally:

Definition 3.7 *Responder Well-Formedness (responder checks unacknowledged responses). For any $X \in \text{execs}(AS)$, X is well-formed for a responder host b with home switch n if for each $r \in R$ enqueued by an event e_r at b and each $q \in Q$ received by b: $e_r <_X receive(q)_{(n,b),b} <_X receive(\text{ack}(r))_{(n,b),b} \wedge \text{conflicts}(q,r) \Rightarrow receive(q)_{(n,b),b} \not\rightarrow_X send(r_q)_{(n,b),b}$, for any $r_q \in R$.*

Note that the condition does not restrict the processing order of two requests q_1 and q_2 when the response to neither request conflicts with the other, allowing responder implementations to process such requests concurrently. Also note that if b receives a request q_r that does not cause a response (because it conflicts and is dropped) then Responder Well-Formedness holds vacuously and the reception of q_r does not impact other requests. Hence, in the case when the responses to two requests q_1 and q_2 do (mutually) conflict, the responder can freely choose which one to execute and which one to drop. For example, a highly concurrent server implementation, running on a modern multiprocessor, might preserve well-formedness through some internal form of concurrency control, using locks, transactional memory [42] or even Atomic Transfer! A sequential responder, by contrast, can preserve well-formedness in a straightforward way. For example, it can maintain a set of generated but un-acknowledged responses, similar to *responses* sets on switches and only execute requests that do not conflict with any response in that set. The server host models of Chapter 4, for example, use this approach.

We have the following extension of Theorem 3.1, stating that if a requester a sends a request on path $fp_{ab} = s_1, s_2, ..., s_n$ after responder b receives some request causing a conflicting response but before a itself sends an ACK for that response, then the request will not cause a response. In other words: if the request and a conflicting response cross in transit, the request is dropped.

Theorem 3.2 *(End-To-End Atomicity): For all $X \in execs(AS)$ where X is well-formed for hosts a, $b \in HOSTS$, each $r \in R$ enqueued by an event e_r at b and all $q \in Q$: $e_r <_X receive(q)_{(n,b),b} \wedge send(q)_{(a,1),a} <_X send(ack(r))_{(a,1),a} \wedge conflicts(q,r)$ $\Rightarrow send(q)_{(a,1),a} \not\to_X send(r_q)_{(n,b),b}$, for any $r_q \in R$.*

Proof: Let $X \in execs(AS)$ be an execution that is well-formed for hosts a, $b \in HOSTS$ and let $r \in R$ be enqueued by an event e_r at b and let $q \in Q$ be a request such that $e_r <_X \text{receive}(q)_{(n,b),b} \wedge \text{send}(q)_{(a,1),a} <_X \text{send}(ack(r))_{(a,1),a} \wedge conflicts(q,$ $r)$. We must show that $\text{send}(q)_{(a,1),a} \not\to_X \text{send}(r_q)_{(n,b),b}$, for any $r_q \in R$. We separate the cases where request q is received at some switch on fp_{ab} before s_n receives r or after s_n receives r. Let *R-RCV-FIRST* denote $\exists g \in [1,\ n] : \text{receive}(r)_{(n,b),n} <_X \text{receive}(q)_{(g-1,g),g}$.

1. If *R-RCV-FIRST* is false, then $\text{receive}(q)_{(n-1,n),n} <_X \text{receive}(r)_{(n,b),n}$. Since $\text{receive}(r)_{(n,b),n}$ enqueues ack(r), By FIFO we have $\text{receive}(q)_{(n,b),b} <_X \text{receive}(ack(r))_{(n,b),b}$, so we have $e_r <_X \text{receive}(q)_{(n,b),b} <_X \text{receive}(ack(\text{r}))_{(n,b),b}$, and by Responder Well-Formedness, $\text{receive}(q)_{(n,b),b} \not\to_X \text{send}(r_q)_{(n,b),b}$, for any $r_q \in R$.

2. If *R-RCV-FIRST* is true, let k be the latest (greatest) k such that $\text{receive}(r)_{(n,b),n}$ $<_X \text{receive}(q)_{(k-1,k),k}$. If there is no such k then q is not received at the first switch and the Theorem holds trivially. Otherwise, we show that $\text{receive}(q)_{(k-1,k),k}$ $<_X \text{receive}(ack(r))_{(k-1,k),k}$.

   For $k = 1$, $\text{send}(q)_{(a,1),a} <_X \text{send}(ack(r))_{(a,1),a}$ directly implies $\text{receive}(q)_{(a,1),1} <_X \text{receive}(ack(r))_{(a,1),1}$, as required. For $k > 1$ (see figure 3.2) observe that by the definition of k, $\text{receive}(q)_{(k-2,k-1),k-1} <_X \text{receive}(r)_{(n,b),n}$, so we have $\text{receive}(q)_{(k-2,k-1),k-1} <_X \text{receive}(r)_{(n,b),n} <_X \text{receive}(q)_{(k-1,k),k}$. By Lemma 3.3, $\text{receive}(q)_{(k-2,k-1),k-1} <_X \text{receive}(r)_{(k-1,k),k-1}$, so by switch FIFO and ACK well-formedness, $\text{send}(q)_{(k-1,k),k-1} <_X \text{send}(ack(r))_{(k-1,k),k-1}$ and by channel FIFO, $\text{receive}(q)_{(k-1,k),k} <_X \text{receive}(ack(r))_{(k-1,k),k}$. Hence we have $\text{receive}(r)_{(n,b),n} <_X \text{receive}(q)_{(k-1,k),k} <_X \text{receive}(ack(r))_{(k-1,k),k}$, and by Theorem 1, $\text{receive}(q)_{(k,k+1),k+1}$ $\not\to_X \text{send}(q)_{(k+1,k+2),k+1}$.

   In both cases, we have $\text{receive}(r)_{(n,b),n} <_X \text{receive}(q)_{(k-1,k)} <_X \text{receive}(ack(r))_{(k-1,k),k}$, and by Theorem 3.1, $\text{receive}(q)_{(k-1,k),k} \not\to_X \text{send}(q)_{(n,b),n}$. By the definition of $\to_X$ therefore, $\text{send}(q)_{(a,1),a} \not\to_X \text{send}(r_q)_{(n,b),b}$ $\square$

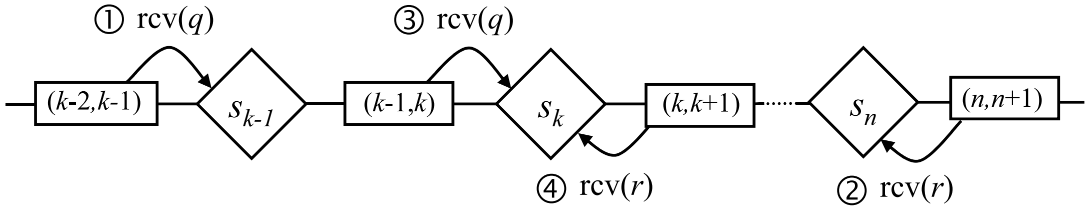


Figure 3.2: the case of $k > 1$

Note that we only invoke Responder Well-Formedness for the "loophole" case where a request is received on the last switch before that switch receives a conflicting response; if *R-RCV-FIRST* is true then the network path takes care of detecting the conflict.

By reversing the implication of Theorem 3.2 we get a corollary for the case when a request is *not* dropped but successfully causes a response, namely:

Corollary 3.3 *For all* $X \in execs(AS)$ *where* X *is well-formed for hosts* a, $b \in HOSTS$, $r \in R$ *enqueued by an event* e_r *at* b *and* $q \in Q$*:* $receive(q)_{(n,b),b} \rightarrow_X send(r_q)_{(n,b),b} \Rightarrow receive(q)_{(n,b),b} <_X e_r \vee send(ack(r))_{(a,1),a} <_X send(q)_{(a,1),a} \vee \neg conflicts(q, r)$.

This justifies calling the transfer "atomic": a successful request q is received at b before any conflict-causing response r is enqueued or else requester a had already sent an ACK for r before issuing q and so presumably took r into account when issuing q. Informally, if we think of responses as carrying information and let each request conflict with information that could have prevented the request from being issued, then every successful request is based on up-to-date information. By analogy with serializability of transactions [8], every execution of the network corresponds to a serial execution where all non-conflicting requests are received in the same order but only a single message is in-flight in the network at any one time.

Notice that once b has received the ACK for a response r, it can effectively behave as if it had received the ACK from *all* the subscribers of r, no matter how many or far removed they are. Since b does not need to track the identities of these subscribers, Atomic Transfer should scale in a similar way as reliable multicast protocols.

Also note that AT by itself doesn't enforce any particular correctness standard, such as serializability. The consistency guarantees of AT depend entirely on the semantics of requests and responses and the *conflicts* relation. The related work section of Chapter 5 presents an example of a simple read/write system that preserves serializability.

3.5 Liveness

We've shown the safety of Atomic Transfer, that is: a request cannot traverse a path containing a conflicting response. We now show that this safety property is not trivial, that is: a request that does not encounter a conflicting response is eventually delivered to its receiver. This is straightforward, since the network behaves very much like a normal store-and-forward network in this case.

We require for liveness that requesters never attempt to send a query that cannot be forwarded (which is trivial if each $qHop_i$ is total).

Definition 3.8 *An execution or trace E is Subscription Well-Formed for a requester $a \in HOSTS$ if for every event $receive(q \in Q)_{(a,i),i}$ in E at the home switch i of a, $qHop_i(q)$ is defined.*

Our proof shows that the distance between a message and its destination, measured as the number of messages ahead of it in switch and channel queues on its forwarding path, continues to decrease in any fair execution. Eventually, it falls to zero and the message is delivered. We define message distances with respect to channels and switches as follows:

Definition 3.9 *For any message $m \in c.outQueue_{i,j}$ where $c \in CHANNELS$, let $distance_c(m)$ be the number of messages in front of m on $c.outQueue_{i,j}$.*

Definition 3.10 *For any message $m \in i.outQueue_c$ where $i \in SWITCHES$ and $c \in CHANNELS_i$, let $distance_{ci}(m)$ be the number of messages in front of m on $i.outQueue_c$.*

For every channel $c = (i, j) \in CHANNELS$ we define the following tasks, recalling the I/O Automata definition of tasks from Section 2.5:

- $rcv_{c,i}$, containing all receive$(m \in M)_{c,i}$ actions.
- $rcv_{c,j}$, containing all receive$(m \in M)_{c,j}$ actions.
- $snd_{c,i}$, containing all send$(m \in M)_{c,i}$ actions.
- $snd_{c,j}$, containing all send$(m \in M)_{c,j}$ actions.

This ensures that in any fair execution, a channel or switch enabled to send or receive will always get a chance to make progress. We show that a message enqueued on a channel is eventually delivered. A switch implementation would need to implement a fair scheduling policy to preserve liveness, for example FIFO or else some fair queueing discipline such as (weighted) round-robin.

Lemma 3.5 *(Channel Liveness) For all $X \in$ fairexecs(AS), for any message $m \in M$ such that $m \in c.outQueue_{i,j}$ in a state $t \in X$, where $c \in$ CHANNELS, there is a receive$(m)_{c,j}$ later in X.*

Proof: Let $d = distance_c(m)$ at t and let X' denote the suffix of X beginning with t. Each $receive(m' \in M)_{c,j}$ event in X' decreases $d = distance_c(m)$ by 1, since it removes the head of the queue. Since $receive(m' \in M)_{c,j}$ is always enabled when the queue is non-empty, it cannot become disabled before m is removed from the queue. Since X is fair and $receive(m' \in M)_{c,j}$ is in its own task, either there are infinitely many occurrences of $receive(m' \in M)_{c,j}$ in X' or there are infinitely many states where $receive(m' \in M)_{c,j}$ is disabled. In the first case, the events must eventually drive d to zero, making m the head of the queue at the next receive event, $receive(m)_{c,j}$. In the latter case, the queue will eventually become empty, so X' must contain a $receive(m)_{c,j}$ event that removes m. In each case, the Lemma holds $\Box$.

Similarly, we show that a message enqueued on a switch is eventually sent. Note that we used "cause" in the formal sense defined in Lemmas 3.1 and 3.2.

Lemma 3.6 *(Switch Liveness) For all $X \in$ fairexecs(AS), for any message $m \in M$ such that $m \in i.outQueue_c$ in a state $t \in X$, where $i \in$ SWITCHES and $c \in$ CHANNELS$_i$, there is a send$(m)_{c,i}$ later in X.*

Proof: The proof proceeds almost exactly like the proof of Lemma 3.5, using $distance_{ci}(m)$ as the progress metric and send$(m' \in M)_{c,i}$ events $\square$.

Theorem 3.3 *(Atomic Transfer Liveness) For all $X \in fairexecs(AS)$, where X is Subscription Well-Formed for a host $a \in HOSTS$ with home switch i, for each event $e = send(q \in Q)_{(a,i),i}$ such that no message caused by e is detected as a conflict in X, there is a $receive(q \in Q)_{(b,j),b}$ event in X, where b =destination(q), and j is the home switch of b.*

Proof: Let let X' denote the suffix of X whose first event is e. Let $fp(q) = s_1, \ldots, s_n$, the forwarding path of q. By Lemma 3.5, every message enqueued on a channel is eventually received. Since a is Subscription Well-Formed, $qHop_{s_1}(q)$ is defined. Since q is never detected as a conflict, every receive$(q)_{(k-1,k),k}$ event caused by e adds q to $s_k.outQueue_{k+1}$, for $1 \leq k < n$. By Lemma 3.6, every message enqueued on a switch is eventually sent. Inductively, therefore, there must be a receive$(q \in Q)_{(b,j),b}$ event in X' $\square$

One might wonder how a requester a knows whether a request q it issues is dropped due to a conflict or not. Requester a can in fact infer the dropping of q if it recognizes its own responses, that is: if it knows for any response r whether r is a response to q (a reasonable implementation assumption). If q is dropped, it is due to a conflict with some other response r', and by Conflict Locality a is subscribed to r'. Requester a can therefore store the latest request sent to each responder and conflict-check incoming responses against that request. If a receives a response r' that conflicts with q but is not a response to q, then a knows that q was dropped.

3.6 Optimizations

This section presents a few enhancements for atomic switches, for better performance or increased flexibility.

3.6.1 Permissible Message Reordering

Although the proofs for Theorems 3.1 and 3.2 assume strict FIFO communication between nodes, we can relax these a bit, demonstrating that Quality of Service (QoS) processing, such as assigning message priorities, is compatible with Atomic Transfer. We describe the permissible reorderings of adjacent pairs of messages in channel or switch queues. This captures how implementations would reorder messages by placing them on one of a channel's several outbound queues that have different priorities. The main restriction is that the relative order of requests and ACKs from a particular sender may not be altered.

A request q can swap place with an adjacent request or ACK m if q and m have different senders. In this dissertation, we impose the restriction that a requester only issue two requests concurrently if both can execute regardless of whether the other one is dropped or not. Then, two requests q and q' from the same sender can be swapped, but a request q cannot be swapped with an ACK a from the same sender. If q moved ahead of a, it might be erroneously detected as a conflict. Conversely, if q moved behind a, it could have a conflict that is not detected.

A response r can swap place with an adjacent response or ACK m if r and m have different senders. Two responses r and r' from the same sender can be swapped if they commute, that is: if it doesn't matter to receivers in which order they are received. Even if they don't commute, their order can be swapped if the receiver can detect that a response is missing when it receives an out-of-order response, and put them back in the original order. A response r can swap places with a request q. Since q and r are heading the same direction, q cannot be bound for the responder of r and the messages are unrelated.

3.6.2 Discarding doomed requests

To reduce the number of messages processed, atomic switch implementations can remove "doomed" outgoing requests as soon as a conflicting response is received, without affecting the proofs of Theorems 3.1 and 3.2. This can be modeled as a simple addition to the receive$(r \in R)_{c,i}$ action for AtomicSwitch$_i$, as follows:

```
receive$(r \in R)_{c,i}$
Effect:
    $outQueue'_c = outQueue_c \cdot ack(r)$
    for each $d \in CHANNELS_i$ such that $(r, d) \in rHops_i$
        $outQueue'_d = outQueue_d \cdot r$
        $responses'_d = responses_d \cup \{r\}$
    // remove requests that would be detected as conflicts on receiving switch
    let $CQ = \{\ q \in outQueue_c : conflicts(q, r)\ \}$ in
        $outQueue'_c = outQueue_c \setminus CQ$
```

Whether this optimizations helps in practice depends on whether the cost of the additional detection of conflicting requests outweighs the gain from not having to transmit them.

3.6.3 Mixed Networks and Dynamic Shirking

In our models, atomic switches are connected via reliable FIFO channels. These can represent any reliable link, including end-to-end constructs such as TCP/IP connections. More generally, a channel may be composed of multiple links and switches, at a lower level. Hence, Atomic Networks can be composed of a mixture of atomic switches and normal, non-atomic switches, as long as assumptions about path uniqueness and reliability are met. High-throughput switches in the network's core, for example, may not be able (or willing) to perform conflict-checking, while switches closer to requesters and responders may be configured to conflict-check their requests and responses. In the limit, atomic requesters and responders can interact over a network containing no atomic switches at all! In that case, all conflict detection takes place on the responder and Atomic Transfer essentially reduces to an existing optimistic concurrency control algorithm (see Section 6.9.2 in Chapter 6).

An atomic switch can in fact decide whether to behave as an atomic switch or normal switch dynamically, on a response-by-response basis. A switch i can *dynamically shirk* responsibility for checking a response r it receives, by forwarding r but not sending an ACK for it nor entering it into a responses set. This way, the sender j of r remains responsible for conflict-checking requests with r, with i essentially serving as a simple FIFO channel with respect to r. Upon receiving the ACK for r, i forwards it back to j, allowing j to remove r from its $j.\mathit{responses}_{(i,j),j}$ set. By shirking responsibility for conflict-checking r, i shifts conflict-checking effort from itself to j. Shirking is easily implemented when a response is bound for a single channel. If, on the other hand, it is bound for multiple channels then additional steps must be taken to ensure that an ACK is only forwarded back once all outbound channels have ACKed the response. Hence, shirking adds state and/or complexity in the general case. We also observe that shirking increases the retention time of responses, so high fan-out nodes with long response-times to non-shirking switches might face a higher conflict-checking burden.

Yet, dynamic shirking may be an important technique for obtaining a net performance gain from Atomic Transfer. Since conflicts are generally rare, switches may shirk responsibility as a general rule, incurring no forwarding overhead. In this case, responders would hold onto their responses for a relatively long time and conflict checking would be performed more or less end-to-end. However, switches would preferentially hold onto responses affecting "hot" data, experiencing many conflicts. Responders could help, by measuring data temperatures and flagging responses they generate that affect hot areas. This way, switches could focus their conflict-checking efforts where they are most needed.

We keep our models simple in this dissertation by considering only "pure" atomic networks, comprised only of atomic switches that do not shirk. A fuller investigation of shirking awaits future work.

3.7 Discussion and Related Work

This section discuss Atomic Transfer design and performance issues in relation to existing work.

3.7.1 Performance Analysis

The main overhead imposed by AT is the computation needed to check a request against responses held on a switch. Chapter 8 presents an implementation with modest time and space overhead. We note that AT does not increase messages residency times on switches beyond what would be needed for hop-by-hop reliable transfer, for example. Hence, the memory overhead of AT can be made modest.

The main motivation for Atomic Transfer is to enable atomic cache systems based on Optimistic Concurrency Control (OCC) to better handle heavy data contention [16]. As data contention increases at a responder host using traditional OCC, the ratio of requests that fail validation rises and the useful throughput of the responder host falls. The loss of goodput leads to a rise in request response times, increasing the probability of conflicting responses being issued. This can create a negative performance feedback loop. We note that even if conflict checking is fast, which it must be in order to run on switches, a host's reception of a message may by itself require significant processing, including the copying of data between buffers and crossings of operating system protection domains, etc.

The basic idea behind Atomic Transfer is to drop most conflicting requests before they reach a responder, protecting responders from overload due to a deluge of conflicting requests during periods of high contention. One might ask whether this cannot be achieved by simply slowing the influx of requests using per-connection flow-control, as responder multiprogramming and load levels rise above desired levels. This is a poor option, though, because barring information about the accesses of requests awaiting delivery[7] it would mean a uniform slowdown of delivery of *all* requests, not only those accessing contended data. Even if only a small fraction of a server's data were contended, its overall performance for all requests would be adversely affected. By contrast, Atomic Transfer filters out the conflicting requests, delivering an unimpeded flow of (mostly) conflict-free requests even if some data are heavily contended. This is particularly important for multi-processing and/or virtualized [124] servers, that process multiple unrelated requests in parallel.

[7]It could be known if certain connections have relatively static affinity for certain data, e.g.

Theorem 3.1 shows that the only way for a request q to reach a responder host b and be detected as a conflict with a (recently created) response r at b is for q to be received at the home switch i of b before r is received at i. If r is created in response to a request q', then q must be received at i within $rtt_{ib} + p_{q'}$ time units of q' being sent from i, where rtt_{ib} is the round-trip-time between i and b and $p_{q'}$ is the amount of time it takes b to process q' and generate response r. If q is received later than this, then r has been received at i and q will be detected as a conflict on i. Let δ_b denote $p_b + rtt_{ib}$, where p_b is the average time it takes b to process a request.

Let q and q' be two conflicting requests issued to a responder b, or more precisely: where the response r' to q' causes q to be dropped at b. Let a and a' be the requesters of q and q', respectively, and let t and t' be the round-trip-times to b from a and a', respectively. Since q is detected as a conflict and dropped at b, it must be sent by a no later than $\frac{1}{2}t + \delta_b + \frac{1}{2}t'$ time units after a' sends q', since otherwise r' will reach a before it issues q and prevent q from being issued. If we assume that a is equally likely to issue q at any point in this period after q' is sent, then the probability that q will be received at i within δ_b time units of q' and be detected as a conflict at b is:

$$\frac{\delta_b}{\frac{1}{2}t + \delta_b + \frac{1}{2}t'} \tag{3.1}$$

If we assume $t \approx t'$ then the equation simplifies to $\delta_b/(\delta_b + t)$. Hence, the probability of a conflict being detected at the server is the ratio of responder response time δ_b to round-trip request / response time. Equation 3.2 confirms the intuition that the efficacy of Atomic Transfer in shielding responders from wasting effort on validating conflicting requests depends substantially on the relationship between network latency and request processing times.

In general we expect rtt_{ib} to be low in relation to t, e.g. on the order of tens or hundreds of microseconds. In current systems, the factor p_b for processing would usually dominate. For example, if a request involves one or two hard disk accesses, then p_b could be on the order of 10ms. If requesters were widely distributed, with average round-trip-time of around 100ms, b would have to detect about about one out of ten conflicts. If requesters were in the same LAN, however, with average round-trip-time of around 1ms, then b would have to detect the conflict in nine out of ten cases.

There are reasons to believe that processing times for transactional requests will decrease significantly in the near future. Historically, the time needed to force-write state updates or log entries [125] to a hard disk has contributed at least several milliseconds of latency to atomic operation response times. Scattered data reads from hard disks are similarly expensive. Emerging storage technologies such as solid-state disks (SSDs) reduce this latency by one to two orders of magnitude. Furthermore, non-volatile memory capacities have grown to the point that many databases fit entirely in main memory [126], largely obviating the need for disk reads. Techniques have also been suggested for building persistent stores from volatile memory, using battery back-up, software protection [127] and/or redundancy [128]. By contrast, while network bandwidth has been growing apace, network latency cannot decrease to the same measure [129, 130].

More importantly, though, equation 3.2 concerns a pair of conflicting requests. For server concurrency control performance, what matters is the proportion of all conflicting requests that are detected at the server instead of the network. Since a single response can cause an arbitrary number of conflicting requests to be dropped in the network, this proportion actually falls as contention increases. More precisely, let $p_q = \delta_b/(\delta_b + t)$ be the probability that a conflicting request q is dropped at b, as defined for the derivation of equation 3.2 assuming uniform network round-trip time t. If k conflicting requests are issued no earlier than $\delta_b + t$ time units before q and arrive at the home switch i of b ahead of q, the probability of q being dropped at b is the probability of q and the other k requests arriving at i within δ_b time units of each of other. Hence, the probability of q being dropped at b as a function of k is p_q^k. If we assume that n requests that conflict with q are issued on average during any time period $\delta_b + t$, the probability that j of these requests are received ahead of q at i is $1/n$, for $1 \leq j \leq n$. Hence, the probability that request q is dropped at b as a function of n is:

$$\sum_{k=1}^{n} \frac{1}{n} \cdot p_q^k = \frac{1}{n} \cdot \frac{p_q^{n+1} - p_q}{p_q - 1} \tag{3.2}$$

This is also the proportion of conflicting requests dropped at b, tending to 0 as p_q tends to 0 and 1 as p_q tends to 1. Hence, even for a relatively high pair-wise probability of $p_q = 0.9$, the proportion of conflicts detected at b is only around 0.73, 0.58 and 0.08, for $n = 5, 10$ and 100, respectively. A more favorable ratio of $p_q = 0.25$ yields ratios 0.07, 0.003 and close to zero, respectively. The expected number of conflicting requests detected at b per unit of time is proportional to $(p_q^{n+1} - p_q)/(p_q - 1)$, which tends to $p_q/(1-p_q)$ as n tends to infinity, for $|p_q| < 1$. Given these somewhat idealized assumptions and simplified case, the number of conflicting requests reaching b and wasting its resources therefore stays relatively constant, regardless of data contention levels.

These calculations also assume that requesters are equally likely to issue their conflicting requests at any point in the interval before q is issued. That assumption can easily be violated in practice if requests are correlated, for example if requesters are equidistant from b and react to the same responses with the same reaction times. Requesters could attempt to detect such situations and resolve them by adding a random delay on the order of δ_b to their reactions, similar to the randomized backoff delays used in the physical layer in some shared-medium network protocols[8]. As an alternative, Section 9.3.1 of Chapter 9 on future work sketches an approach that may help in such cases, by completely obviating b from conflict-checking.

We observe that in our model, the maximum rate of mutually conflicting requests that a server can execute is $1/(\delta_b + t)$, the inverse of the time for request execution and a network round-trip. This is because the next conflicting request that is successfully executed can only be issued after its requester receives the response for the prior request, or else it will be detected as a conflict. Note that this limit is independent of the number of concurrent requesters. Also note that his is not an artifact of our approach, but a more general limit imposed by a requester's need to receive recent enough state information to issue a request that can be serialized. Still, Section 9.3.2 of chapter 9 on future work suggests ways to permit servers to surpass the throughput limit imposed by round-trip delays in the non-contended case, by allowing requesters to pipeline requests to a server.

[8]Note that there is no possibility for livelock, since a request is only dropped due to a successfully executed request

3.7.2 Related Work: Isotach Networks

While supercomputers and other specialized multiprocessor systems are often designed around specialized interconnect networks, the only relatively general work we're aware of that specifically seeks to use networks to accelerate concurrency control are Isotach Networks [131]. The idea is to ensure that the network delivers messages in an order consistent with logical time [123]. Furthermore, the network preserves the *isotach invariant*, which says that a message takes exactly one unit of logical time to travel from one switch to another. This allows a node that has knowledge of a network's topology to ensure that a set of messages are delivered at multiple destinations at the same logical time, by setting the logical time of each send event e_s to $e_r - d$, where e_r is the desired receive logical time and d is the distance (in logical time units / network hops) to the receiver of the message. A set of messages can be bundled together as an *isochron*, so they are delivered (in a sender-specified order) at the same instant of logical time. Switches and network interface adapters delay messages as to ensure that messages are received and delivered in logical time order. Isotach Networks can readily preserve sequential consistency [11] and implement totally ordered multicast, but they cannot directly ensure serializability, as write operations have no dependencies associated with them but simply overwrite values. New operations are introduced for this purpose: a *SCHED* operation that effectively locks variables and an *ASSIGN* operation that updates them.

This approach is quite different from ours. Incrementing and comparing logical timestamps is faster than comparing requests and responses. However, while simulations show impressive gains over locking [131], assigning a single logical clock to each host creates an artificial ordering on otherwise independent activities on the host. By contrast, in our approach, unrelated (non-conflicting) requests and responses do not impede one another. We believe the ability to run multiple, unrelated applications on the same host is important for flexibility and high host utilization. Also, delaying messages in the network is problematic. At higher (gigabit) data rates, storing messages for even a short time requires significant amounts of memory. Indeed, the prototype Isotach implementation [132], on top of a reliable Myrinet interconnect, does not delay messages but buffers them at end-points and essentially performs Isotach processing end-to-end.

End-to-end arguments [133] loom large over any suggestions to add functionality to the network layer. Ultimately, Atomic Transfer is only justified if it improves performance. Operation conflict-detection can certainly be done end-to-end. However, the algorithm of Chapter 7 for hierarchical consistent caching takes direct advantage of network topology and in-network processing in a way that does not seem to have an efficient end-to-end counterpart. As a general thought, in the uphill struggle against latency [130], it seems significant that the earliest time conflicting requests and responses can physically meet in a networked system is within the network.

Chapter 4

Transactional Cache using Atomic Transfer

As our main and motivating example of the application of Atomic Transfer, we show how to build a system of application hosts that send atomic operation requests to data server hosts. The system ensures that concurrent operations from one host do not cause errors by interference with those of other hosts. An application host executes each request optimistically using locally cached server state and subsequently sends the request to its server, which executes it and/or incorporates it into the server's state, barring conflicts.

4.1 Application Hosts, Data Server Hosts and Executions

Let *HOSTS* now be partitioned into the sets *APPS* and *DATAS* of *application hosts* and *data server hosts*, respectively. As a starting point in this simple system, an application host has no state of its own but caches a complete copy of a recent state of each data server it calls. Initially, we assume that each application host is permanently subscribed to the responses of each data server host it calls. We remove this assumption in Chapter 5, where we describe a protocol allowing hosts to dynamically vary their subscriptions.

As a concrete example of a system environment, application hosts and data servers might be hosted in the same data center, with multiple ongoing applications contending for their data. Additional application hosts might connect from outside the center, executing interactive applications used by remote and / or mobile end-users. The low-latency atomic execution afforded by AT caching would be highly suitable for interactive multi-user applications such as shared document editing, conferencing and multi-user virtual worlds, for example.

Associate with each server $b \in DATAS$ a (possibly infinite) *state space* $STATES_b$ and a unique *start state* $start_b \in STATES_b$. As before, each request is handled entirely by one data server. Requests spanning multiple hosts require an atomic commit protocol such as two-phase commit [2] to ensure that a transaction is committed at one host if and only if it is committed at all the other hosts. Chapter 9 on future work outlines how to incorporate such a protocol into our model.

For each server $b \in DATAS$ let $execute_b$ be a relation $STATES_b \times Q \times R$, relating states of b and requests to possible responses given that state. We use $execute_b(s,\ q)$ to denote the set of possible responses to q in state s, so $r \in execute_b(s,\ q) \Leftrightarrow (s,\ q,\ r) \in execute_b$. For convenience, we define $execute_b$ to be total for any pair of state and request, returning designated error responses for requests that are malformed for the server or otherwise invalid in the current state.

When a server $b \in DATAS$ *executes* a request q it moves to a new state according to a deterministic state transition function $TRANS_b$: $STATES_b \times R \rightarrow STATES_b$, known to all application hosts that call b. We define $TRANS_b$ as total, returning a designated error state $error_b \in STATES_b$ for responses that are malformed or otherwise invalid for a state. We also stipulate that $STATES_b$ is the identity mapping for error responses.

Note that the state transition takes a response as an argument, not a request; a server's mapping from requests to responses is defined by $execute_b$. The $TRANS_b$ function roughly correspond to the part of an implementation and protocol that updates state caches in applications and keeps them synchronized, while the $execute_b$ relation corresponds to (possibly non-deterministic) server-side software. A server may also update its state and create a response spontaneously, without being prompted by an external request.

4.2 Cache Consistency through Atomic Transfer

We use Atomic Transfer to prevent requests based on stale cache information from being executed. More specifically, the AT cache system drops requests whose set of possible responses may be different from what the application expected, due to concurrent interference. To this end, we define the conflict relation *conflicts-inv* as containing all pairs of requests and responses (q, r) such that r can *invalidate* q, by altering the set of possible responses to q. Precisely:

Definition 4.1 *conflicts-inv* $= \{ (q \in Q,\ r \in R) \mid \exists s \in STATES_b :$ $\mathit{execute}_b(s, q) \not\subseteq \mathit{execute}_b(TRANS_b(s, r), q)\}$, *where* $b = \mathit{destination}(q)$.

In the case where *execute*$_b$ is deterministic for q, the condition simplifies to: *execute*$_b$(s, q) $\neq$ *execute*$_b$($TRANS_b(s, r), q$). The *conflicts-inv* relation may be conservative: the presence of a pair implies that a particular response can invalidate a particular request, depending on the server state. On the other hand, the absence of a pair means that the response can never invalidate the request, in any server state.

As a simple but concrete example of an application adhering to this model, let requests be sets of variable reads and writes and let responses be notifications about variable value changes. In this case, the state transition function is simply the function yielding the old state with the component corresponding to the updated variables replaced with new values. The *conflicts-inv* relation could include all requests and responses where the response has a variable name in common with a read in the request. However, our model does permit a wider range of requests and response types, corresponding to more complex conflict relations. We further discuss this and related work in Section 4.5.

The system we present may seem somewhat constrained, with stateless applications that maintain caches of data and do not even keep track of the requests they have issued! But keep in mind that application hosts can cache large amounts of data for arbitrary lengths of time, and that transactions access this data with performance similar to a purely local application. When contention is light the overhead can be made very small, but when data sharing and contention arises the concurrency control ensures that executions interleave in a correct, serializable manner. Also, at the implementation level, an application will likely keep track of its outstanding requests and, possibly, their (predicted) effects on its cached state. However, effects do not take permanent effect until a response is received.

We note that our model as presented does not guarantee starvation-freedom: an application can fail to make progress in the case where its requests repeatedly conflict with responses from other applications. The network's topology can influence the likelihood of this occurring. For example, if several applications react to the same state change, the application host closest to the data server in the network is most likely to get its request executed. In cases where this is undesirable, techniques where losing request queue up for a chance at re-execution can be used to enforce fairness [134]. Conflict checking might play a role in establishing the queuing order. Section 9.3.3 in Chapter 9 on future work sketches how this might work.

Applications are permitted to have more than one request outstanding at a time, but only if the requests are independent, that is: if a response to neither request can conflict with the other. This assumption can be lifted to allow applications to "pipeline" dependent requests, as we discuss in Chapter 9 on future work.

We contend that atomic caching is a flexible building block for scalable systems and a sound basis for dynamic balancing of workloads across a collection of hosts. Hosts in today's data centers are woefully underutilized, often running at 3-10% of capacity [135], which is wasteful since they draw significant power even as they idle or perform at low levels. Atomic, "stateless" applications can be migrated between hosts with relative ease to maintain efficient load levels and additional copies of applications can be swiftly started and stopped to meet fluctuating demands. Conversely, when loads are low, applications can migrate closer to the servers hosting their data and in the limit, onto the same hosts. Such techniques are widespread for simple applications such as serving of Web content, but challenging for more demanding applications that both read and update shared state. While various ad-hoc techniques have been used to scale such systems, we believe that the vision of universally accessible and shareable applications needs a stronger and more rational foundation. We believe atomic execution models and high-performance, transparent caching to be such a foundation.

4.3 SimpleAppHost$_a$ and SimpleDataServerHost$_b$

We provide I/O automata for application hosts and data server hosts below. We then show that composing these with atomic switches and channels results in a system where the cached state on application hosts stays synchronized with the corresponding server(s) and invalid requests are never executed.

SimpleAppHost$_a$

Models an application host, that issues requests and consumes responses by updating its cached state for the server sending the response. Non-deterministically chooses (valid) requests to issue to servers.

State:

```
for each $b \in DATAS$ that SimpleAppHost$_a$ calls
    $state_b$: a state from $STATES_b$, initially $start_b$.
$outQueue$: queue of outbound messages, initially empty.
```

Input actions:

```
receive$(r \in R)_{c,a}$
Effect:
    // update cached state
    $state'_b = TRANS_b(state_b, r)$, where $b = sender(r)$
    // enqueue an ACK back
    $outQueue' = outQueue \cdot ack(r)$
```

Internal actions:

```
createRequest$(q \in Q)_a$
Precondition:
    // $q$ must be valid, given the current cached state
```

$\exists r \in R : (state_b, q, r) \in execute_b$, where $b = destination(q)$

Effect:

// enqueue a request

$outQueue' = outQueue \cdot q$

Output actions:

send$(m \in M)_{c,a}$

$head(outQueue) = m$

Effect:

$outQueue' = tail(outQueue)$

The receive$(r \in R)_{c,a}$ action receives a response and updates the corresponding data server's cached state, using the server's state transition function. Note that the application host behaves the same way whether the response is due to its own request or not. The action also enqueues back an ACK, that will allow the home switch to remove the response from its *responses* set.

The createRequest$(q \in Q)_a$ action enqueues an arbitrary request, with the sole restriction that it must be possible to execute the request in the cached version of the destination data server's state.

The send$(m \in M)_{c,a}$ action moves a pending message to the outbound channel.

SimpleDataServerHost$_b$

Models a data server host, that executes requests and creates responses.

State:

state: the state of the data server $\in STATES_b$, initially $start_b$.
outQueue: queue of outbound responses, initially empty.

responses: set of non-acknowledged responses, initially empty.

Input actions:

```
receive($q \in Q$)$_{c,b}$
Effect:
    // if the request conflicts with an un-acknowledged response we've created
    if $\exists r \in \mathit{responses}$ such that $\mathit{conflicts\text{-}inv}(q, r)$
        // do not enqueue it, but handle it somehow
        $\mathit{handleConflict}(q)$
    else let $r \in R$ be a response such that $(q, \mathit{state}, r) \in \mathit{execute}_b$ in
        // update state, as per response r
        $\mathit{state}' = \mathit{TRANS}_b(\mathit{state}, r)$
        // enqueue response
        $\mathit{outQueue}' = \mathit{outQueue} \cdot r$
        // remember response, for conflict checking
        $\mathit{responses}' = \mathit{responses} \cup \{r\}$
```

```
receive($a \in A$)$_{c,b}$
Effect:
    $\mathit{responses}' = \mathit{responses} \setminus \{\mathit{message}(a)\}$
```

Output actions:

```
send($r \in R$)$_{c,b}$
Precondition:
    $\mathit{head}(\mathit{outQueue}) = r$
Effect:
    $\mathit{outQueue}' = \mathit{tail}(\mathit{outQueue})$
```

The receive($q \in Q)_{c,b}$ action receives and executes a request, but only if it doesn't conflict with an unacknowledged response. If the request is executed, the corresponding response is enqueued for sending. It is also added to the set of unacknowledged responses.

The receive($a \in A)_{c,b}$ action removes the acknowledged response from the set of unacknowledged responses, since responsibility for conflict-checking the response has shifted to the home switch and, ultimately, the tree of network paths leading to subscribing application hosts.

The send($m \in M)_{c,b}$ action hands a pending message to the outbound channel.

4.4 Properties and Proofs

Let CS be the I/O automaton composed of an AtomicSwitch$_i$ automaton for each switch $s_i \in$ *NODES,* a SimpleAppHost$_a$ automaton for each application host $a \in$ *APPS,* SimpleDataServerHost$_b$ automaton for each data server host $b \in$ DATAS and a Channel$_c$ automaton for each channel $c = (i, j) \in$ *CHANNELS.*

We first prove some straightforward lemmas, showing that requesters receive a server's responses in the order that the server sends them. We subsequently use those results to show that application caches accurately reflect recent server states. Note that these lemmas concern the FIFO properties of channels and switches, not Atomic Transfer. The proof of atomicity follows from Theorem 3.2 in a straightforward manner. We begin with some notation. We define a function to allow us to refer to the message associated with a send or receive event.

Definition 4.2 *For all events e = send($m \in M)_{c,i}$ or receive($m \in M)_{c,i}$ events and any $c \in$ CHANNELS and $i \in$ NODES, let* messageOf(e) $= m$. *We extend* messageOf *to function* messagesOf *over sequences of these two types of events in the obvious way, mapping a sequence of events to the sequence of the messages of each event parameter.*

We also introduce a function to easily refer to send and receive events to and from particular switches and hosts.

Definition 4.3 *For any execution or trace E, i, $j \in$ NODES and message set $G \subseteq M$, let sndSeq$_{c,j}(E,i,G)$ (rcvSeq$_{c,j}(E,i,G)$) denote the maximal subsequence of* trace(E) *or E, respectively, consisting of send($m \in G)_{c,j}$ events (receive($m \in G)_{c,j}$ events) that are caused by send($m \in G)_{d,i}$ events, for some $c \in CHANNELS_j$, $d \in CHANNELS_i$.*

Note that if $i \in$*HOSTS* then for each receive$(m)_{c,j}$ in *rcvSeq*$_{c,j}(E,i,G)$ we have $i =$ *sender*(m). Finally, we define a prefix comparison operator as follows:

Definition 4.4 *Let $\preceq$ denote the relation containing sequences s' and s exactly if they are equal or if s' is a prefix of s, so we write $s' \preceq s$.*

We begin with a Lemma showing that a network path maintains FIFO order of responses from a particular server. More precisely, we show that the sequence of responses sent from a server b to an application a by the home switch of a is a prefix of the sequence of responses received from b by the home switch of b. Note that the Lemma appears "reversed" with respect to $\preceq$, since it talks about the responses sent by an applications home switch and received by a server's home switch, not the responses received by an application and sent by a server.

Lemma 4.1 *(switch response order): For each $E \in$ traces(CS), any application host $a \in APPS$ with home switch i and any data server host $b \in DATAS$ with home switch j,* messagesOf(sndSeq$_{(a,i),i}(E,b,R)$) $\preceq$ messagesOf(rcvSeq$_{(j,b),j}(E,b,R)$).

Proof: If a is not subscribed to b then *sndSeq*$_{(a,i),i}(E,b,R)$ is empty and the result holds vacuously. Otherwise, we proceed by induction on path $fp_{ab} = i$, s_2, s_3, ..., j, the (unique) forwarding path in CS from host a to b.

The base case (Figure 4.1a) is the path comprised of a single switch s. Since the send$(r \in R)_{d,s}$ action consumes, in order, each response enqueued on *outQueue*$_d$ by the receive$(r \in R)_{c,s}$ action, *messagesOf*(*sndSeq*$_{d,s}(E,b,R)$) $\preceq$ *messagesOf*(*rcvSeq*$_{c,s}(E,b,R)$).

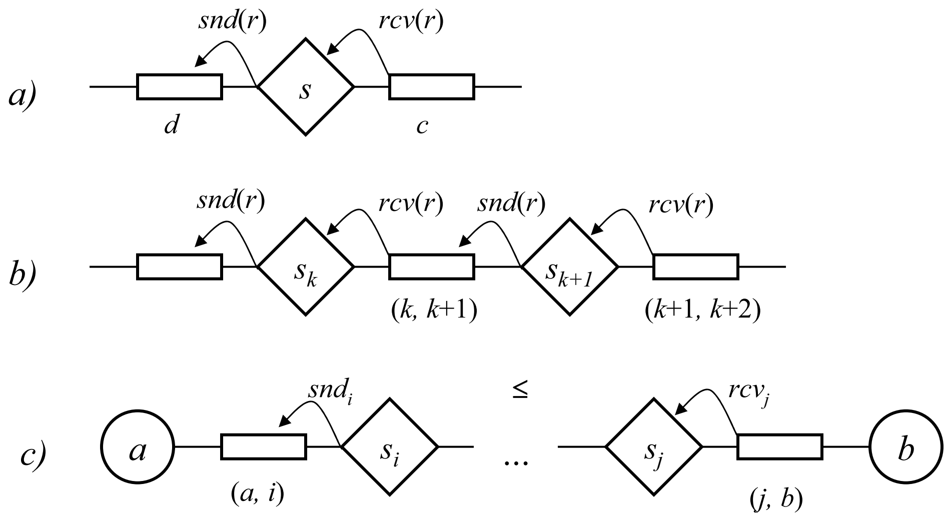


Figure 4.1: Base case, inductive case and Lemma 4.1

For the inductive step (Figure 4.1*b*), let *fp′*= *i*, ..., s_k, s_{k+1} denote the first k+1 switches of fp_{ab}, where k is less than the length of fp_{ab}. Let rcv_{k+1} = *message-sOf*($rcvSeq_{(k+1,k+2),k+1}(E, b, R)$) and let snd_{k+1} = *messagesOf*($sndSeq_{(k,k+1),k+1}(E, b, R)$). Since the send($r \in R)_{(k,k+1),k+1}$ action consumes, in order, each response enqueued on $outQueue_d$ by the receive($r \in R)_{(k+1,k+2),k+1}$ action, $snd_{k+1} \preceq rcv_{k+1}$. Now let rcv_k = *messagesOf*($rcvSeq_{(k,k+1),k}(E, b, R)$). By Channel FIFO, $rcv_k \preceq snd_{k+1}$. By the inductive hypothesis, the lemma holds for the path comprised of the first k switches of *fp′*, so $snd_i \preceq rcv_k$, where snd_i = *messagesOf*($sndSeq_{(a,i),i}$). We have $snd_i \preceq rcv_k \preceq snd_{k+1} \preceq rcv_{k+1}$, which completes the induction (Figure 4.1*c*) □

Lemma 4.1 concerns a path of atomic switches. We can readily extend it to the sequence of responses received by an application host and the sequence of responses sent by a server.

Definition 4.5 *For any execution or trace E, any i, j ∈ NODES, let snd-seq-*$r_i(E)$ = *messagesOf*($sndSeq_{c,i}(E, i, R)$) *and let rcv-seq-*$r_{ji}(E)$ = *messagesOf*($rcvSeq_{d,j}(E, i, R)$)*, where c and d are channels incident to i and j, respectively.*

Lemma 4.2 *(End-to-end response order): For any trace E ∈* traces(CS)*,* *rcv-seq-*$r_{ab}(E) \preceq$ *snd-seq-*$r_b(E)$.

Proof: If a is not subscribed to b then $\textit{rcv-seq-r}_{ab}(E)$ is empty and the result holds vacuously. Otherwise let i, s_2, s_3, ..., $j = \textit{fp}_{ab}$. Let $\textit{rcv}_c(E) = \textit{messagesOf}(\textit{rcvSeq}_{c,j}(E,b,R))$, where $c = (j,\ b)$. By Channel FIFO, $\textit{rcv}_c(E) \preceq \textit{snd-seq-r}_b(E)$. By Lemma 4.1, sequence $\textit{snd}_i(E) = \textit{messagesOf}(\textit{sndSeq}_{d,i}(E,b,R)) \preceq \textit{rcv}_c(E)$, where $d = (a,\ i)$. By Channel FIFO, sequence $\textit{rcv-seq-r}_{ab}(E) = \textit{messagesOf}(\textit{rcvSeq}_{d,a}(E, b, R)) \preceq \textit{snd}_i(E)$, so we have $\textit{rcv-seq-r}_{ab}(E) \preceq \textit{snd}_i(E) \preceq \textit{rcv}_c(E) \preceq \textit{snd-seq-r}_b(E)$ □

Since responses are delivered in FIFO order, we can show that the states cached by an application correspond exactly to those on the original server. We define the sequence of values (states) taken on by caches and servers, as well as the sequence of responses generated, as follows:

Definition 4.6 *For any execution $X \in$ execs(CS) and server $b \in DATAS$, let state-seq$_b(X)$ denote the sequence of values of b.state (the state field of server host b) from states following receive$(q)_{c,b}$ events in X. Similarly, let* cache-seq$_{ab}(X)$ *denote the sequence of values of a.state$_b$ (the state$_b$ field of application host a) following receive$(r)_{d,a}$ events in X. Let resp-seq-r$_b(X)$ denote the sequence of responses chosen by the receive$(q \in Q)_{c,b}$ action of SimpleDataServerHost$_b$ in X.*

Lemma 4.3 *(Cache synchronization): For each $X \in$* execs(CS) *and any application host $a \in APPS$ that subscribes to a server b:* cache-seq$_{ab}(X)$ $\preceq$ *state-seq$_b(X)$.*

Proof: For each response $r \in \textit{resp-seq-r}_b(X)$, in order from first to last, b assigns $\textit{TRANS}_b(\textit{b.state},\ r)$ to *b.state* while simultaneously enqueuing r on *b.outQueue.* By the FIFO property of *b.outQueue*, $\textit{snd-seq-r}_b(X) \preceq \textit{resp-seq-r}_b(X)$.

For each response r in $\textit{rcv-seq-r}_{ab}(X)$, in order from first to last, a assigns $\textit{TRANS}_b(\textit{a.state}_b,\ r)$ to $\textit{a.state}_b$. Since $\textit{TRANS}_b$ is a (deterministic) function, since $\textit{b.state} = \textit{a.state}_b$ in the start state of CS and since $\textit{rcv-seq-r}_{ab}(X) \preceq \textit{resp-seq-r}_b(X)$, the result follows □

Before invoking Theorem 3.1, we must establish that SimpleAppHost$_i$ and SimpleDataServerHost$_i$ preserve well-formedness.

Lemma 4.4 *SimpleAppHost$_a$ preserves ACK Well-Formedness.*

Proof: Immediate, since the only action in $\mathit{SimpleAppHost}_a$ that enqueues $\mathit{ack}(r)$ is receive$(r)_{d,i}$, where d is the channel incident to a □

Lemma 4.5 *$\mathit{SimpleDataServerHost}_b$ preserves Responder Well-Formedness.*

Proof: Let $r \in R$ be the response sent by b in response to a request $q_r \in Q$ and let $q \in Q$ be a request received by b, such that receive$(q_r)_{c,b} <_X$ receive$(q)_{c,b} <_X$ receive$(\mathit{ack}(r))_{c,b} \wedge \mathit{conflicts}(q, r)$, with c the channel incident to b.

We see from SimpleDataServerHost$_b$ that following the receive$(q_r \in Q)_{c,b}$ event we have $r \in b.\mathit{responses}$, up to the receive$(\mathit{ack}(r))_{c,b}$ event that removes r from $b.\mathit{responses}$. Hence, $r \in b.\mathit{responses}$ at receive$(q)_{c,b}$ and the receive$(q)_{c,b}$ action drops q □

The SimpleDataServerHost$_b$ has only one type of event that *enqueues* requests, as defined in Chapter 3. Since SimpleDataServerHost$_b$ executes a request q and enqueues its response r in a receive$(q)_{d,b}$ event, that event is the enqueuing event e_r for r. Observe that this is the only event that modifies the server's *state* field.

We are now ready to state the main result of this section, which says that if a server executes a request q then the response is among those possible given the application's cached state at the time the application issued q. Our proof shows that otherwise the intervening execution of the request that altered the server's state would have caused a response that conflicted with q and q would have been dropped.

Theorem 4.1 *(Cache Operation Atomicity): For any execution $X \in \mathit{execs}(CS)$, application $a \in \mathit{HOSTS}$, server $b \in \mathit{DATAS}$ and channels d and c incident to a and b, respectively: send$(q \in Q)_{d,a} \rightarrow_X$ send$(r_q \in R)_{c,b} \Rightarrow r_q \in \mathit{execute}_b(s_a,\ q)$, where s_a is the value of $a.\mathit{State}_b$ at the createRequest$(q)_{d,a}$ event enqueuing q.*

Proof: Assume for contradiction that there exists some execution $X \in \mathit{execs}(CS)$ and server $b \in$ DATAS where send$(q)_{d,a} \rightarrow_X$ send$(r_q \in R)_{c,b}$ but $r_q \notin \mathit{execute}_b(\mathrm{s}_a,\ q)$.

From Lemma 4.3 we have $\mathit{cache\text{-}seq}_{ab}(X) \preceq \mathit{state\text{-}seq}_b(X)$, so $b.\mathit{state}$ has value s_a at a point in X before the receive$(q)_{c,b}$ event that executes q. Therefore, there must be an event e_r at b preceding receive$(q)_{c,b}$ that assigns to $b.\mathit{state}$ a value s_b such that $r_q \notin$

$execute_b$(s_b, q). If we let r be the response enqueued by e_r, then by the definition of *conflicts-inv*, *conflicts-inv*$(q,\ r)$. Since $a.state_b = s_a$ at the createRequest$(q)_a$ event issuing request q, we have createRequest$(q)_a <_X$ receive$(r)_{d,a}$ and therefore by FIFO send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. This gives us $e_r <_X$ receive$(q)_{c,b} \wedge$ send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. Since by Lemma 4.4 a is ACK Well-Formed and by Lemma 4.5 b is Responder Well-Formed, by Theorem 3.2 we have send$(q)_{d,a} \not\rightarrow_X$ send$(r_q)_{c,b}$, a contradiction □

4.5 Discussion and Related Work

This section discuss performance issues and related work in concurrency control for Abstract Data Types.

4.5.1 Performance Analysis

Theorem 4.1 shows that CS implements a cache consistency algorithm, as requests based on stale cache information are dropped, either in the network or at the destination server. The host and server automata are fairly simple, depending on the guarantees of Atomic Transfer for their correctness. We claim that cache consistency based on Atomic Transfer is efficient in at least two respects.

First it admits all the concurrency afforded by the *conflict-resp* relation, since non-conflicting requests are never dropped. Schemes based on timestamps or logical time are more conservative and will generally reject some valid requests because they *might* be invalid given their timestamp or assigned logical time, reducing concurrency and performance.

Second, successful requests complete with minimum latency, in the sense that no message hops are added beyond those needed to get a message to the server and back again. An algorithm based on locking, by contrast, would require at least two round-trips: one to request and receive back the confirmation of a lock and another round-trip to issue the actual request and release the lock. Locking can also block and delay requests from other servers, reducing concurrency. Finally, locking can lead to deadlocks, while our method is deadlock-free.

We observe that since responses atomically update the cache, an application always has a consistent cache, corresponding to the current or, when stale, recent state of the server.

4.5.2 Serializability for Abstract Data Types

Our model is defined in terms of abstract requests and responses, rather than simple reads and writes of variables. Aside from generality and concord with Abstract Data Types (ADTs) and encapsulation, abstract operations can achieve higher levels of concurrency than simple reads and writes. For example, commutative operations such as addition can be defined as being non-conflicting and allowed to proceed in parallel. Adding a constant i to a state variable v using reads and writes, by contrast, requires a request that reads (depends on) v having a certain value x and then writes the new value $x + i$. All such operations on v must be totally ordered, so if v is contended it can become a *hot spot*, a concurrency control bottleneck in the system.

By contrast, the $TRANS_b$ relation permits arbitrary response semantics, as long as the computation of a new state is a function of the old state and the response. Hence, responses may be defined at a relatively high level, performing complex state-dependent processing such as deleting all values satisfying a certain predicate or transforming a 3D model according to some complex method, etc. This can lead to smaller response messages than explicit encoding of all values modified by a response.

But defining conflict relations that preserve an appropriate consistency standard for applications is a challenge. Classic serializability theory [8] deals with reads and writes, not abstract operations. However, Weihl and Herlihy have developed a theory of concurrency control for ADTs [19, 18] which is highly applicable to our model. Simplifying somewhat, let a request be comprised of a group of operation invocations on a group of typed objects (we will in fact model requests thus in the following Chapters). For each ADT we define an *invalidated-by* conflict relation such that operations p' and p are related if inserting p' into some valid execution ending with p renders the execution invalid. Using a generalization of serializability to ADTs [18, 63], Herlihy and Weihl prove several concurrency control algorithms, pessimistic, optimistic [136] and hybrid, using only the algebraic properties[9] of (arbitrary) conflict relations. The theory can also be applied to systems where objects use heterogeneous concurrency control methods. This may aid the interfacing of highly concurrent shared-memory hosts with Atomic Networks, since locking is often the preferred method for concurrency control within such hosts.

The theory can provide a foundation for deriving conflict relations for ADTs in systems based on Atomic Transfer. Transactions whose operations do not conflict may execute in parallel, while conflicting transactions must be totally ordered. Table 4.1 shows an example conflict relation, for a "register" ADT with only simple read and write operations. Operations are shown as invocation/reply pairs, and the operation in a row depend on the operation in a column when the condition in that row/column is true.

	read()/v	write(v)/OK
read()/v'		$v \neq v'$
write(v')/OK		

Table 4.1: Invalidated-by relation for a simple read / write register.

A read can be invalidated by an earlier write using a different value, so a read depends on such writes. If we replace $v \neq v'$ with "true", this corresponds to classic serializability. But invalidated-by relations may also be defined in terms of return values of operations, which can be exploited to yield smaller relations, that is: that define fewer operations as conflicting. For example, the example from Herlihy [136] in Table 4.2 gives a conflict relation for a bank account ADT, with debit and credit operations where the latter can return "over" to indicate insufficient funds for a debit.

Credits can proceed in parallel, but one debit may invalidate another, by lowering the balance so that the latter results in overdraft. A debit that results in an overdraft, on the other hand, conflicts with a credit, since the credit may turn the debit into a successful one. A conflict relation using only reads and writes would have to define

[9]The fact that they are *serial dependency relations* [19]

	credit(n)/OK	debit(n)/OK	debit(n)/over
credit(m)/OK			
debit(m)/OK		true	
debit(m)/over	true		

Table 4.2: Invalidated-by relation for a bank account ADT.

debits as being in conflict. This relation could be extended further to allow some degree of concurrency among credits [136, 137], should the need arise.

To use this type of conflict detection with AT caching, a requester would include its expected replies in its requests, that is: the replies generated by the local execution using cached state. Conversely, responses would include the invocation for each of the replies they contain. Switches would then check whether operations on the same datum conflict or not, which can be done efficiently by arranging operation codes in bitmap masks, for example (see Chapter 8 for one possible approach).

Striking a good balance between ADT operation expressivity and the complexity of the implementation of conflict checking on switches may be a challenge. In general, we expect programmers to re-use existing ADTs and conflict relations more often than they invent new ones. In many cases the choices might be constrained by the available programming languages, compilers and libraries, as well as the conflict relation types supported on the switches in a network.

In summary, basing our execution model on abstract request is a better fit with abstraction in programming languages, defined in terms of queues, sets, relations and other ADTs instead of simple reads and writes. It also and creates opportunities for networking and concurrency control optimizations, which can be used to solve "hot spot" problems without abandoning atomicity.

Chapter 5

Dynamic Subscriptions

Our models have hitherto used the $qHop_i$ and $rHops_i$ relations for message forwarding and assumed that requesters are subscribed to every message of each responder they call throughout any execution. This section relaxes these impractical assumptions, allowing requesters to dynamically vary their subscriptions. Furthermore, they may subscribe to only a subset of a responder's responses. We show that our dynamic subscription protocol preserves Atomic Transfer. We also relax the assumption that switches never drop messages, but use Atomic Transfer to prevent conflicts from escaping detection when this happens. This provides a safe way for switches to discard responses during overload or error conditions, for example.

5.1 Subscriptions and Conflicts

A static switch i of Chapter 3 forwards messages based on the static qHop_i and r$Hops_i$ relations. A *dynamic switch* i, on the other hand, uses dynamic relations stored in its *qHop* and *rHops* fields. Requesters configure these fields using a multicast subscription protocol, establishing and tearing down the network paths that carry their requests and the responses to those requests. In general, a requester a must provide its home switch with some representation of the domain $Q_a \subseteq Q$ of requests that the requester wishes to be able to make, as well as some representation of the set $R_a \subseteq R$ of responses that conflict with requests in Q_a. Our particular protocol's representation for subscription requests in fact captures both domains.

The main restriction on a multicast protocol for atomic switches is that it preserve Conflict Locality. Restated in terms of the new fields, Conflict Locality requires for each pair of adjacent switches i and j that if an entry $(q,\ j)$ is present in $i.qHop$, then for every response r such that *conflicts*(q, r) there is an entry (r, i) in $j.rHops$. Since fields in distinct switches cannot be updated atomically, this means that when configuring a forwarding hop from i to j, the conflicting response entries must be added to *j.rHops* before the request entry for q is added to *i.qHop*. Conversely, when tearing down a forwarding hop from i to j, the entry for q must be removed from *i.qHop* before the corresponding entries of *j.rHops* are removed. This is illustrated in Figure 5.1. As we will show, if Conflict Locality is upheld on the forwarding path *fp*(q) of a request q then the Atomic Transfer safety guarantees of Theorem 1 hold for q.

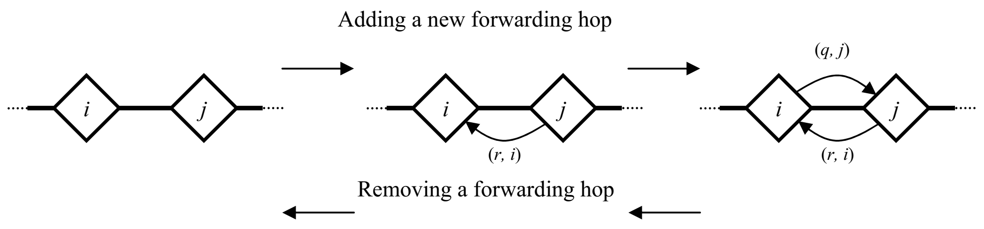


Figure 5.1: configuring a forwarding hop for conflicting request/response q and r.

Instead of developing Dynamic Atomic Transfer in terms of purely abstract requests and responses, we do it for a system where the dependencies and effects of requests and responses can be related to independent subcomponents of responder states. We introduce the notion of names and values, for this purpose, basically corresponding to variable names and values (or seen another way, object identities and states).

The refined switch introduced in this section forwards messages based on the names contained in them. As well as simplifying our presentation, names abstract away from host identities, allowing seamless subscriptions to subsets of host names as well as to names hosted across multiple hosts. The performance objectives informing the design of our subscription protocol are:

1. To minimize the delay from when a requester host initiates a subscription to a responder host until the requester can start issuing request to that responder.

2. To keep that delay relatively independent of the load on the responder and the rate at which the responder is sending out responses.

3. To keep that delay relatively independent of the number of requesters concurrently in the process of subscribing to the responder.

The protocol defined in this Chapter does well with respect to the first of these objectives, but the second and third one are only fully addressed in the scalable refinement of the algorithm presented in Chapter 7.

5.2 Names and Conflicts

To support *partial subscriptions* to subsets of the responses sent by a host, the host's state must be partitioned in such a way that a requester can execute correctly while observing only responses for those partitions containing data relevant to the requester. A relational database, for example, could be partitioned along tables, rows and/or columns, while hierarchical data such as XML files or virtual worlds may be partitioned along sub-hierarchies.

We use a simple scheme for partitioning hosts, akin to those commonly used for operational definition of simple programming languages. For each host $b \in \mathit{HOSTS}$, let NAMES_b be some countable set of *names*, disjoint from the name sets of all other hosts. Let VALUES be the set of *values* for names. Let NAMES be the union of all NAMES_b sets, for each $b \in \mathit{HOSTS}$, and for any $n \in \mathit{NAMES}$ let $\mathsf{host}(n)$ denote the host $b \in \mathit{HOSTS}$ such that $n \in \mathit{NAMES}_b$. We say that $\mathsf{host}(n)$ is the *host of* n, or that name n is *hosted on* $\mathsf{host}(n)$. Note that we do not impose types on names or values, as this is outside our scope.

For every request $q \in Q$, we define the *dependency set* of q, denoted $\mathit{depends}(q)$, as the subset of NAMES_b of host $b = \mathit{destination}(q)$ whose values may affect the execution of q. For every request $r \in R$, we define the *effect set* of r, denoted $\mathit{effects}(r)$, as the subset of NAMES_b of responder $b = \mathit{sender}(r)$ potentially affected by r. While Chapter 6 gives a more concrete refinement for a dynamic atomic cache system, this section does not further specify the semantics of dependency or effect sets. Our only requirement for now is that conflicting requests and responses always have a name in common, that is:

Assumption 5.1 *Names and Conflicts: for any $q \in Q$, $r \in R$: $\mathit{conflicts}(q, r) \Rightarrow \mathit{effects}(r) \cap \mathit{depends}(q) \neq \emptyset$.*

Note that the converse need not be true in general; even if we have $\mathit{effects}(r) \cap \mathit{depends}(q) \neq \emptyset$ there may exist responder states where response r does not affect the outcome of executing q.

The reversal of the implication, however, does tell us that if $\mathit{effects}(r) \cap \mathit{depends}(q) = \emptyset$ then q does not conflict with r. Hence, the set of all responses whose effect includes some name in the dependency set of q contains all responses that conflict with q. Formally, for any set $N_b \subseteq \mathit{NAMES}_b$ for some $b \in \mathit{HOSTS}$, let $\mathit{effect\text{-}resp}(N_b)$ be the set $\{\ r \in R \mid \mathit{effects}(r) \cap N_b \neq \emptyset\ \}$. Then:

Corollary 5.1 $\mathit{conflicts}(q, r) \Rightarrow r \in \mathit{effect\text{-}resp}(\mathit{depends}(q))$.

Hence, if a host issues a request q and the host is subscribed to $\mathit{depends}(q)$, then it is subscribed to all responses that conflict with q and any conflict will be detected. This is the crucial connection between name sets, requests and responses upon which our Dynamic Atomic multicast subscription protocol depends.

5.3 Names, Forwarding and Subscriptions

This section discusses the setting up and tearing down of subscriptions. While multicasting is not our focus, we fundamentally assume it as the basis for Atomic Transfer and need to include subscription management in our models in order to reason about their interactions with Atomic Transfer. We strive keep the multicasting aspects of our protocol abstract and simple, yet somewhat representative of well-known multicast algorithms [138, 139, 140].

Dynamic switches forward requests and responses based on their *name sets*: dependency sets and effect sets, respectively. To abstract from network routing, for each switch $i \in \mathit{SWITCHES}$ let nHop_i be a total *routing oracle function* : $\mathit{NAMES} \rightarrow \mathit{CHANNELS}_i$, mapping from each name n to a channel that leads toward $\mathit{host}(n)$, for example the next channel on the shortest path towards the host. It represents the network knowledge that a switch implementation would acquire through some unspecified routing protocol. We assume the absence of circular forwarding paths. Note that since nHop_i is a function, there is a unique forwarding path from each node to the host of a name. As a notational aid, for any $c \in \mathit{CHANNELS}_i$ let $\mathit{nHop}_i(c)$ denote the set of names that i forwards on channel c, that is: $\{\ n \in \mathit{NAMES} : (n,\ c) \in \mathit{nHop}_i\}$.

Since hosts may potentially subscribe to any set of names and issue requests depending on them, we redefine qHop_i in terms of nHop_i, that is: $\mathit{qHop}_i = \{\ (q,\ c) \mid \mathit{depends}(q) \subseteq \mathit{nHop}_i(c)\ \}$. Since nHop_i is a function, request forwarding paths are unique, as in Chapter 3. We also redefine rHops_i, as the relation that forwards all responses along each path away from the sending responder, that is: $\mathit{rHops}_i = \{\ (r,\ c) \mid \mathit{effects}(r) \cap \mathit{nHop}_i(c) = \emptyset\ \}$. Since requests always travel towards their responders, this implies Conflict Locality for qHop_i and rHops_i.

Note that the nHop_i oracle is static and that our model does not capture dynamic route changes. A practical Atomic Network implementation would have to adapt to changes in network topology, but any re-routing must be carefully choreographed as to preserve Conflict Locality. Since switches forward messages based on data names, network re-routing, renaming of data and migration of data between hosts must be made to appear atomic with the corresponding changes in the forwarding tables on switches. We leave investigation of these issues to future work.

A host or switch $o \in \mathit{NODES}$ subscribes to some set of names at each adjacent switch i, termed *the subscription* of o at i and denoted by $\mathit{sub}_o(i)$. To add names to its subscription, o sends a subscription request message to i, specifying the set of names it now wants forwarded to itself. If switch i is not already subscribed to some of the names requested then i must in turn add to its subscription(s) at adjacent switches. This chain of events may lead all the way to the home switches of one or more responder hosts. Home switches receive all their hosts's responses by our definition.

However, as in most scalable multicast subscription protocols, a subscription request only travels as far as needed; as soon as it reaches a switch that is already subscribed to the requested names, no further subscription requests are sent by that switch.

A DynamicSwitch automaton s_i, as defined in Section 5.6, has a field $outNames_c$ for each $c \in CHANNELS_i$, storing the subscription $sub_j(i)$ of the other node j incident to c, and a field $inNames_c$ storing a copy of the switches own subscription $sub_i(j)$ at j. The field names remind us that the fields contain the names being sent "out" to other switches and the names that funnel "in" to the switch, respectively. Note that at any time, the union of the $inNames_c$ fields of s_i must contain the union of the $outNames_c$ fields of s_i, in order for s_i to receive all the data needed by its subscribers.

In the I/O Automata we define the *qHop* and *rHops* fields of a switch i as variant functions of the subscriptions of i. There is an entry $(q, d) \in i.qHop$ for each $(q, d) \in qHop_i$ where the dependency set of q is contained in $i.outNames_c$, with c the channel on which q is received at i. There is an entry $(r, c) \in i.rHops$ for each $(r, c) \in \mathrm{r}Hops_i$ such that $i.outNames_c$ contains a name affected by r.

Let a be a requester that subscribes to the name set N_{Qa} containing all names in the union of the dependencies of the set of requests $Q_a \subseteq Q$ that a may issue, that is: $N_{Qa} = \{\ n \in NAMES \mid \exists\ q \in Q_a\colon\ n \in depends(q)\ \}$. Switch i forwards each response r it receives onto each channel $c \in CHANNELS_i$ such that subscription $outNames_c$ has a name in common with *effects*(r). By Corollary 5.1, if a is subscribed to Qa then switch i will forward every response conflicting with requests issued by a a.

A switch i does not add a name n to its subscription set $i.inNames_c$ for channel $c = nHop_i(n)$ until it receives a subscription confirmation messages for n from the switch j incident to c. This ensures that a name n is added to $j.outNames_c$ and hence the $j.rHops$ relation of j before it is added to $i.qHop$, preserving Conflict Locality. A confirmation message with a name set N therefore signals that a forwarding path has been established all the way to the host(s) of N.

5.4 Overflow, Failures and Purging

Conflict Locality implies that all responses must be forwarded to each receiver and never be dropped. This may be hard to achieve in practice, since bursts in traffic can exceed network capacity, causing packet queues on switches to overflow and forcing switches to discard packets. Switches can preclude this scenario through some form of lossless flow control [141], especially if the flow control creates *backpressure* on responders to slow down their execution rate and response generation as capacity nears exhaustion. While such schemes may apply within a single autonomously managed subnetwork, they may not be applicable in general, since they can lead to the slowest receiver dictating the flow of responses in the entire network of subscribers[10]. Last but not least, real-world switches and links occasionally fail, losing all responses buffered. For Atomic Transfer to be practical, it must be able to handle these issues.

The solution comes from noting that a response r can be safely dropped as long as all in-flight requests conflicting with r are dropped as well. We achieve this by *purging* names from subscriptions at the dropping switch, as to prevent conflicting requests from being forwarded. In effect, the switch can safely breach its obligation to forward a response by severing the paths for conflicting requests from the response's subscribers. Allowing switches to unilaterally purge subscriptions has another practical use, since it supports the common practice of maintaining subscription and forwarding tables in switches as *soft-state*, that expires and is discarded after some length of time and/or disuse.

There is a potential pitfall, though: a forwarding path could be re-established without some subscriber noticing that it was severed in the first place. The subscriber might then receive a stream of responses where one or more responses have been omitted, which could violate Atomic Transfer of a request conflicting with names in an omitted response. In addition to local purging and severing of forwarding paths, all affected requesters must be made aware of the purge.

We handle the problem as follows. When a switch unilaterally purges a subscription to some set of names, it sends a special cancellation notification message on all affected channels, which ultimately propagates to all subscribers of the names.

[10]Allowing any receiver to dictate the flow would also enable easy denial-of-service attacks.

These notifications are defined as responses with the purged names in their effect set, and they are defined to conflict with all requests that have a name in common with that set. As we will show, cancellation responses ensure that all request/response conflicts are detected, regardless of the timing of a forwarding path's severing and re-establishment.

A switch that fails can similarly notify its neighbors and subscribers when it recovers again, by sending a cancellation response for the universal set *NAMES* on each of its channels. This causes all subscriptions passing through the switch to be reset and all requests heading its way to be dropped. We do not explicitly model a switch failing, but such an event can be represented in an execution as a sequence of consecutive purge events for *NAMES* and all queued responses on each of its channels.

5.5 Subscription Consistency

To model subscription processing messages, let $SQ \subseteq Q$, $SR \subseteq R$ be the designated sets of *subscription request messages* and *subscription response messages*, respectively. A subscription request contains a set of names that should be added to the sender's subscription and a set of names that should be removed from it. Let total function $SQ(N_s, N_u)$: $2^{NAMES} \times 2^{NAMES} \rightarrow SQ$ returns the unique message $sq \in SQ$ such that functions sub: $SQ \rightarrow 2^{NAMES}$, $unsub$: $SQ \rightarrow 2^{NAMES}$ and $names$: $SQ \rightarrow 2^{NAMES}$ map sq to N_s, N_u and $N_s \cup N_u$, respectively. A subscription response contains the set of names that were added or removed from a subscription (due to an earlier subscription request, for example) as well as the set of names that are pending for subscription but whose forwarding path has not yet been fully established. Let total function $SR(N_s,\ N_u,\ N_p)$: $2^{NAMES} \times 2^{NAMES} \times 2^{NAMES} \rightarrow SR$ return the unique message $sr \in SR$ such that functions sub: $SR \rightarrow 2^{NAMES}$, $unsub$: $SR \rightarrow 2^{NAMES}$, $pend$: $SR \rightarrow 2^{NAMES}$ and $names$: $SQ \rightarrow 2^{NAMES}$ map sr to N_s, N_u, N_p and $N_s \cup N_u \cup N_p$, respectively. The SQ and SR functions are only defined for mutually exclusive argument sets, i.e. with no names in common.

We permit a DynamicSwitch i to arbitrarily drop a queued request, using the non-deterministic internal action drop$(m \in Q)_{c,i}$. We keep assuming reliable FIFO channels, as modeling the acknowledgement/retry protocols generally used to overcome failure-prone links adds no insight to our discussion and is better done in conjunction with modeling of lower-level network protocols. We leave the question of how hosts learn about dropped requests and responses undefined, but one solution would be to use a time-out and a retransmission.

Modeling dropped requests increases the complexity of subscription processing, since subscription requests may then also be dropped. Switches and applications must resend subscription requests to overcome such message loss, but the asynchronous modification and unilateral purging of subscriptions invites race conditions. Further race conditions may result from the fact that switches cannot satisfy all subscription requests immediately but must sometimes add names to their own subscriptions first. Subscription and unsubscription requests and responses can interact in subtle ways with such pending subscriptions. Furthermore, even if all switches were to eventually reach a state where their *inNames* were consistent with the *outNames* of all adjacent nodes, this does not suffice. The reason is that we must also ensure Conflict Locality at all times, lest a conflicting request were to escape detection.

We address the problem using Atomic Transfer. Using the terminology of Chapter 4, we let each switch act like a data server with respect to its *outNames* sets and like an application with respect to its *inNames* sets, treating each $inNames_c$ field as a cached version of the $outNames_c$ field of the adjacent node incident to channel c. A subscription request inbound on channel c, therefore, is simply a request to modify $outNames_c$, with the switch replying with a subscription response encoding the changes, similar to how a data server would respond. For each name n in the request, the response either confirms the requested change of n to a subscribed or unsubscribed status or else to a *pending* status, meaning that the switch will complete the change later, upon adding n to its own subscription. The switch completes the change in a non-deterministic action predicated on *inNames* sets containing names tagged as pending in *outNames* sets.

We define switches as being permanently "subscribed" to the *outNames* sets of their neighbors. This guarantees Conflict Locality for subscription requests and responses between adjacent switches and allows us to invoke Theorem 3.1 for requests and responses going between the switches. We take as our conflict relation the relation such that all subscription requests and responses that have names in common conflict. We then use similar argument as used for Theorem 4.1 to show that all (successful) subscription requests are based on an up-to-date view of adjacent subscriptions, leading to consistent behavior. We require each subscription response to conflict with all requests sharing a name with, that is:

Assumption 5.2 *For all* $q \in Q$, $sr \in SR$:
$\textit{names}(sr) \cap \textit{names}(q) \neq \emptyset \Rightarrow \textit{conflicts}(q, sr)$

It would suffice for ordinary requests to conflict with the *unsub* names of subscription responses, for the purposes of ensuring safety of subscription purges. Conflicts with other parts of a subscription response would imply an erroneous sender, forwarding a request without being fully subscribed to its dependency set. But since we will rule out such erroneous nodes, we use this simpler definition.

5.6 DynamicSwitch$_i$

We now define DynamicSwitch$_i$, the dynamic version of the AtomicSwitch$_i$ automaton. We model the decision of a switch i to purge subscriptions on a channel $c \in \textit{CHANNELS}_i$ with the non-deterministic internal action purge$(N)_{i,c}$, which is enabled whenever the subscription of channel c is non-empty. To capture the steps a real switch might take in response to a channel failure or overflow, we allow the action to simultaneously purge some subset of the responses queued for c, but only if no adjacent node will be subscribed to any of those responses following the purge. Purged responses are replaced in the queue with a subscription cancellation message for N.

Dynamic switches also forward messages in the sets $VQ \subseteq Q$, $VR \subseteq R$ of *value read request messages* (value requests) and *value response messages* (value messages), respectively, disjoint with subscription requests and responses, respectively. These messages are used in the Dynamic Atomic Cache system of Chapter 6, for hosts to request the current values of names and respond with messages carrying name values. For the purposes of this section, all we need to know is that a value request or response message m has an associated set of names $\textit{names}(m) \subseteq \textit{NAMES}$, and that for any

$N \subseteq \mathit{NAMES}$ we let $VQ/VR(N)$ denote the unique value request/response message such that $\mathit{names}(m) = N$. Furthermore, the *conflicts* relation does not include any members of VQ or VR, so these messages never conflict with other messages.

Note that the receive$(q \in Q \setminus SQ \setminus VQ)_{c,i}$, receive$(r \in R \setminus SR \setminus VR)_{c,i}$ and send$(m \in M)_{c,i}$ actions of DynamicSwitch$_i$, for normal requests and responses, are virtually identical to those of AtomicSwitch$_i$. Indeed, our proofs essentially argue that if all subscriptions for $\mathit{depends}(q)$ on the forwarding path $\mathit{fp}(q)$ of a request q are stable during an execution interval, then the dynamic switches of $\mathit{fp}(q)$ behave as if the path were made up of static atomic switches. When subscriptions are unstable and change, we show that cancellation responses ensure that Atomic Transfer safety is preserved.

DynamicSwitch$_i$

Models the behavior of an atomic switch supporting dynamic name subscriptions. Note that some fields are defined as variant functions of the switch's current subscription state. Let *SUBSTATUS* denote the set { *sub, unsub, pend* }, of possible *subscription statuses* of a name at a node: subscribed, not subscribed or pending (waiting to become subscribed).

State:

for each $c \in \mathit{CHANNELS}_i$

- $\mathit{outQueue}_c$: same as in AtomicSwitch$_i$.
- $\mathit{responses}_c$: same as in AtomicSwitch$_i$
- $\mathit{outNames}_c$: a total function $\mathit{NAMES} \rightarrow \mathit{SUBSTATUS}$
  - returning the subscription status of n at i, initially *unsub*, for all $n \in \mathit{NAMES}$.
- $\mathit{inNames}_c$: a total function $\mathit{NAMES} \rightarrow \mathit{SUBSTATUS}$
  - returning the subscription status i caches for $o.\mathit{outNames}_c$,
  - where o is the other node incident to c.
  - If $o \in \mathit{SWITCHES}$ then initially *unsub*, for all $n \in \mathit{NAMES}$.
  - Else, initially $\mathit{inNames}_c(n) = \mathit{sub}$ for each $n \in \mathit{nHop}_i(c)$, *unsub* for the rest.

Derived State:

```
for each $c \in \mathit{CHANNELS}_i$
    // the subscriptions this switch serves / receives
    $\mathit{outSub}_c$ / $\mathit{inSub}_c$ = { $n \in \mathit{NAMES}$ | $\mathit{outNames}_c(n)$ / $\mathit{inNames}_c(n) = \mathit{sub}$ }.
    // the subscriptions pending at this switch / other switches
    $\mathit{outPend}_c$ / $\mathit{inPend}_c$ = { $n \in \mathit{NAMES}$ | $\mathit{outNames}_c(n) = \mathit{pend}$ / $\mathit{inNames}_c(n) = \mathit{pend}$ }.
$\mathit{inSubs}/\mathit{outSubs}/\mathit{inPends}/\mathit{outPends}$: the union of $\mathit{inSub}_c/\mathit{outSub}_c/\mathit{inPend}_c/\mathit{outPend}_c$,
$for\ each\ c \in CHANNELS_i$.
// the subset of $\mathit{qHop}_i$ for which the switch has sufficient subscriptions
$\mathit{qHop}$: $Q \rightarrow \mathit{CHANNELS}_i$ = { $(q, d) \in \mathit{qHop}_i$ | $\mathit{depends}(q) \subseteq \mathit{outSub}_c$ },
    where $c$ is the channel on which $q$ is received at $i$.
// the subset of $\mathit{rHops}_i$ for which the switch has subscribers
$\mathit{rHops}$: $R \times \mathit{CHANNELS}_i$ = { $(r, d) \in \mathit{rHops}_i$ | $\mathit{outSub}_d \cap \mathit{effects}(r) \neq \emptyset$ }
```

Input actions:

```
receive$(q \in Q \setminus SQ \setminus VQ)_{c,i}$
Effect:
    // the same as in AtomicSwitch$_i$
    if $\exists\ r \in \mathit{responses}_c$ such that $\mathit{conflicts}(q, r)$
        $\mathit{handleConflict}(q)$
    else let $d = \mathit{qHop}(q)$ in
        $\mathit{outQueue}'_d = \mathit{outQueue}_d \cdot q$

receive$(r \in R \setminus SR \setminus VR)_{c,i}$
Effect:
    // the same as in AtomicSwitch$_i$
    $\mathit{outQueue}'_c = \mathit{outQueue}_c \cdot \mathrm{ack}(r)$
    for each $d$ such that $(r, d) \in \mathit{rHops}$
        $\mathit{outQueue}'_d = \mathit{outQueue}_d \cdot r$
        $\mathit{responses}'_d = \mathit{responses}_d \cup \{ r \}$

receive$(a \in A)_{c,i}$
Effect:
    // the same as in AtomicSwitch$_i$
    $responses'_c = responses_c \setminus \{message(a)\}$

receive($sq \in SQ$)$_{c,i}$
Effect:
    if $\exists\ r \in responses_c$ such that $conflicts(sq, r)$
        $handleConflict(sq)$
    else
        // update the subscription status of the names
        let $sub_{ready} = sub(sq) \cap inSubs$, $sub_{pend} = sub(sq) \setminus inSubs$ in
            $\forall n \in NAMES$: $outNames'_c(n) =$
                $outNames_c(n)$, if $n \notin names(sq)$,
                $sub$, if $n \in sub_{ready}$,
                $pend$, if $n \in sub_{pend}$,
                $unsub$, if $n \in unsub(sq)$.
        // send back a subscription response for all affected names
        let $sr = SR(sub_{ready}, sub_{pend}, unsub(sq))$ in
            $outQueue'_c = outQueue_c \cdot sr$
            $responses'_c = responses_c \cup \{\ sr\ \}$
        // send out subscription requests for newly pending names
        for each $d \in CHANNELS_i$ where $sub_{pend} \cap nHop_i(d) \neq \emptyset$
            $outQueue'_d = outQueue_d \cdot SQ(sub_{pend} \cap nHop_i(d), \emptyset)$

receive($sr \in SR$)$_{c,i}$
Effect:
    // enqueue an ACK back for the response
    $outQueue'_c = outQueue_c \cdot ack(sr)$
    // update the cached status of the names
    $\forall n \in NAMES$: $inNames'_c(n) =$
        $inNames_c(n)$, if $n \notin names(sr)$,
        $sub$, if $n \in sub(sr)$,
        $pend$, if $n \in pend(sr)$,
        $unsub$, if $n \in unsub(sr)$.
    // for each channel whose subscription is affected by this change
    let $sub_d = sub(sr) \cap nHop_i(d)$, $unsub_d = unsub(sr) \cap nHop_i(d)$ in
        for each $d \in CHANNELS_i$ where $sub_d \cup unsub_d \neq \emptyset$
            // update the subscriptions, for names we've lost or added to ours
            $\forall n \in NAMES$: $outNames'_d(n) =$
                $outNames_d(n)$, if $n \notin sub_d \cup unsub_d$,
                $sub$, if $n \in sub_d$,
                $unsub$, if $n \in unsub_d$.
            // send a subscription response, notifying of the change
            let $sr = SR(sub_d, \emptyset, unsub_d)$ in
                $outQueue'_d = outQueue_d \cdot sr$
                $responses'_d = responses_d \cup \{ sr \}$
```

receive($vq \in VQ$)$_{c,i}$

Effect:

```
    // forward value request
    for each $d$ such that $names(vq) \cap nHop_i(d) \neq \emptyset$
        $outQueue'_d = outQueue_d \cdot vq$
```

receive($vr \in VR$)$_{c,i}$

Effect:

```
    // forward value response
    for each $d$ such that $names(vr) \cap outSub_d \neq \emptyset$
        $outQueue'_d = outQueue_d \cdot vr$
```

Internal actions:

purge($N \subseteq NAMES$)$_{c,i}$

Precondition:

```
    $N \subseteq (outSub_c \cup outPend_c) \wedge N \neq \emptyset$
```

Effect:

```
    // set the status of all the names to unsubscribed
    $\forall n \in NAMES$:
        $outNames'_c(n) =$
            $outNames_c(n)$ if $n \notin N$,
            $unsub$, otherwise.
    // remove and notify about cancellation
    // while possibly purging a suffix of responses no node subscribes to anymore
    let $\rho_c$ be any suffix of $outQueue_c$ where $\forall r \in \rho_c$ : $effects(r) \cap outSub'_c = \emptyset$ in
        let $R_c = \{ r \in \rho_c \} \setminus SR$, $sr = SR(\emptyset, \emptyset, N)$ in
            $outQueue'_c = (outQueue_c \setminus R_c) \cdot sr$
            $responses'_c = (responses_c \setminus R_c) \cup \{ sr \}$
```

drop$(q \in Q)_{c,i}$
Precondition:

```
    $q \in outQueue_c$
```

Effect:

```
    $outQueue'_c = outQueue_c \setminus \{ q \}$
```

unsubscribe$(N \subseteq NAMES)_{c,i}$
Precondition:

```
    $N \subseteq (inSubs_c \cup inPends_c) \setminus outSubs \land N \neq \emptyset$
```

Effect:

```
    // spontaneously request to unsubscribe to some names
    $outQueue'_c = outQueue_c \cdot SQ(\emptyset, N)$
```

Output actions:

send$(m \in M)_{c,i}$
Precondition:

```
    $head(\text{outQueue}_c) = m$
```

Effect:

```
    $\text{outQueue}'_c = tail(outQueue_c)$
```

The receive$(q \in Q \setminus SQ \setminus VQ)_{c,i}$ action is the same as the receive$(q \in Q)_{c,i}$ action of AtomicSwitch$_i$, except it refers to the *qHop* field instead of the static $qHop_i$ relation. We do not specify the case when $(q, d) \notin qHop$ since, as our proofs show, this case cannot occur in the system we model unless there is a cancellation response in $\mathit{responses}_c$ that conflicts with q.

The receive$(r \in R \setminus SR \setminus VR)_{c,i}$ and send$(m \in M)_{c,i}$ actions are exactly the same as the receive$(r \in R)_{c,i}$ and send$(m \in M)_{c,i}$ actions of AtomicSwitch$_i$, respectively, except that receive$(r \in R \setminus SR \setminus VR)_{c,i}$ refers to the *rHops* field instead of the static rHops_i relation.

The receive$(a \in A)_{c,i}$ action is the same as in AtomicSwitch$_i$.

The receive$(sq \in SQ)_{c,i}$ action receives a subscription request from the node incident to channel c and modifies the subscription $\mathit{outNames}_c$ for c. Note that the action conflict-checks the request, just like any other request. It immediately tags the names of $\mathit{unsub}(sq)$ as not subscribed. However, it only tags a name n in $\mathit{sub}(sq)$ as subscribed if the switch is already subscribed to n. Otherwise it tags n as pending, not upgrading it to subscribed status until the switch itself becomes subscribed to n. The action enqueues back a subscription response, containing the new status of the requested names. Also, for each channel leading to a newly pending name(s), it sends a subscription requests for the name(s).

The receive$(sr \in SR)_{c,i}$ action receives a response notifying of changes to the sender's $\mathit{outNames}_c$ set, and updates the "cached" $\mathit{inNames}_c$ set of i accordingly. The changes could be due to an earlier subscription request from i or due to a purge event on some other switch. The action acknowledges the response, like any other response message. For each channel subscribed to name(s) whose status is changing to subscribed or unsubscribed, the action updates the channel's subscription with the new status of the name(s) and enqueues a subscription response with the changes. There is no need to forward names that are becoming pending: these names will already be tagged as such in any *outNames* set they are present in.

The receive$(vq \in VQ)_{c,i}$ action receives a value request message and forwards it toward the appropriate host(s). The action could split the message and forward different subsets of it on different channels, but we model it simply here.

The receive$(vr \in VR)_{c,i}$ action receives a value response message and forwards it towards the appropriate subscribers. The action could forward different, possibly overlapping subsets of the message onto different channels, but we model it simply here.

The purge$(N)_{c,i}$ action models the switch-initiated purging of a subset of the subscription on some channel c, possibly along with one or more of the responses enqueued for c, as captured by the non-deterministic R_c set. A soft-state subscription time-out might remove no responses, while a link failure or overflow might remove all of them. The action sends a cancellation subscription response message for any names removed and adds it to the $\mathit{responses}_c$ set.

As the definition of R_c reflects, a response can be removed only if the channel will no longer be subscribed to any name in its effect set, as this guarantees that no conflicting requests received via c will be forwarded by the switch. Also note that the action removes responses from the tail of the queue only. This ensures that subscribers never see a sequence of responses where one or more responses have been omitted. This is not required for Conflict Locality, as any request depending on an omitted response will be detected as a conflict with the corresponding cancellation response. However, this stronger guarantee is needed in Chapter 6 and may be generally useful for other types of end-system hosts as well, since they may otherwise receive requests that are not applicable to their current state, before receiving the cancellation response.

The drop$(q \in Q)_{c,i}$ action models the switch dropping a request, for example due to an overflowing packet queue. As mentioned before, we opt to model such drops in the switch rather than the channels. These actions highlight the fact that delivery of these types of messages is not guaranteed, and such guarantees are not needed for safety.

The unsubscribe$(N \subseteq \mathit{NAMES})_{c,i}$ action takes some subset of set of names the switch needlessly subscribes to on c (after one or more adjacent nodes have unsubscribed to the names, for example) and sends a request to remove them from its subscription. It does not remove them from its subscription set immediately, as the subscription request may yet dropped due to a conflicting subscription response. We model unsubscriptions as a separate non-deterministic action to give implementations the freedom to perform them at their own chosen time scales and granularity levels. They could, for example, sometimes defer unsubscriptions in anticipation of future re-subscriptions, to avoid subscripton/unsubscription hysteresis.

5.7 Properties and Proofs

Let *DAS* be the I/O Automaton composed of an DynamicSwitch$_i$ automaton for each switch $i \in$ *SWITCHES* and a Channel$_c$ automaton for each channel $c = (i, j) \in$ *CHANNELS*. The definition of responsibility intervals is the same as in Chapter 3 and Corollary 3.1 from that Chapter still applies, stating that conflicting receives in responsibility intervals never cause sends. In particular, it holds for cancellation responses, since they are a subset of the set R of responses.

The event ordering relation $<_E$ and *causes* relation $\rightarrow_E$ for an execution or execution trace E are the same as before, with the addition to $\rightarrow_E$ of each send$(sr' \in SR)_{d,i}$ event caused by a receive$(sr \in SR)_{c,i}$ event. Note that the message sent is generally not the same as the one received, but the ordering is nonetheless well-defined. We begin with a simple invariant, stating that a switch is subscribed to all names for which adjacent nodes have subscriptions at the switch.

Lemma 5.1 *In every state of DynamicSwitch$_i$: i.outSubs $\subseteq$ i.inSubs.*

Proof: Only the receive$(sq \in SQ)_{c,i}$ and receive$(sr \in SR)_{c,i}$ actions change names in *i.outNames* to subscribed status, and these names are already in *i.inSubs* or are being added to *i.inSubs*, respectively. Only the receive$(sr \in SR)_{c,i}$ action changes names in *i.inNames* to unsubscribed status, and these names are changed to unsubscribed status in *i.outNames* at the same time. □

Conflict Locality for dynamic switches differs slightly from that of Chapter 3, as it is not a constraint on static relations but an invariant on the value of asynchronously varying *qHop* and*rHops* fields in switches. When a switch decides to purge a subscription, for example, the adjacent switches do not learn about it until they receive a subscription response message. We will refer to Conflict Locality as defined in Chapter 3 as Static Conflict Locality, from here on. Our correctness condition, therefore, requires either that Static Conflict Locality holds for every conflicting request and response pair on each pair of adjacent switches, or else that a cancellation subscription response is en route, signaling that the forwarding path has been severed. Formally:

Definition 5.1 *Dynamic Conflict Locality: for any pair of messages $q \in Q$ and $r \in R$ where conflicts(q, r) and any pair of switches $i, j \in$ NODES: $i.qHop(q) = (i, j) \Rightarrow (r, (j, i)) \in j.rHops \vee (\exists\ sr \in j.responses_{(i,j)} \cap SR : \mathit{conflicts}(q, sr))$.*

Our proof strategy is to show that Dynamic Conflict Locality holds in the executions of *DAS*. We then argue that either Static Conflict Locality and hence Theorem 3.1 holds on a request's forwarding path or else the request will have a conflict with a subscription response and be dropped. In fact we show that it is sufficient for Static Conflict Locality to hold on the suffix of the path where the request has not yet been received; a purge on the part of the path "behind" a request does not affect it.

We introduce a new invariant called Name Conflict Locality, which implies Dynamic Conflict Locality but is easier to work with. We begin with precise definitions of what it means for a node to be subscribed to a name.

Definition 5.2 *For any node $o \in$ NODES adjacent to a switch $j \in$ SWITCHES and any $n \in$ NAMES, let subscribed$_{o,j}(n, t)$ be true in state $t \in$ execs(DAS) exactly if $n \in j.outSub_{(o,j)} \vee \exists\ cr \in j.responses_{(o,j)} \cap SR : n \in$ names(cr). We say that o is dynamically subscribed to n in t. For notational convenience, we define subscribed$_{o,c}(n, t)$ to be equivalent to subscribed$_{(o,j)}(n, t)$ for channel $c = (o, j)$.*

A host $a \in$ HOSTS, in particular, is dynamically subscribed to a name $n \in$ NAMES in t, denoted subscribed$_a(n, t)$, exactly if subscribed$_{(a,i)}(n,\ t)$, where i is the home switch of a.

One way of reading these definitions is that either $n \in j.outSub_{(o,j)}$ or else t is in the responsibility interval of a conflicting cancellation response *cr*, that is: $t \in$ *resp-interval*$(cr)_{(o,j),j}$ where $n \in$ *names*(cr).

Name Conflict Locality says that if a switch i believes itself to be subscribed to a name n at an adjacent switch j then either j is forwarding responses affecting n to

i or else j has a cancellation response bound for i that contains n. We define Name Conflict Locality and show that it implies Dynamic Conflict Locality.

Definition 5.3 *Name Conflict Locality holds in a state $t \in$ states(DAS) exactly if for all pairs of switches i, $j \in$ SWITCHES: $n \in i.inSub_{(i,j)} \Rightarrow subscribed_{(i,j)}(n, t)$.*

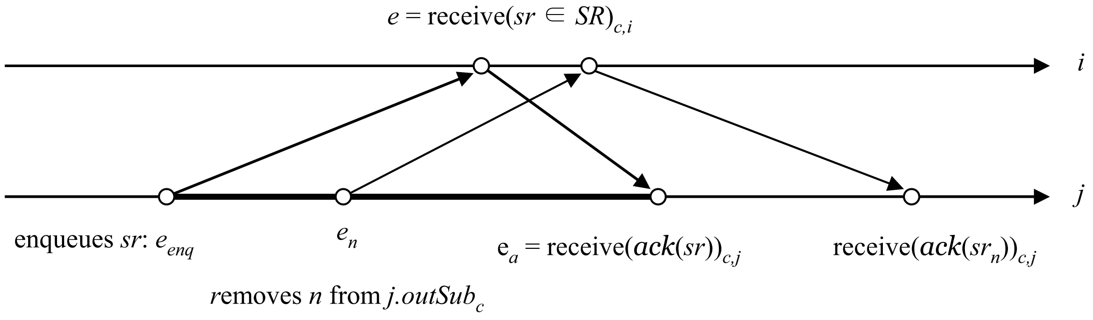
Lemma 5.2 *(Name Conflict Locality implies Dynamic Conflict Locality): if Name Conflict Locality holds in a state of t of DAS then Dynamic Conflict Locality holds in t.*

Proof: Fix any pair of messages $q \in Q$ and $r \in R$ where *conflicts*$(q,\ r)$ and a pair of switches i, $j \in$ *NODES* such that $i.qHop(q) = (i,\ j) = c$. Then *depends*$(q) \subseteq i.outSub_d$ where d is the channel on which i receives q, by the definition of *qHop* (noting also that *depends*$(q) \subseteq nHop_i(c)$, by the definition of *qHop*). By Lemma 5.1, *depends*$(q) \subseteq i.outSubs \subseteq i.inSubs$. By Name Conflict Locality, therefore, for each $n \in$ *depends*(q) either $n \in j.outSub_c$ or $j.responses_c$ holds a cancellation response cr such that $n \in$ *names*(cr). In the first case $(r,\ c) \in j.rHops$, by the definition of $j.rHops$. In the second case *conflicts*$(q,\ cr)$ is true, by the definition of conflicts for cancellation responses. In both cases, the consequent of *Dynamic Conflict Locality* is satisfied □

To show that Name Conflict Locality holds in DAS we use the following Lemma, which says that once a switch enqueues a subscription response confirming the addition of a name to a subscription, the response's receiver is guaranteed to be subscribed to the name until the switch receives the acknowledgement for the response. Recall that the term "causes" refers to the $\rightarrow_X$ causes relation.

Lemma 5.3 *(Safety of Switch Subscription Expansion): Let e be an event receive($sr \in SR)_{c,i}$ in any execution $X \in$ execs(DAS), where i, $j \in$ SWITCHES and $c = (i, j)$. Let e_{enq} be the event enqueuing the sr enabling the send($sr \in SR)_{c,j}$ causing e and let e_a be the receive(ack(sr))$_{c,j}$ caused by the enqueuing of ack(sr) by e (see figure 5.2). Then for each $n \in$ sub(sr) $subscribed_{i,j}(n, t)$ holds in each state t in the interval of X beginning after e_{enq} and ending immediately after e_a, in particular after e.*

Proof: Inspecting DynamicSwitch$_j$, e_{enq} must either a receive$(sr_d \in SR)_{d,j}$ event with $n \in sub(sr_d)$ and some $d \in CHANNELS_j$ or a receive$(sq \in SQ)_{c,j}$ event with $n \in sub(sq)$. By inspecting these actions, it is clear that $n \in j.outSub_c$ in the state immediately after e_{enq}. If n is not removed from $j.outSub_c$ before e_a, then $subscribed_{i,j}(n, t)$ holds as asserted. Otherwise, n is first removed from $j.outSub_c$ after e_{enq} by some intervening event e_n, where $e_{enq} <_X e_n <_X e$. Inspecting DynamicSwitch$_i$, event e_n must be one of receive$(sr' \in SR)_{g,j}$ with $n \in unsub(sr')$ and $g \in CHANNELS_j$, purge$(N_p \subseteq NAMES)_{c,j}$ with $n \in N_p$ or receive$(sq \in SQ)_{c,j}$ with $n \in unsub(sq)$. In each case, e_n adds a subscription response sr_n to $j.responses_c$, where $n \in$ names(sr_n). Since e enqueues $ack(sr)$, by FIFO $e_a <_X$ receive$(ack(sr_n))_{c,j}$. Since subscription responses are never purged, $sr_n \in j.responses_c$ in the interval of X beginning after e_{enq} and ending immediately after e_a, so $subscribed_{i,j}(n, t)$ holds in every state t in that interval □



e = receive($sr \in SR)_{c,i}$
i
j
enqueues sr: e_{enq}
e_n
e_a = receive($ack(sr))_{c,j}$
receive($ack(sr_n))_{c,j}$
removes n from $j.outSub_c$

Figure 5.2: Lemma 5.3 illustrated

We now prove that Name Conflict Locality is invariant in executions of DAS.

Lemma 5.4 *(Name Conflict Locality in DAS): In each state t of X of any execution $X \in$ execs(DAS), Name Conflict Locality holds in t.*

Proof: Fix any $X \in execs(DAS)$ and any pair i, j of distinct switches from $SWITCHES$. Without loss of generality, we use i to denote the switch requesting subscriptions and sending requests, j to denote the switch satisfying subscription requests and forwarding responses and c to denote channel (i, j). We proceed by induction on the prefixes of X, to show that $n \in i.inSub_{(i,j)} \Rightarrow subscribed_{(i,j)}(n, t)$ for any $t \in X$ and $n \in NAMES$, as required. As our base case, Name Conflict Locality holds in the start state of X, since the $inSub_c$ sets of the switches are empty. For the inductive step, assume the theorem holds in the final state t_x of some prefix X' of X such that $X' \cdot e \cdot t = X$, where $(t_x, e, t) \in trans(DAS)$. We show that DAS *preserves* Name Conflict Locality, that is: the invariant still holds in the state t following e, which completes the induction. We claim that for each action e of DynamicSwitch$_i$ enabled in t_x the invariant still holds in t because:

1. If $e =$ receive$(q \in Q \setminus SQ \setminus VQ)_{c,j}$, receive$(vq \in VQ)_{c,j}$, receive$(vr \in VR)_{c,j}$, drop$(q \in Q)_{c,j}$, unsubscribe$(N \subseteq NAMES)_{c,i}$ or send$(m \in M)_{c,j}$, then e affects no $j.outSub$, $j.inSub$ or $j.responses$ set, preserving the invariant.

2. If $e =$ receive$(r \in R \setminus SR \setminus VR)_{c,j}$, then e affects no $j.outSub$ or $j.inSub$ sets, preserving the invariant. It may add r to one or more $j.responses$ fields, but adding responses to a *responses* set cannot falsify $subscribed_{i,j}(n, t)$, preserving the invariant.

3. If $e =$ receive$(sq \in SQ)_{c,j}$ then e affects no $j.inSub$ fields and e can only add a response to $j.responses$, preserving the invariant. It may add names from $sub(sq)$ to $j.outSub_c$, preserving the invariant, but it may also remove names $N = unsub(sq)$ from $j.outSub_c$. But then e also adds a cancellation response cr with $unsub(cr) = N$ to $j.responses_c$, so the invariant is preserved.

4. If $e =$ receive$(sr \in SR)_{c,i}$ then e only adds response to $i.responses$ fields, preserving the invariant. It may also add names from $sub(sr)$ to $i.outSub$ fields, preserving the invariant. If e adds a non-empty set $N = sub(sr)$ to $i.inSub_c$, then by Lemma 5.3, $subscribed_{i,j}(n, t)$ holds for each $n \in N$ after e, preserving the invariant. If e removes a set $N_d = unsub(sr) \cap nHop_i(d)$ from $j.outSub_d$ for a channel $d \in CHANNELS_i$ incident to a node o, then e also adds a cancellation response sr where $unsub(sr) = N_d$ to $i.responses_d$, so $subscribed_{o,i}(n, s)$ holds for each $n \in N_d$, preserving the invariant.

5. If $e =$ receive$(a \in A)_{c,j}$ then e affects no $j.inNames$ or $j.outNames$ fields, preserving the invariant, but removes response $r_a = message(a)$ from $j.responses_c$. If $r_a \notin SR$ then e cannot falsify $subscribed_{i,j}(n, t)$, preserving the invariant. Otherwise, $r_a \in SR$ and a is enqueued by an event $e_r =$ receive$(r_a \in SR)_{c,i}$. By lemma 5.3 (with $e = e_r$ and $e_a =$ e), $subscribed_{i,j}(n, t)$ holds after e, preserving the invariant.

6. If e = purge($N \subseteq$ *NAMES*$)_{c,j}$ then e affects no $j.inNames$ fields and removes no subscription responses from *j.responses* fields, preserving the invariant. It does remove set $N_c = N \cap nHop_i(c)$ of names from $j.outSub_c$, but it also adds subscription response $SR(\emptyset,\ N_c)$ to $j.responses_c$, so $subscribed_{i,j}(n, s)$ holds for each $n \in N_c$.

Since all possible extensions of X' preserve Name Conflict Locality, Name Conflict Locality is preserved for X, completing the induction □

Consider the case when a receive($q \in Q)_{c,j}$ event occurs and $q \notin j.qHop$ but where there is no conflicting cancellation response in $j.responses_c$. We claim that if Name Conflict Locality holds then q cannot have been forwarded to j from a switch. If we suppose for contradiction that q is forwarded from a switch i then $(q,\ c) \in i.qHops$, so $depends(q) \subseteq i.outSub_d$ for the channel $d \in CHANNELS_i$ on which q is received at i. Since by Lemma 5.2 $i.outSub_d \subseteq i.inSub_c$, by Name Conflict Locality we have $subscribed_{(i,j)}(n,\ t)$ for each $n \in depends(q)$. Since $q \notin j.qHop$, there is a name $n \in depends(q)$ such that $n \notin j.outSub_c$ and by Name Conflict Locality $j.responses_c$ holds a cancellation response conflicting with n, which is a contradiction. We state this insight as a corollary.

Corollary 5.2 *At each event e = receive($q \in Q)_{c,j}$ event in any execution $X \in$ execs(DAS) where e is caused by a switch : $q \in j.qHop$.*

From the definition of *qHop* we see that the $qHop_i$ relation dictates which mappings may appear in the $i.qHop$ fields of each switch $i \in SWITCHES$ and hence the forwarding path $fp(q)$ of any request $q \in Q$. However, a forwarding path is not effective unless each switch along actually has an entry for q in its $i.qHop$ field at the time it receives q. Let $fp_i(q)$ denote the suffix of $fp(q)$ beginning with switch $s_i \in fp(q)$. We define the effective part of a dynamic forwarding path as follows:

Definition 5.4 *For any state $t \in$ states(DAS) and any $q \in Q$ let $dfp_i(q,t)$, the dynamic forwarding path of q in t, be the (possibly empty) longest prefix s_i, s_{i+1}, ..., s_m of forwarding path $fp_i(q) = s_i,\ s_{i+1},\ ...,\ s_n$ of switches such that for each $k \in [i, m] : s_k.qHop(q) = (k, k+1)$.*

Note that $dfp_i(q,\ t_{start}) = \lambda$ for all $q \in Q, i \in SWITCHES$ in any start state t_{start} of DAS. Dynamic / Name Conflict Locality yields the following result, saying that if a switch i has a *qHop* entry for a request q in a state t reachable in DAS, then either $dfp_i(q,\ t) = fp_i(q)$ or else some switch in $fp_i(q)$ holds a cancellation response conflicting with q. This captures the fundamental intuition about dynamic atomic subscriptions: if some name n is in $i.inSub_c$ at a switch i for the channel c leading towards $host(n)$, then all the switches along the path are subscribed to n or else the path has been (recently) severed and a cancellation response containing n is traveling back along the path, i.e. the response is buffered on at least one switch on the path.

Definition 5.5 *Let* $hasCR(fp, q, t)$ *be true for a path of switches* $fp = s_1, s_2, ..., s_n$, $q \in Q$ *and* $t \in states(DAS)$ *exactly if in state* t : $\exists\ s_k \in fp$: ($\exists\ cr \in s_k.responses_{(k-1,k)} \cap SR$: $depends(q) \cap names(cr) \neq \emptyset$).

Lemma 5.5 *(dfp transitivity) In each reachable state* $t \in states(DAS)$, *for each* $q \in Q$ *and any switch* $s_i \in fp(q) = s_1,\ s_2,\ ...,\ s_n$: $s_i.qHop(q) = (i, i+1) \Rightarrow dfp_i(q,t) = fp_i(q) \vee hasCR(fp_i, q, t)$.

Proof: Fix any $q \in Q$, switch $s_i \in fp(q)$ and reachable state $t \in states(DAS)$ where $s_i.qHop(q) = (i,\ i{+}1)$. The invariant holds if $hasCR(fp_i, q, t)$ is true, so consider the case when it is false, meaning no switch on $fp_i(q)$ has a cancellation response conflicting with q. We must show in this case that $dfp_i(q,\ t) = fp_i(q)$. We proceed by induction, showing that if $s_k \in dfp_i(q)$ then $s_{k+1} \in dfp_i(q)$, for $k \in [i,\ n]$.

As our base case, $s_n \in dfp_i(q,\ t)$, since $s_n.qHop(q) = (n,\ destination(q))$, by the antecedent. As our inductive hypothesis, assume $s_k \in dfp_i(q,\ t)$ for some $i \leq k < n$. Then $s_k.qHop(q) = (k, k+1)$ and $depends(q) \subseteq s_k.inSub_{(k,k+1)}$, by Lemma 5.1 and the definition of *qHop*. By Lemma 5.4 and Name Conflict Locality, either $depends(q) \subseteq s_{k+1}.outSub_{(k,k+1)}$ or $\exists\ cr \in s_{k+1}.responses_c \cap SR$: $depends(q) \cap names(cr) \neq \emptyset$. Since the latter is false in our case ($k \geq i$), the former holds, that is: $depends(q) \subseteq s_{k+1}.outSub_{(k,k+1)}$. Since $s_{k+1} \in fp(q)$, $qHop_{k+1}(q) = (k{+}1,k{+}2)$ and so $depends(q) \subseteq nHops_{k+1}((k{+}1,k{+}2))$, by the definition of $qHop_{k+1}$. This implies that $s_{k+1}.qHop(q) = (k{+}1,k{+}2)$, by the definition of $s_{k+1}.qHop$, and hence $s_{k+1} \in dfp_i(q)$, completing the induction □

By Lemma 5.5 and Dynamic Conflict Locality, if no switch on forwarding path $\mathit{fp}_i(q)$ holds a cancellation responses conflicting with q, then for any $r \in R$ where *conflicts*(q, r) and any pair of switches i, j on the path: $i.\mathit{qHop}(q) = (i, j) \Rightarrow (r, (j, i)) \in j.\mathit{rHops}$. Since by definition $i.\mathit{qHop}$ and $j.\mathit{rHops}$ contain subsets of qHop_i and rHops_j, respectively, this implies Static Path Conflict Locality for $\mathit{fp}_i(q)$, as per Definition 3.4 on page 36. We have the following corollary:

Corollary 5.3 *(Dynamic Path Conflict Locality) For each reachable state* $t \in \mathit{states}(DAS)$ *and any switch* $s_i \in \mathit{fp}(q) = s_1, s_2, \ldots, s_n$ *for some* $q \in Q$: $s_i.\mathit{qHop}(q) = (i, i+1) \wedge \neg \mathit{hasCR}(\mathit{fp}_i(q), q, t) \Rightarrow$ *Static Path Conflict Locality holds on* $\mathit{fp}_i(q)$ *in* t.

That switches can purge responses may lead one to worry that a conflicts might go undetected. However, switches don't really purge responses but rather subsume them with new subscription cancellation responses, whose *unsub* sets contain the union of any name sets removed. Hence, if an incoming request conflicts with a response in a response set in a state before a purge event e, it will also conflict with a response in that set in the state following e, as we now show.

Lemma 5.6 *(safety of response purges) In any state* $t \in \mathit{states}(DAS)$, *for any switch* $i \in \mathit{SWITCHES}$: *if* $\exists\, q \in Q$ *such that* $\mathit{depends}(q) \subseteq i.\mathit{outSub}_c$ *and* $\exists\, r \in i.\mathit{responses}_c$ *such that* $\mathit{conflicts}(q, r)$ *in* t *for any* $c \in \mathit{CHANNELS}_i$, *then* $\exists\, r' \in i.\mathit{responses}_c$ *such that* $\mathit{conflicts}(q, r')$ *in any state* t' *such that* $(t, \mathit{purge}(N \subseteq \mathit{NAMES})_{c,i}, t') \in \mathit{trans}(\mathit{DynamicSwitch}_i)$.

Proof: Let $r \in R$ be any response in $i.\mathit{responses}_c$ at t such that *conflicts*(q, r), for some $q \in Q$ where $\mathit{depends}(q) \in i.\mathit{outSub}_c$. If r is not a member of the set R_c non-deterministically chosen in the purge action, then the invariant trivially holds with $r' = r$. On the other hand, if $r \in R_c$ then let $N_p = \mathit{effects}(r) \cap N$, that is: those names in the effect set of r that are being purged from the subscription. Since *conflicts*(q, r) then $\mathit{depends}(q)$ and $\mathit{effects}(r)$ have some non-empty set N_{qr} of names in common, by assumption 5.1. Observe that $N_{qr} \subseteq \mathit{depends}(q) \subseteq i.\mathit{outSub}_c$. Since $\mathit{effects}(r) \cap i.\mathit{outSub}'_c = \emptyset$ then $N_{qr} \cap i.\mathit{outSub}'_c = \emptyset$, implying $N_{qr} \subseteq N_p$, that is: all the names that q and r have in common are purged. This also shows that N_p is non-empty.

Since $N_{qr} \subseteq N_p \subseteq N$, $N_{qr} \subseteq$ *unsub*(sr), with $sr = SR(\emptyset,\ \emptyset,\ N)$, the cancellation response that the purge action adds to *i.responses*$_c$. Since *conflicts*$(q,\ sr)$, by the definition of cancellation responses and conflicts, the invariant holds with $r' = sr$ □

Note that cancellation responses are never purged. Practically speaking, it would be straightforward to show that replacing a set of cancellation responses $C \subseteq$ *i.responses*$_c$ with a single cancellation response *cr* such that $\bigcup_{r \in C}$ *effects*$(r) \subseteq$ *effects*(cr) preserves Name Conflict Locality.

We can now show the main result of this Section, a Path Atomic Transfer theorem for dynamic atomic system DAS. It says that Path Atomic Transfer (Theorem 3.1) holds in the dynamic system. Let $fp_{ab} = s_1,\ s_2,\ ...,\ s_n$, for some nodes a, $b \in NODES$ and let s_k denote the k-th switch on fp_{ab}. Our proof is structured in a similar way as the proof for Theorem 1.

Theorem 5.1 *(Dynamic Path Atomic Transfer): For all* $X \in$ *execs*(DAS), *all* $i, j \in [1, n]$ *where* $i \leq j$, *and all* $r \in R$, $q \in Q$ *with forwarding path fp*$(q) =$ *fp*$_{ab}$*:* *receive*$(r)_{(j,j+1),j} <_X$ *receive*$(q)_{(i-1,i),i} <_X$ *receive*$(ack(r))_{(i-1,i),i} \wedge$ *conflicts(q, r)* $\Rightarrow$ *receive*$(q)_{(i-1,i),i} \not\rightarrow_X$ *send*$(q)_{(j,j+1),j}$.

Proof: Suppose for contradiction that there exists some $X \in$ *execs*(DAS) and some $r \in R$, $q \in Q$ where *conflicts*(q, r) such that receive$(r)_{(j,j+1),j} <_X$ receive$(q)_{(i-1,i),i} <_X$ receive$(ack(r))_{(i-1,i),i}$ and receive$(q)_{(i-1,i),i} \rightarrow_X$ send$(q)_{(j,j+1),j}$, where $i \leq j$. Let I be the interval of X beginning after receive$(r)_{(j,j+1),j}$ and ending after receive$(ack(r))_{(i-1,i),i}$. Observe that since receive$(q)_{(i-1,i),i} \rightarrow_X$ send$(q)_{(j,j+1),j}$, there is no drop(q) event in I. We define *q-switch*$(q,\ E)$ and *r-switch*$(r,\ E)$ the same way as in the proof of Theorem 3.1 in Chapter 3 on page 36.

We separate the cases when Static Path Conflict Locality holds and when it doesn't hold on the part of the forwarding path where cancellation responses can affect the forwarding of q and r. For any finite execution or trace E let *fp*$_{qr}(E)$ denote the *relevant path of* q *and* r *after* E*,* the interval $s_{k+1}, ..., s_m$ of *fp*$_i$*(q)* where $k =$ *q-switch*$(q,\ E)$ and $m =$ *r-switch*$(r,\ E)$. In other words, the path starts with the next switch after the one where q was last received in E and ends at the switch where r was last received in E*,* as illustrated in figure 5.3 a). Let $PCL(I)$ be the predicate

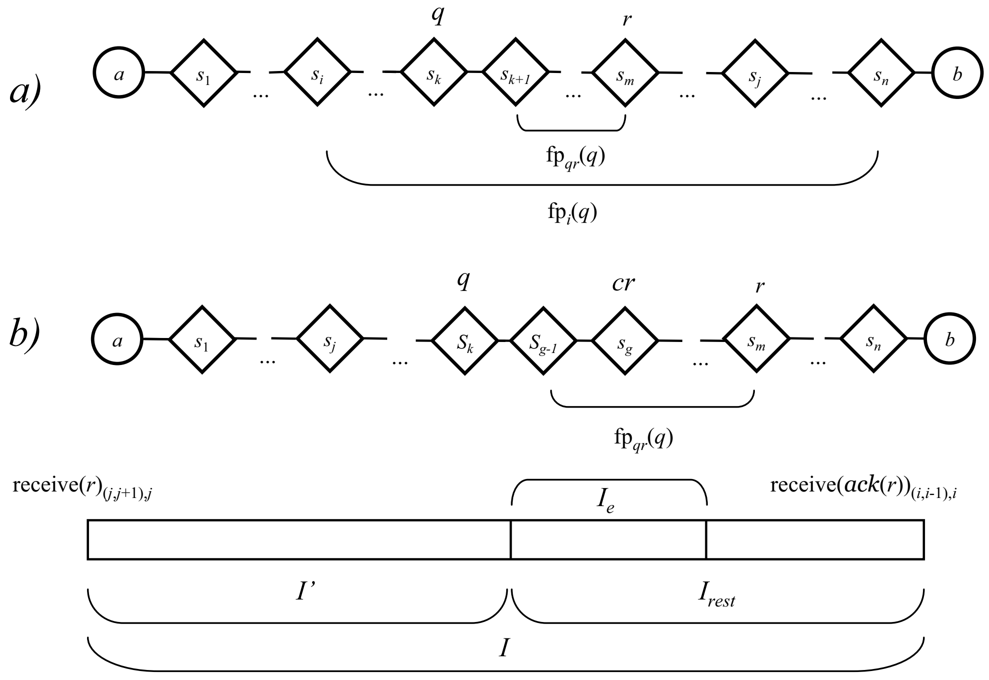


Figure 5.3: Case 2 of Theorem 5.1 illustrated

that is true exactly if for each prefix I' of I : $\neg hasCR(fp_{qr}(I'), q, t_{I'})$, where $t_{I'}$ is the last state of I'. In other words, PCL is true for an execution interval I exactly if $\neg hasCR$ holds for the relevant path of q and r in each state of I, so at no point in I are there any cancellation responses conflicting with q on the path between q and r.

1. If PCL is true, then by Corollary 5.3 Static Path Conflict Locality holds on $fp_{qr}(I')$ in the final state $t_{I'}$ of every prefix I' *of I*, so $s_k.qHop(q) = qHop_k(q)$ and $s_k.rHops(r) = rHops_k(r)$ at every switch $s_k \in fp_i(q)$ at its receive event for q and r. By comparing the receive$(q \in Q\setminus \text{SQ} \setminus \text{VQ})_{c,i}$, receive$(r \in R\setminus SR \setminus \text{VR})_{c,i}$ and receive$(a \in A)_{c,i}$ actions of DynamicSwitch$_i$ to the corresponding actions in AtomicSwitch$_i$, we see that each pair of actions behaves the exactly the same in this case.

   There may be a purge$(N \subseteq NAMES)_{c,g}$ event at some switch $s_g \in fp_{qr}(I')$ during I, but it doesn't purge r since by the definition of *qHops, depends*(q) $\subseteq s_g.outSub_{(g-1,g)}$ in the last state of I', and so by Lemma 5.6 the resulting cancellation response would conflict with q, contradicting $\neg hasCR$.

Response r is therefore not discarded but forwarded in the opposite direction to q along $fp_i(q)$ and by the same reasoning as for Theorem 3.1, there must be some prefix I_e of I with a last event e after which $q\text{-}switch(q, I_e) = r\text{-}switch(r, I_e) = k$, for some $k \in [i, j]$. Event e must be one of receive$(q)_{(k-1,k),k}$ or receive$(r)_{(k,k+1),k}$, so repeating the subsequent argument of Theorem 3.1, receive$(q)_{(k-1,k),k} \not\rightarrow_I$ send$(q)_{(k,k+1),k}$, a contradiction.

1. If PCL is false then there exists some prefix I' of I such that in the last state $t_{I'}$ of I' : $\exists\ s_g \in fpqr(I')$: $\exists\ cr \in s_g.responses_{(g-1,g)} \cap SR$: $depends(q) \cap names(cr) \neq \emptyset$, for some g where $q\text{-}switch(q,\ I') < g \leq r\text{-}switch(r,\ I')$. Let I' be the shortest such prefix. Note that other cancellation responses may get added to path $fpqr$ during the remainder of I due to non-deterministic purge events, but observe that purge events remove neither requests nor cancellation responses. For any finite execution fragment E, let $cr\text{-}switch(E)$ denote the index g of the first switch on $fp_{qr}(E)$ that has held a cancellation response $cr \in SR$ in E where $conflicts(q, cr)$, that is: the lowest g, $q\text{-}switch(q, E) < g \leq r\text{-}switch(r, E)$, such that receive$(cr)_{(g,g+1),g} \in E$ with $conflicts(q, cr)$ or purge$(N \subseteq NAMES)_{(g-1,g),g} \in E$ with $depends(q) \cap N \neq \emptyset$.

   Recall that $q\text{-}switch$ only increases in steps of 1. Observe that each receive$(cr \in SR)_{(g,g+1),g}$ event similarly decreases $cr\text{-}switch$ by at most 1, since each such event is caused by a prior send$(cr\ \in\ SR)_{(g,g+1),g+1}$ event. A purge$(N \subseteq NAMES)_{(g-1,g),g}$ can decrease $cr\text{-}switch$ directly to g, but then $g > q\text{-}switch$ by the definition fp_{qr}, so purge events can only move $cr\text{-}switch$ closer to $q\text{-}switch$ without reaching it or going below it. In figure 5.3 b), for example, either a receive$(cr)_{(g-1,g),g-1}$ event or a purge$(N \subseteq depends(q))_{(g-2,g-1),g-1}$ event could change $cr\text{-}switch$ from g to g-1, but a purge$(N \subseteq depends(q))_{(j-1,j),j}$ event would not change $cr\text{-}switch$, since $j \notin fpqr$ at that point.

   Let I_{rem} denote the suffix of I starting with the last state of I', as illustrated in figure 5.3 b). Let i_q and i_{cr} denote $q\text{-}switch(q,\ I')$ and $cr\text{-}switch(I')$, respectively. We have $i_q < i_{cr} \leq j$. Since $q\text{-}switch(q, I') = i_q$ but $q\text{-}switch(q, I) = j$, $q\text{-}switch$ takes on every value in $[i_q,\ j]$ during I_{rem} . Since $cr\text{-}switch(I') \in [cr\text{-}switch(I), i_{cr}]$ in every state of I_{rem}, where $cr\text{-}switch(I) \geq i_{cr}$, there must be some prefix I_e of I_{rem} with last event e after which $q\text{-}switch(q, I_e) = cr\text{-}switch(I_e) = k$, for some $k \in [i_q,\ j]$. Event e must be one of receive$(q)_{(k-1,k),k}$ and receive$(cr)_{(k,k+1),k}$, so repeating the subsequent argument of Theorem 3.1, receive$(q)_{(k-1,k),k} \not\to_I$ send$(q)_{(k,k+1),k}$, which is a contradiction □

5.8 Dynamic End-To-End Atomicity

We now show that an application's requests are executed on a server only if they do not conflict with other concurrently issued requests. Theorem 5.2 is stated in terms of the network of switches and channels, with no reference to the end-host attached to it. For End-to-End Atomicity to hold, requesters must ACK Well-Formed and responders must be Responder Well-Formed, as in Section 1.

The theorem can now be stated, and easily proven since Theorem 5.1 provides exactly the guarantees needed to reprise the proof of Theorem 3.2.

Theorem 5.2 *(End-To-End Atomicity): For all $X \in$ execs(DAS) where X is well-formed for hosts a, $b \in$ HOSTS, all $r \in R$ enqueued by an event e_r at b and all $q \in Q$: $e_r <_X$ receive$(q)_{(n,b),b} \wedge$ send$(q)_{(a,1),a} <_X$ send$(ack(r))_{(a,1),a} \wedge$ conflicts(q, r) $\Rightarrow$ send$(q)_{(a,1),a} \not\to_X$ send$(r_q)_{(n,b),b}$, for any $r_q \in R$.*

Proof: Since Theorem 5.1 provides identical guarantees as Theorem 3.1, the proof of Theorem 3.2 for *AS* applies □

5.9 Liveness

Recall from Corollary 5.2 that the case $q \notin j.qHop$ at a receive$(q)_{(o,j),j}$ event in a switch j cannot occur if o is a switch. However, this case can occur if o is a host in a state where $depends(q) \nsubseteq o.inSub_{(o,j)}$. But this means that host o is issuing a request without being subscribed to its entire dependency set. We consider such hosts to be in error and rule them out by placing a technical well-formedness condition on the environment of *DAS*, roughly corresponding to the Subscription Well-Formedness of Chapter 3 (Definition 3.8 on page 43).

Definition 5.6 *An execution $X \in$ execs(DAS) is Dynamic Subscription Well-Formed for a host $a \in$ NODES with home switch i if at each event $e =$ receive$(q \in Q)_{(a,i),i}$ in X, subscribed$_a(n, t)$ holds for each $n \in$ depends(q) in the state t preceding e.*

As before, the reason for this condition is to exclude cases where *qHop* is undefined for a request, which is the same property we ensured for Theorem 3.3 of Chapter 5[11]. In fact, the theorem is very similar to that Theorem.

Theorem 5.3 *(Dynamic Atomic Transfer Liveness) For all $X \in$ fairexecs(DAS), where X is Dynamic Subscription Well-Formed for a host $a \in$ HOSTS, for each event $e =$ send$(q \in Q)_{(a,i),i}$ such that no message caused by e is detected as a conflict in X and q is not dropped by a drop(q) event, there is a receive$(q \in Q)_{(b,i),b}$ event in X, where $b =$destination(q).*

Proof: Let let X' denote the suffix of X whose first event is e. Let $fp(q) = s_1, s_2, \ldots, s_n$ be the forwarding path of q, and let I be the shortest suffix of X' containing every event caused by e. For any finite execution or trace E let $fp_{qb}(E)$ denote the *relevant path of q after E,* the interval $s_{k+1}, ..., s_m$ of $fp(q)$ where $k = q\text{-}switch(q, E)$. Let $PCL(I)$ be the predicate that is true exactly if for each prefix I' of I : $\neg hasCR(fp_{qb}(I'), q, t_{I'})$, where $t_{I'}$ is the last state of I'. Since q is never detected as a conflict, $PCL(I)$ must hold. Hence, Static Conflict Locality holds on fp_{qb} throughout I. Therefore, Lemma 3.6 still holds and every message enqueued on each switch $s_k \in fp_{qb}(I)$ is eventually sent. By Lemma 3.5, every message enqueued on a channel is eventually received. Furthermore, since a is Dynamic Subscription Well-Formed, $qHop_i(q)$ is defined. Since by assumption q is not dropped by a drop(q) event, the argument for Theorem 3.3 still applies, and every receive$(q)_{(k-1,k),k}$ event caused by e adds q to $s_k.outQueue_{k+1}$, for $1 \leq k < n$. Therefore, there must be a receive$(q \in Q)_{(b,i),b}$ event in I $\Box$

This is a fairly weak liveness condition, essentially saying that when no conflicts or network problems occur, requests will complete. Any stronger guarantee must make some assumptions about the occurrences of drop(q) failure events. Time-out and retransmission schemes can be added for request reliability and starvation due to

[11] Another approach that would suffice for safety would be to let receive$(q \in Q)$ drop a request that cannot be forwarded, as a switch implementation would likely do anyway.

repeated conflicts alleviated by giving a preference to re-issued requests, as mentioned at the beginning of this chapter. While liveness and fairness issues can be treated orthogonally to Atomic Transfer, it could be worth investigating whether conflict checking and cancellation responses can contribute to efficient algorithms for tackling these issues.

5.10 Discussion and Related Work

This section discusses the relationship between Atomic Transfer and reliable multicast, and work related to reliable multicast.

5.10.1 Atomic Transfer and Reliable Multicast

Reliable multicast is a fundamental part of our approach. Requests "swim upstream" towards the root of the multicast trees of destination hosts, and are dropped by any conflicting responses they encounter along the way.

The choice of underlying multicast routing protocol is mostly orthogonal to Atomic Transfer. Our multicast model is closest to the *source-specific multicast* or core-based tree models [138, 139, 142], since the host of a set of names is the root of the multicast tree for those names. Holbrook and Cheriton [139] argue that this model is simpler to implement than multi-source models [143, 144] oriented towards group communication, since it can reuse the existing unicast (e.g. IP) routing infrastructure directly. They argue that one-to-many multicast suffices for most applications, especially the large-scale ones that might benefit most from it, such as Internet broadcasting and content distribution. Our design would seem to agree with that viewpoint.

Atomic Transfer requires reliable multicast, but scalable and reliable multicast is non-trivial and still an active area of research [145, 144, 146, 147, 84]. Methods for reliable transmission and flow control of unicast (point-to-point) links do not easily transfer to multicast settings, as different receivers may receive different subsets of messages and have wildly different network latencies, bandwidth and congestion conditions. Treating n multicast receivers as n instances of reliable point-to-point connections does not scale well, as the source host must track membership and reception progress for all receivers. Also, receiver ACKs lead to congestion as they "implode" back onto the source host. This problem can be somewhat alleviated by having receivers send only negative acknowledgements or *repair requests* for messages they fail to receive, but ultimately a protocol that relies on the source host for all retransmission repairs for missing data cannot scale. This has prompted research on receiver-based reliable multicast [148, 144, 147, 149, 84], where one or more nodes in the system log messages and receivers are responsible for obtaining missing messages from such nodes. Such protocols must consider various issues, such as minimizing the number of redundant repair requests and responses, localizing repair message traffic to the area of network affected by a message loss and minimizing repair latency, for example. We discuss these aspects somewhat further in Section 7.5.3.

Atomic Transfer places an important restriction on it's multicast protocol's reliability mechanism, namely that (Dynamic) Conflict Locality be preserved at all time. The approach we take in this chapter, of severing subscriptions upon discarding responses, may be needlessly drastic in many cases. It might also lead to liveness problems, as sporadic drops of a few responses could force a large number of hosts to re-subscribe to names and, in the systems of Chapters 6 and 7, re-synchronize the affected part of their cached state.

As a part of integrating AT with a reliable multicast protocol, we plan to investigate the separation of cancellation responses from cancellation of subscriptions, enabling a switch to discard responses and send a "drop response" without severing subscriptions. This would preserve Dynamic Conflict Locality, but give a receiver the chance to repair its response stream by requesting the missing responses from other nodes in the system, as in receiver-based reliable multicast system.

One might worry that sending additional messages in response to congestion could make the problem worse. However, as noted earlier, a drop response message can be made smaller than the combination of dropped responses it represents. For example, during a transient overflow at a switch, the initial drop response would contain the exact set of names from the first response dropped. As more dropped responses accrue, their names can be merged into existing drop responses. In the prototype scheme of Chapter 8, this happens in the normal case anyway, as a part of maintaining *responses* sets, so no new mechanism is required. In addition, drop responses can be coarsened in that scheme by replacing groups of names with a prefix they share, shrinking drop responses at the cost of overstating the name set affected. If the congestion is transient, the drop response is eventually sent, but if the congestion is durable (due to a link being a persistent bottleneck link, for example) the switch would eventually purge some of the names from its own subscription and change the corresponding drop responses into cancellation responses.

As mentioned earlier, our model assumes static routes, with no route changes during execution. This assumption must be changed for practical implementation. One way a route change can occur is when a switch decides to attach to a different parent switch in the multicast tree. This corresponds to the entries for a set of names N in a dynamic *nHops* field changing from one channel to another. A straightforward way to do this would be for the switch to unsubscribe from one channel and then, upon receiving its cancellation response (and forwarding it on, possibly as a drop response), subscribe to the new channel. Clearly this preserves Conflict Locality, but at a cost to downstream receivers. A less disruptive way would be for the switch to first subscribe to the new channel and only unsubscribe from the old one upon receiving the first duplicate message from the new one[12], allowing a seamless route change while preserving Conflict Locality. So while Conflict Locality restricts how route changes may occur, the restrictions do not appear too onerous.

The last major assumption we make with respect to multicast is that subscriptions can be made on a very fine-grained level, namely on a name-by-name basis. This makes our models simple, but may present implementation challenges. On the other hand, if direct name-based forwarding remains infeasible or inferior to address-based forwarding, it is relatively simple to separate name resolution from forwarding; a requester can pre-determine the network address(es) of the server(s)[13] for a set of names and then join the multicast tree(s) rooted at the address(es).

Handling multicast subscription issues for fine-grained names may be a thornier issue. Although modern routers are equipped with considerable memory, keeping track of potentially billions of subscriptions is a challenge. Any solution will likely make use

[12]These issues have been investigated in the context of mobile networking [150]

[13]Or the network address of the gateway to the host's data center.

of names that are at least partly hierarchically structured, such as is the case with IP addresses and the prototype naming scheme of Chapter 8. For example, a data name n could be structured as a pair $(auth, id)$, where $auth$ is the cryptographic digest [151] of the public key [152] claiming authority over n [153]. The id part would be a variable-length bit string with a hierarchical structure, that is: the longer the prefix shared by two names in the $auth$ "name space", the shorter the expected network distance between their hosts. Subscription processing could take advantage of this structure, by recording subscriptions to prefixes instead of individual names as much as possible. The $auth$ part of n coupled with some prefix of id would suffice to map n to its current host. Core switches might maintain relatively coarse-grained subscriptions while switches closer to senders and receivers would retain more fine-grained subscriptions, limiting the flow of superfluous messages onto network edge links.

In sunmmary, several obstacles to efficient realization of name-based forwarding remain. Indeed the multicast subscription problem might limit initial deployment of Atomic Networks to data centers, where name spaces and (virtual) network topologies can be carefully controlled. On the other hand, if visions of global, virtual *Metanetworks* [154, 118] come to fruition, operators of large-scale distributed applications might configure relatively stable dedicated multicasting trees for their applications, making some of these problems more manageable.

Chapter 6

Dynamic Atomic Cache and State Transfer

This chapter augments the Atomic Cache system of Chapter 4 to work with dynamic subscriptions, based on the dynamic atomic network described in Chapter 5. In the simplified, static system of Chapter 4, application caches initially contain server start states. They stay synchronized by applying each subsequent server response to their cached states. But for a dynamic application to subscribe to a server at a later point in an execution, it must either obtain the complete sequence of server responses up to that point or else obtain a current copy of the server's state.

Server state is partitioned using names, as defined in Chapter 5, permitting caching and transfer of subsets of a server's state. This Chapter presents a straightforward cache synchronization protocol where state is transferred directly from servers to applications. Chapter 7 refines the algorithm to allow applications to synchronize their caches using other application caches. That protocol is designed to ensure liveness despite high response generation rates.

6.1 States, Names, Values, Updates and Conflicts

We refine the definition of $STATES_b$, the state space of server $b \in DATAS$ that is completely abstract in Chapter 4. Let $STATES_b$ now be some set of finite, partial functions from names of b to $VALUES$, so each state $s \in STATES_b$ is a function yielding the value of each name in that state, if defined.

We introduce operators for updating the values of names in states. Let *UPDATES* be the set of *state updates*, each element of which is a finite partial function mapping *NAMES* to $\mathit{VALUES} \cup \{\perp\}$, where $\perp \notin \mathit{VALUES}$. For any $u \in \mathit{UPDATES}$. let $\mathit{names}(u)$ denote the (finite) domain of u. We will sometimes interpret u as a set of elements $(n \mapsto v)$, each mapping a name n to some $v \in \mathit{VALUES} \cup \{\perp\}$.

An update is used to transform one state of a server $b \in \mathit{DATAS}$ into another. We define the *function override operator* $\oplus : \mathit{STATES}_b \times \mathit{UPDATES} \rightarrow \mathit{STATES}_b$, for any $s \in \mathit{STATES}_b$, $u \in \mathit{UPDATES}$, $n \in \mathit{NAMES}_b$, as follows:

- If $u(n)$ is undefined then $(s \oplus u)(n) = s(n)$, or is undefined if $s(n)$ is undefined.
- If $u(n) = \perp$ then $(s \oplus u)(n)$ is undefined.
- If $u(n) = v$ for some $v \in \mathit{VALUES}$, then $(s \oplus u)(n) = v$.

Clearly, for all s_1, $s_2 \in \mathit{STATES}_b$, there exists an update $u \in \mathit{UPDATES}$ such that $s_1 \oplus u = s_2$.

We define the projection operator $|$: $\mathit{UPDATES} \times \mathit{NAMES} \rightarrow \mathit{UPDATES}$, as the operator that for any update u and name set N yields the maximal subset of u which updates only names in N, that is: $\{\ (n \mapsto v) \in u \mid n \in N\ \}$.

We refine the definition of *depends* and *effects* functions for requests and responses, in a way that matches the definition of the *conflicts-inv* conflict function of Chapter 4. We restate the definition here below.

$\mathit{conflicts\text{-}inv} = \{\ (q \in Q,\ r \in R) \mid \exists s \in \mathit{STATES}_b :$
$\mathit{execute}_b(s, q) \not\subseteq \mathit{execute}_b(\mathit{TRANS}_b(s, r), q)\ \}$, where $b = \mathit{destination}(q)$.

We define $\mathit{depends}(q)$ as the set of names whose change of value may expand the set of possible responses to q. Intuitively, it has names whose value changes could lead to a new response not expected by a requester given the responder's cached state. Formally:

Definition 6.1 $\mathit{depends}(q) = \{\ n \in \mathit{NAMES}_b \mid \exists\ s \in \mathit{STATES}_b,$
$v \in \mathit{VALUES} \cup \{\perp\} : \mathit{execute}_b(s, q) \not\subseteq \mathit{execute}_b(s \oplus \{(n \mapsto v)\}, q)\ \}$

If *execute*$_b$ is deterministic then the predicate simplifies to: *execute*$_b(s, q) \neq$ *execute*$_b(s \oplus \{(n \mapsto v)\}, q)$.

We define *effects*(r) as the subset of names of server $b =$ *sender*(r) that may change as a result of r being applied. Formally:

Definition 6.2 *effects*$(r) = \{\, n \in NAMES_b \mid \exists\, s \in STATES_b\text{: } s(n) \neq TRANS_b(s, r)(n) \,\}$.

Recall assumption 5.1 of Chapter 5, that conflicting requests and responses have a name in common, for the safety of dynamic subscriptions. Our definitions lead to this assumption being satisfied, as shown in the following Lemma:

Lemma 6.1 *(Names and Conflicts): For any $q \in Q$, $r \in R$:*
conflicts-ing$(q,\ r) \Rightarrow$ *effects*$(r) \cap$ *depends*$(q) \neq \emptyset$.

Proof: Let $b =$ *destination*(q). If $(q,\ r) \in$ *conflicts-ing* then there exists a state $s \in STATES_b$ such that *execute*$_b(s, q) \not\subseteq$ *execute*$_b(TRANS_b(s, r), q)$. Let N denote the set $\{\ n \in NAMES_b \mid s(n) \neq s'(n)\ \}$ where $s' = TRANS_b(s,r)$, that is: the set of names whose values differ in s and s'. By the definitions of response effect sets, $N \subseteq$ *effects*(r). Since *depends*(q) contains by definition all names whose value change may expand the set of possible responses, there is some $n \in N$ such that $n \in$ *depends*(q), so $n \in$ (*effects*$(r) \cap$ *depends*(q)) and *effects*$(r) \cap$ *depends*$(q) \neq \emptyset$ $\Box$

6.2 Responses and partial states

Recall from Chapter 4 that responses are not direct carriers of new data for caches. Rather, they carry back the result of a request's execution, whose effect the application recreates on its cached version of the state using the $TRANS_b$ function. $TRANS_b$ can range from a simple write mapping that assigns constant values to names to a complex, state-dependent computation that transforms the state in arbitrary ways. The fact that dynamic cache application hosts may cache only a subset of a server's state creates a complication for this scheme. Clearly, a proper subset s' of a state $s \in STATES_b$ is not the same as s, and may even be a non-member of $STATES_b$.

Whether this matters ultimately depends on TRANS_b. A simple value overwrite function, for example, behaves the same way regardless of its input state. But in general, a response in the dynamic cache system effectively has a dependency set: the set of names whose values determine the response's effect. We could easily define a dependency set for each response $r \in R$ along the lines of:

$\mathit{depends}(r) = \{\ n \in \mathit{NAMES}_b \mid \exists\ s \in \mathit{STATES}_b,\ v \in \mathit{VALUES} \cup \{\bot\} :$
$\mathit{TRANS}_b(s, r) \neq \mathit{TRANS}_b(s \oplus \{(n \mapsto v)\}, r)\ \}$

However, the design and structure of responses and their dependency sets requires careful consideration, since an application must be subscribed to and have an up-to-date copy of a response's dependency set to be able to apply it. This could lead to problems, if applications were repeatedly forced to re-synchronize their caches after receiving a response they were unable to apply. This difficulty can be somewhat mitigated if responses have an internal structure such that subsets of a response can be individually applied. In fact, this section will take that approach to its limit and require that each response is *separable*, meaning that it can be applied individually to each name in its effect set. This ensures that a response can always be correctly applied to any subset of a state's names. While a simplification and a restriction, this allows us to focus on the caching and state transfer protocol without worrying about response semantics. Formally:

Definition 6.3 *A response r is separable if for all $s \in \mathit{STATES}$, $r \in R$ where $b = \mathit{sender}(r)$ and state $s_b \in \mathit{STATES}_b$ and $s \mid N = s_b \mid N$: $\mathit{TRANS}_b(s_b, r) \mid N = \mathit{TRANS}_b(s, r) \mid N$, for any $N \subseteq \mathit{NAMES}_b$.*

This assumption does limit the expressiveness of responses and TRANS_b functions and we plan to better investigate these issues in the future, as discussed in Section 6.9.1.

6.3 Names and Caches

Once an application a has subscribed to a set of names N_a via subscription requests and responses as described in Chapter 5, it initiates state transfer(s) from the server host(s) of N_a. The server(s) send back the data that a needs to *synchronize* its cache with their state, that is: set as the value cached for each name $n \in N_a$ the current value of n on $host(n)$. Once an application has synchronized its cache, it keeps it in synch by applying responses to it, as before.

Here we use the term *synchronized* loosely, as servers generally execute requests and send responses concurrently with state transfers. An application host must carefully combine state transfer messages and any concurrent response messages it receives to obtain a correct, up-to-date state. Indeed, a state transfer from a server b to an application host a may never "complete", in the sense of reaching a state where $a.state_b$ $= b.state$, particularly if b is generating responses rapidly. Still, host a must somehow know when it becomes "synchronized enough" with b to safely issue requests. This is the task of a *cache synchronization protocol*, to get an application host synchronized with a name set so that it can safely issue requests depending on the names.

Recall the definition of value read requests and response messages, from Chapter 5. We let each value message $vr \in VR$ correspond to an update $update(vr) \in UPDATES$, and let $VR(u)$ denote the unique value message vr such that $update(vr) = u$. Each read request $vq \in VQ$ is associated with a set of names $names(vr) \subseteq NAMES$, and we let $VQ(N)$ denote the unique read request vq such that $names(vq) = N$. For notational clarity, if a value message vr is used in a context where an update is required, as for example in the expression $s \oplus vr$, then vr is taken to mean $update(vr)$. Similarly, if a read request vq is used in a name set context then vq is taken to mean $names(vq)$.

For an application a to obtain the latest values for some set of names $N_a \subseteq NAMES_b$ from a server b, it must receive a set $V_a \subseteq VR$ of value messages from b that together provide values for each name in N_a. Application a can request the names explicitly using a value request message, but we leave open the possibility for b to proactively generate value messages. For example, it could multicast them periodically, for parallel synchronization of multiple application caches.

Host a can receive and apply the messages of V_a in any order. Furthermore, we will show that our state transfer protocol never delivers stale value messages, so as soon as a has received values for each name in some set $N \subseteq NAMES$ that a is subscribed to, a has a consistent view of N and can issue any request q such that $depends(q) \subseteq N$, even if it has yet to receive other names from N_a. We note that some concurrency control schemes only guarantee that transactions which commit see a consistent state. Guaranteeing that all transactions see consistent state, as in our model, allows software developers to assume that state invariants hold, rather than having to check them explicitly.

6.4 Direct Cache Synchronization

The cache synchronization protocol in this chapter is relatively simple, transferring state directly from the server where it is hosted. Just after an application host a has completed subscribing to a name set N, each name of N is marked in its cache as being *unsynched*. Upon receiving a value message vr, a applies the message's update to its cache and marks $\mathit{names}(vr)$ as being *synched*. Requests depending on $\mathit{names}(vr)$ may be safely issued at that point. An optimized protocol could combine subscription requests and value requests in the same message, to establish subscriptions and initiate state transfer within the same network round-trip. The message could "detach" at the first switch that is subscribed to the names, such that only the value request is forwarded onwards. We leave out this optimization, for simplicity.

The protocol requires that response messages and value messages are delivered in the order sent, which holds in our model since all responses in R are delivered in the order sent, if at all. We make the simplifying assumption that switches forward a value message to all subscribers of a name in the message, irrespective of whether that subscriber is currently in the process of synchronizing with the name or not. A refinement of the protocol could restrict the flow of value messages to those parts of the network where they are currently needed. Alternatively, value messages could be unicast to their receivers. We do not explore these options here.

Also note that we do not consider liveness for state transfers in the general case, that is: ensuring that applications eventually succeed in synchronizing their caches despite dropped requests and responses. While these are important issues in practice, the approaches to solving them are well understood, such as re-transmission of messages after a time-out, e.g. At any rate, strong liveness guarantees require strong assumptions about the failures that may occur and their detection [155, 156].

6.5 DynamicAppHost$_i$ and DynamicServerHost$_i$

We define I/O Automata for dynamic application hosts and dynamic data server hosts below. We then show that composing these with dynamic atomic switches and channels results in a system where invalid requests are never executed, similar to Theorem 4.1 for the Atomic Cache (CS) system.

DynamicAppHost uses the single field *cache* to store its cached state, instead of a separate $state_b$ field per server as in *SimpleAppHost*. We define the *global set of states STATES* as the set of all finite, partial functions from the global name domain *NAMES* to value domain *VALUES*. We define the projection operator $| : STATES \times NAMES \rightarrow STATES$, as the operator that for any global state $s \in STATES$ and name set $N \subseteq NAMES$ yields the restriction of the state to the names, that is: $\{(n \mapsto v) \in s \mid n \in N\}$.

We emphasize again that while *STATES* can represent any valid state of any server, not every state in *STATES* corresponds to a valid state for every server. The projection $s_{ab} = a.cache \mid NAMES_b$ of the cached state at an application a to the names of a server b is generally *not* an actual state of b: only the subset of s_{ab} corresponding to the synched names of a is guaranteed to be equal to the same subset in a (recent) state of b, as we will show. But by the definition of *depends*, this suffices to correctly execute any request whose dependency set is synched, since other names do not affect the request's execution. Furthermore, if a has an up-to-date version of some subset N of the *effects*(r) name set of a response r from b, response separability ensures that the effects of r on names N in the cache of a will be the same as the effect of r on names N in $b.state$ in the state where b created r.

$DynamicAppHost_a$

A refinement of SimpleAppHost$_a$ that dynamically adds or drops names from its subscription set during execution.

We overload operator $\oplus$ to $STATES \times STATES_b \rightarrow STATES$, to yield for any global state s and server state s_b the state s updated with s_b, that is: $(s \oplus s_b)(n) = s_b(n)$ if $n \in NAMES_b$ or $s(n)$ otherwise.

State:

cache: the application's cached state, an element of *STATES*, initially the empty map.
outQueue: queue of outbound messages, initially empty.
inNames: A total function $NAMES \rightarrow SUBSTATUS$ storing the name statuses that
  a caches for $i.outNames$, where i is the home switch of a.
  Initially *unsub* for all $n \in NAMES$.
synched: the subset of names of *inSub* (see below) that is synched at a, initially $\emptyset$.

Derived State:

$inSub \;/\; inPend = \{\ n \in NAMES \mid inNames(n) = sub \;/\; inNames(n) = pend\ \}$.

Input actions:

receive$(r \in R \setminus SR \setminus VR)_{c,a}$
Effect:
  $cache' = cache \oplus TRANS_b(cache, r)$, where $b = sender(r)$
  $outQueue' = outQueue \cdot ack(r)$

receive$(vr \in VR)_{c,a}$
Effect:
  $cache' = cache \oplus (vr \mid (synched' \setminus synched))$
  $synched' = synched \cup (\text{names}(vr) \cap inSub)$

receive$(sr \in SR)_{c,a}$
Effect:
  $outQueue' = outQueue \cdot ack(sr)$
  $\forall\ n \in NAMES$: $inNames'(n) =$
    $inNames(n)$, if $n \notin \text{names}(sr)$,

$\qquad\qquad$ *sub*, if $n \in sub(sr)$,

$\qquad\qquad$ *pend*, if $n \in pend(sr)$,

$\qquad\qquad$ *unsub*, if $n \in unsub(sr)$.

$\qquad synched' = synched \setminus \text{unsub}(sr)$

Internal actions:

createRequest$(q \in Q)_a$

Precondition:

$\qquad \exists\, r \in R : \; (cache \mid NAMES_b,\; q, r) \in execute_b,$

$\qquad\qquad$ where $b = destination(q)$ and $depends(q) \subseteq synched$

Effect:

$\qquad outQueue' = outQueue \cdot q$

subscribe$(N \subseteq NAMES)_a$

Precondition:

$\qquad N \cap (inSub \cup inPend) = \emptyset$

Effect:

$\qquad outQueue' = outQueue \cdot SQ(N, \emptyset)$

unsubscribe$(N \subseteq NAMES)_a$

Precondition:

$\qquad N \subseteq (inSub \cup inPend)$

Effect:

$\qquad outQueue' = outQueue \cdot SQ(\emptyset, N)$

Output actions:

send$(m \in M)_{c,a}$

Precondition:

$\qquad head(outQueue) = m$

Effect:

$\qquad outQueue' = tail(outQueue)$

The receive($r \in R \setminus SR \setminus VR)_{c,a}$ action receives a request response and updates the cached values of names affected by the response, using the appropriate server's state transition function. It also enqueues back an ACK for the response. The action is essentially the same as in *SimpleAppHost*. Note that we could technically add the names updated to *synched* here, but that is incompatible with the scalable refinement developed in Chapter 7. Also note that for simplicity we model the cache as storing synched as well as old, unsynched values. An implementation, however, might or might not store old unsynched values, depending on whether it has uses for them or not.

The receive($vr \in VR)_{c,a}$ action processes a value message by applying its update to the cache. It also notes that the message's names are now synched. More specifically, it notes exactly those names it is subscribed to, to preserve the $\mathit{synched} \subseteq \mathit{sub}$ invariant. The fact that the action ignores any non-subscribed names in value messages is important for correctness. Note that the action only updates the values of names that were not synched before.

The receive($sr \in SR)_{c,a}$ action receives a notification of changes to the application's subscription, either due to an earlier subscription request it sent itself or due to subscription changes initiated in the network. Note that the action adds and/or removes names from *inSub* in this action, not the subscribe/unsubscribe actions. The action enqueues back and ACK for the message. It also removes any unsubscribed names from *synched*, to maintain the $\mathit{synched} \subseteq \mathit{inSub}$ invariant.

The createRequest($q \in Q)_{c,a}$ action is very similar to the synonymous action in *SimpleAppHost*, non-deterministically issuing a new request. The crucial difference is the additional precondition that a request only be issued if its dependency set is synched.

The subscribe($N \subseteq \mathit{NAMES})_a$ action non-deterministically decides to subscribe to some set of names, issuing a subscription request to the home switch. As mentioned here above, no names are added to *inSub* by this action.

Similarly, the unsubscribe($N \subseteq \mathit{NAMES})_a$ action non-deterministically decides to unsubscribe to some set of names, issuing a subscription cancellation request to the home switch. It does not remove the names from its subscription set immediately, as the subscription request may yet dropped due to a conflicting subscription response.

This is consistent with treating *inNames* as an application-cached version of the home switch's *outNames* set and simplifies reasoning about Conflict Locality.

The send$(m \in M)_{c,a}$ action moves a pending message to the outbound channel.

$DynamicDataServerHost_b$

A refinement of *SimpleServerHost* that sends value messages, proactively or in response to value read requests.

We define the operator $\Delta : STATES \times NAMES \rightarrow \mathit{UPDATES}$, to yield for any global state s and name set N the update u such that for any global state t: $(t \oplus u) \mid N = s \mid N$. The operator can also be characterized as follows:

s Δ $N = \{ \ (n \mapsto v) \in$ s $\mid n \in N \ \} \cup \{ \ (n \mapsto \bot) \mid n \in N \setminus \mathit{names}(s) \ \}$.

Additional State:

pending: a subset of NAMES_b containing names whose values have been requested but not yet served, initially $\emptyset$.

Additional Input actions:

receive$(vq \in VQ)_{c,b}$
Effect:
    $\mathit{pending}' = \mathit{pending} \cup \mathit{names}(vq)$

Additional Internal actions:

value$(N \subseteq NAMES_b)_b$
Precondition:
    $N \subseteq \mathit{pending}$ and $N \neq \emptyset$
Effect:
    $outQueue' = outQueue \cdot VR(b.state \ \Delta \ N)$

$pending' = pending \setminus N$

broadcast$(N \subseteq NAMES_b)_b$
Effect:
$pending' = pending \cup N$

The receive$(sq \in SQ)_{c,b}$ action notes that a set of names has been requested, in its *pending* set. There is no need to conflict-check the request, since $depends(vq) = \emptyset$, by definition of value requests.

The value$(N \subseteq NAMES_b)_b$ action non-deterministically decides to send a value message for some requested subset of its names. It removes that set of names from the pending set.

The broadcast$(N \subseteq NAMES_b)_b$ non-deterministically adds names to the set of names pending for a value message. This actions models the fact that a server is free to send value messages without a value read request prompting it to do so.

6.6 Properties and Proofs

Let DCS be the I/O Automaton composed of an DynamicSwitch$_i$ automaton for each switch $s_i \in NODES$, a DynamicAppHost$_a$ automaton for each application host $a \in APPS$, a DynamicDataServerHost$_b$ automaton for each data server host $b \in DATAS$ and a Channel$_c$ automaton for each channel $c = (i, j) \in CHANNELS$.

The event ordering relation $<_E$ and *causes* relation $\rightarrow_E$ for an execution or execution trace E are the same as before, with the addition to $\rightarrow_E$ of each send$(vr \in VR)_{c,i}$ event caused by a receive$(vr \in VR)_{d,i}$ event, where c and d are channels incident to a switch $i \in SWITCHES$.

We begin with a simple invariant regarding the local state of DynamicAppHost$_a$.

Lemma 6.2 *(Synchronized Implies Subscribed): In any state of $DynamicAppHost_a$: a.synched $\subseteq$ a.inSub.*

Proof: Both fields are empty in the start state. Inspecting the actions of DynamicAppHost$_a$, we readily see that all changes to *a.synched* and *a.inSub* preserve the invariant □

It is easily shown that Lemma 5.3 of Chapter 5 holds even if node i is in *APPS* instead of *SWITCHES*, since the only role i plays in the proofs of these Lemmas is to enqueue an acknowledgement for a received subscription response, and DynamicAppHost acknowledges a subscription the exact same way. Hence, we can use the Lemma when proving the following Lemma, which effectively shows Name Conflict Locality for an application and its home switch.

Lemma 6.3 *(Application Name Conflict Locality): For each state t of each $X \in$ execs(DCS) and host $a \in APPS$: $n \in$ a.inSub $\Rightarrow$ subscribed$_a$(n, t).*

Proof: Fix any$X \in$ *execs*(DCS) and $a \in HOSTS$ communicating via channel $c = (a, i)$ with its home switch $i \in SWITCHES$. We proceed by induction on the prefixes of X. As our base case, the invariant trivially holds in the start state of X since *a.inSub* is empty. We claim that each action e extending a prefix X' into another prefix X'' of X preserves the invariant, because:

1. If e = receive($sr \in SR)_{c,a}$ then e removes names *unsub*(sr) from *a.inSub*, preserving the invariant, but it may also add a set N = *sub*(sr) to *a.inSub*. But by Lemma 5.3, *subscribed*$_a$(n, t) holds for each $n \in N$ after e, preserving the invariant.

2. If e = receive($sq \in SQ)_{c,i}$ then e may add names from *sub*(sq) to $j.outSub_c$, preserving the invariant, but may also remove names N = *unsub*(sq) from $j.outSub_c$. But then e also adds a cancellation response *cr* with *unsub*(*cr*) = N to $j.responses_c$, so the invariant is preserved.

3. If e = purge($N \subseteq NAMES)_{c,i}$ then e may remove a set of names $M = N \cap$ $nHop_i(c)$ from $i.outSub_c$. But then e also adds cancellation response $SR(\emptyset, M)$ to $i.responses_c$ and *subscribed*$_a$(n, t) holds, preserving the invariant.

4. There are no other events in DynamicAppHost$_a$ or DynamicSwitch$_i$ that affect $a.inSub$ or $i.outSub_c$, respectively.

Since all possible extensions of X' preserve the invariant it holds in X, completing the induction □

We prove lemmas regarding the order in which response and value messages from a data server b are delivered to a host a, somewhat corresponding to Lemmas 4.1, 4.2 and 4.3 in Chapter 4. Unlike the static CS system, an application a in DCS may receive only a subset of the update messages sent by a server b, due to dynamically varying subscriptions and non-deterministic purging of subscriptions and dropping of update messages by dynamic switches. Furthermore, dynamic switches may insert any number of cancellation responses into the sequence of responses from b before it reaches a.

We observe, though, that DynamicSwitch never changes the order of any responses (including value messages and subscription responses) in any of its *outQueue*s. We show that if a switch i does receive a pair of responses sent by some other switch j, then i forwards them in the same order as j.

Lemma 6.4 *(Dynamic switch response order): For each $E \in traces(DCS)$, any pair $i, j \in SWITCHES$ (including $i = j$) and any pair of responses m_1, $m_2 \in R$ forwarded along path $F = i,\ s_2,\ s_3,\ \ldots,\ j$: $receive(m_1)_{j,c} <_E receive(m_2)_{j,c} \wedge receive(m_1)_{j,c} \rightarrow_E send(m_1)_{i,d} \wedge receive(m_2)_{j,c} \rightarrow_E send(m_2)_{i,d} \Rightarrow_E send(m_1)_{i,d} <_E send(m_2)_{i,d}$, with c and d some channels incident to j and i, respectively.*

Proof: Let $m_1, m_2 \in R$ be a pair of responses received on a switch j in that order, where their reception causes their sending on a switch i.

We proceed by induction on path F. For the base case of a single switch $j = i$, let d be the channel on which j enqueues the messages. Since receive$(m_1)_{j,c} <_E$ receive$(m_2)_{j,c}$, m_1 is ahead of m_2 in $j.outQueue_d$ immediately after event receive$(m_2)_{j,c}$. Inspecting the actions of DynamicSwitch$_j$, we see that none of them can move m_2 in front of m_1 on $j.outQueue_d$; they only append and / or remove messages from $j.outQueue_d$.

Since send$(m \in M)_{i,d}$ sends and removes only messages from the head of $j.outQueue_d$, send$(m_1)_{j,d} <_E$ send$(m_2)_{j,d}$.

For the inductive step let $F' = s_k,\ s_{k+1},\ ...,\ j$ be the sequence containing the last $k+1$ switches of F, where k is less than the length of F. By our inductive hypothesis, send$(m_1)_{k+1,d} <_E$ send$(m_2)_{k+1,d}$, with d some channel incident to s_{k+1}. By channel FIFO, receive$(m_1)_{k,c} <_E$ receive$(m_2)_{k,c}$. Repeating the argument of the base case for s_k, we get send$(m_1)_{k,d'} <_E$ send$(m_2)_{k,d'}$, with $d' = (k-1, k)$, completing the induction $\square$

The responses received by applications in DCS do not have the simple correspondence with the responses sent by servers as they do in Chapter 4. Still, we show that during the interval where an application has a name n in its *synched* set, the sequence of values it caches for n is an interval of the sequence of values assigned to n at *host*(n). We refine our definitions of response event and message sequences to talk about the subsequences pertaining to particular names as follows:

Definition 6.4 *For any execution or trace E, $i \in$ NODES and $N \subseteq$ NAMES let respSndSeq$_i(E,N)$ (respRcvSeq$_i(E,N)$) denote the maximal subsequence of* trace(E) *or E, respectively, consisting of send$(r \in R \setminus SR \setminus VR)_{c,i}$ (receive$(r \in R \setminus SR \setminus VR)_{c,i}$) events such that $N \cap$* names$(r) \neq \emptyset$*, where c denotes some channel in CHANNELS$_i$, possibly different in different events.*

Let snd-seq-r$_{iN}(E)$ = messagesOf(*respSndSeq$_i(E,N)$*)*, the sequence of response messages sent from node i that contain a name in N. Let rcv-seq-r$_{iN}(E)$ = messagesOf(respRcvSeq$_i(E,N)$), the sequence of response messages received at i that contain a name in N. For any name $n \in$ NAMES we take snd-seq-r$_{in}(E)$ (rcv-seq-r$_{in}(E)$) to denote snd-seq-r$_{i\{n\}}(E)$ (rcv-seq-r$_{i\{n\}}(E)$).*

We also introduce some notation to help us talk about the execution interval during which a node is subscribed to a name, and the corresponding interval during which another node is transmitting responses containing that name.

Definition 6.5 *For any state t of any $X \in$* execs(DCS)*, any node $j \in$ NODES and any name $n \in$ NAMES with* nHop$_j(n)$ *= some channel d, let* sub-interval(j,n,t) *be* *undefined if* $\neg$subscribed$_{j,d}(n,t)$*, or else let it be the longest interval of* X *beginning immediately after the latest event* e_r = *receive*($sr \in SR)_{d,j}$ *event before* t *in* X *such that* $n \in$ *sub*(sr) *and for every state* $t' \in$ *sub-interval*(j,n,t) : *subscribed*$_{j,d}(n,t')$. *We call such an interval a* subscription interval *of* j *for* n.

For any node $i \in (fp_{jb} \cup \{b\})$ *where* b = *host*(n)*, let* pub-interval(i,j,n,t) *be the longest interval of* X *that begins immediately after an event* e_s = *send*$(sr)_{c,i}$ *such that* e_s *causes the event* e_r = *receive*$(sr \in SR)_{d,j}$ *defining a subscription interval* sub-interval(j,n,t) *and for each state* $s' \in$ *pub-interval*(i,j,n,t) : $n \in i.outSub_c \vee i = b$, *where channel* c *is incident to* i. *We call such an interval a* publishing interval *of* i *for* n.

Figure 6.1 gives an example of a subscription interval and a corresponding publishing interval. We observe that if a *sub-interval*(j,n,t) is defined then there exist for each switch $i \in fp_{jb}$ exactly one corresponding *pub-interval*(i,j,n,t). By Lemma 3.2 channels do not spontaneously create new messages, so the event e_r = receive(sr) must be caused by a particular chain of send(sr) events. We present a simple lemma saying that subscription intervals exist for every name in a node's *inSub* set.

Lemma 6.5 *(inSub and subscription intervals) For any state* t *of any* $X \in$ execs(DCS)*, any node* $j \in NODES$ *and any name* $n \in j.inSub_d$ *for a channel* d *incident to* j *(or* $j.inSub$*, if* $j \in HOSTS$*) in state* t : sub-interval(j,n,t) *exists.*

Proof: Inspecting DynamicSwitch and DynamicAppHost we see that n is only added to $j.inSub(_d)$ by an event e_r = receive$(sr \in SR)_{d,j}$ action with $n \in$ *sub*(sr), so e_r is an event before t that defines an interval *sub-interval*(j,n,t) □

The next Lemma says that once a node (application host or switch) receives the subscription confirmation message for a name, the sequence of responses it receives for that name is a prefix of the sequence of responses sent by each node on the forwarding path to the server hosting that name, for as long as the receiving node stays subscribed to the name. It corresponds to Lemma 4.3 but only applies to a particular name and only during its subscription intervals. Figure 6.1 gives an illustration of the main entities involved in the Lemma. It shows *abcd* as an example of the *rcv-seq-r*$_{jn}(I_j)$

for a subscription interval I_j, which is a prefix of *snd-seq-*$r_{in}(I_i)$ = *abcdefg...* of the corresponding publication interval I_i at i. The dashed arrows shows an example of where intervals I_j and I_i might end, due to a cancellation response sent by i.

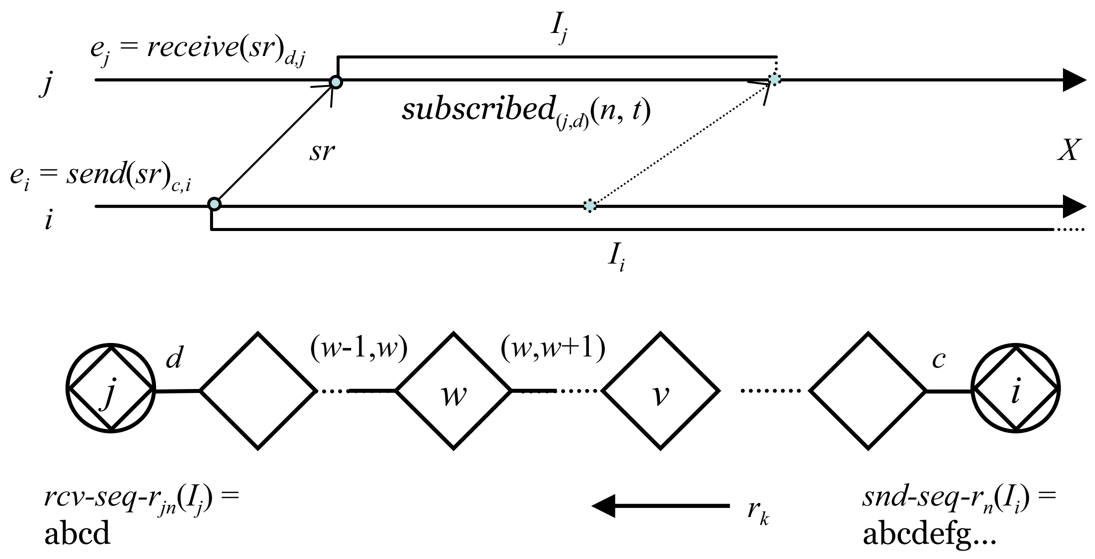


Figure 6.1: The publish/subscribe Intervals of Lemma 6.6

Lemma 6.6 *(Dynamic message order): For any $X \in$ execs(DCS), $j \in$ NODES, b $\in$ DATAS and $n \in NAMES_b$ with* nHop$_j(n)$ *= some channel d, let I_j be any sub-interval(j,n,t) for some state $t \in X$ starting with an event e_j = receive(sr $\in$ SR)$_{d,j}$ and let I_i be any corresponding* pub-interval(i,j,n,t) *for some $i \in$ (fp$_{jb}$ $\cup$ {b}). Then rcv-seq-$r_{jn}(I_j) \preceq$ snd-seq-$r_{in}(I_i)$.*

Proof: If *rcv-seq-*$r_{jn}(I_j)$ is empty then the theorem holds vacuously. Otherwise, by Lemma 6.4 and channel FIFO, every pair of messages from *sndSeq*$_{c,i}(X,i,R)$ that is received by j is received in the order sent by i. Hence, *rcv-seq-*$r_{jn}(X)$ is a subsequence of *snd-seq-*$r_{in}(X)$. Since I_i starts with the e_i = send$(sr)_{c,i}$ event that causes the e_j = receive$(sr \in SR)_{d,j}$ event marking the beginning of I_j, and since by Lemma 6.4 and channel FIFO all messages in R sent by i are received by j in the order sent, we have that *rcv-seq-*$r_{jn}(I_j)$ is a subsequence of *snd-seq-*$r_{in}(I_i)$. To show that *rcv-seq-*$r_{jn}(I_j)$ $\preceq$ *snd-seq-*$r_{in}(I_i)$ holds, it suffices to show that all the messages of *snd-seq-*$r_{in}(I_i)$ are received, and none are dropped.

Assume for contradiction that the k-th response r_k of $\textit{snd-seq-r}_{in}(I_i)$ is dropped. Hence, the k-th response in $\textit{rcv-seq-r}_{jn}(I_j)$ is not the k-th response from $\textit{snd-seq-r}_{in}(I_i)$ but rather the h-th response r_h from $\textit{snd-seq-r}_{in}(I_i)$, for some $h > k$. We show that in this case, j receives a cancellation response containing n during I_j, contradicting the fact that $\textit{subscribed}_{(j,d)}(n,\ t)$ holds in each state t of I_j.

There are only two ways that r_k can fail to be forwarded to j: if r_k is received at a switch where the channel leading to j is not subscribed to r_k or if r_k is purged at some switch. Let s_1, s_2, ..., s_m be the switches comprising the forwarding path $\textit{fp}_{ji}$ from j to i, so $s_1 = \mathrm{s}_j$ and $s_m = s_i$. We observe that the since sr was received at j, the switches of $\textit{fp}_{ji}$ must all be subscribed to n at some point in I_i, that is: for $1 \leq l \leq m$, $n \in s_l.\textit{outSub}_{(l-1,l)}$ immediately after each receive$(sr)_{(l,l+1),l}$ that causes the e_j = receive$(sr)_{d,j}$ event marking the beginning of I_j, since $n \in \mathit{names}(sr)$. Furthermore, since by Lemma 6.4, receive$(sr)_{(l,l+1),l} <_X$ receive$(r)_{(l,l+1),l}$ for any $r \in \textit{rcv-seq-r}_{jn}(I_j)$ and $1 \leq l \leq m$, r_k is not dropped due to arriving at a switch before the subscription for n is established at that switch; it must be because the subscription to n was established and then removed again before r_k was received. Finally, we observe that at any switch l on the path that receives r_k in I_i, receive$(r_k)_{d,j} <_X$ receive$(r_h)_{d,j}$, by Lemma 6.4. Let us consider first the case when r_k fails to be forwarded and then the case when r_k is purged.

1. In the first case, let s_v be the switch on $\textit{fp}_{ij}$ such that $v.\textit{outSub}_{(v,v-1)} \cap \mathit{names}(r_k)$ $= \emptyset$ at event e_{vr} = receive$(r_k)_{(v+1,v),v}$, because n is removed from the subscription of one or more switches on $\textit{fp}_{ji}$ before e_{vr}. Let s_w be the first switch on $\textit{fp}_{ji}$ to remove n from $w.\textit{outSub}_{(w-1,w)}$ after its receive$(sr)_{(w,w+1),w}$ event, so $w \leq v$. The event e_{wcr} removing n is one of receive$(sr \in CR)_{(w,w+1),w}$ with $n \in \mathit{unsub}(sr)$, purge$(N \subseteq \mathit{NAMES})_{(w-1,w),w}$ with $n \in N$ or receive$(sq \in SQ)_{g,w}$ with $n \in \mathit{unsub}(sq)$. In each case, e_{wcr} enqueues a cancellation response $cr \in SR$ on $\textit{outQueue}_{(w-1,w)}$, where $n \in \mathit{names}(cr)$. Since receive$(r_k)_{(v-1,v),v} <_E$ receive$(r_h)_{(v-1,v),v}$ and since $w \leq v$ and $e_{wcr} <_X e_{vr}$, by Lemma 6.4, receive$(cr)_{d,j}$ $<_E$ receive$(r_h)_{d,j}$.

2. In the latter case, r_k is dropped by an event e_v = purge$(N \subseteq \mathit{NAMES})_{(v-1,v),v}$ event at some switch v on $\textit{fp}_{ji}$. If $n \notin v.\textit{outNames}_{v-1,v}$ at e_v then n is removed by some earlier event at v or some other switch $w < v$ (since e_v is the first and

only event that drops r_k). By the argument for the first case, the earliest event e_{wcr} dropping n from an *outSub* field on *fp*$_{ji}$ enqueues a cancellation response $cr \in SR$ where $n \in \mathit{names}(cr)$, and since $w \leq v$ and $e_{wcr} <_X e_v$, by Lemma 6.4, receive$(cr)_{d,j} <_E$ receive$(r_h)_{d,j}$. On the other hand, if $n \in v.\mathit{outSub}_{(v-1,v)}$ at e_v then e_p enqueues a cancellation response message $cr \in SR$ on $\mathit{outQueue}_{(v-1,v)}$, where $n \in \mathit{names}(cr)$. Since e_v drops a contiguous sequence of responses from the tail of $v.\mathit{outQueue}_{(v-1,v)}$ which does not include r_h, r_h is received and added to $s.\mathit{outQueue}_{(v-1,v)}$ after e_v. Hence, cr is ahead of r_h on $s.\mathit{outQueue}_{(v-1,v)}$ so once again by Lemma 6.4, receive$(cr)_{d,j} <_E$ receive$(r_h)_{d,j}$.

Our assumption that r_k is dropped implies that a receives cr ahead of r_h. This means that both receive events occur in interval I_j. But then j removes n from $j.\mathit{inSub}_d$ (or $j.\mathit{inSub}$, if $j \in \mathit{HOSTS}$) in I_j and we have $\neg\mathit{subscribed}_{(j,i)}(n,\ t)$ in the state t following receive$(cr)_{j,d}$, which contradicts the fact that for every state $t \in I_j$: $\mathit{subscribed}_{(j,d)}(n,\ t)$. Hence, response r_k cannot have been dropped and *rcv-seq-r*$_{jn}(I_j) \preceq$ *snd-seq-r*$_{in}(I_i)$ $\square$

In static system CS there is a direct correspondence between server states and the sequence of responses received by a subscribing application. In the dynamic system, the correspondence is more complex because applications subscribe to a subset of a server's responses and their subsets may vary during executions. This added complexity is reflected in our proofs. As a preliminary, we extend lemma 6.6 to talk about the correspondence between the states of a server and the sequence of responses received by an application following a subscription confirmation message. We begin by defining the response sequence of a server with respect to a particular name.

Definition 6.6 *For any execution or trace E and $n \in NAMES$, let resp-seq-r$_n(E)$ denote the maximal subsequence of resp-seq-r$_b(E)$ such that for each response r in the sequence:* names$(r) \cap N \neq \emptyset$, *where b =* host(n).

Lemma 6.7 *(dynamic state change and response order): For any $X \in$ execs(DCS), $a \in APPS$, $b \in DATAS$ and $n \in NAMES_b$, let I_a be any* sub-interval(a,n,t) *for some state $t \in X$ and let I_b be the corresponding* pub-interval(a, b, n, t). *Then rcv-seq-r$_{an}(I_a) \preceq$ resp-seq-r$_n(I_b)$.*

Proof: For each response $r \in$ *resp-seq-*$r_b(I_b)$, in order from first to last, b sets its *b.state* field to *b.state′* $=$ *TRANS*$_b$(*b.state*, r) while simultaneously enqueueing r on *b.outQueue*. Since names(r) by definition contains every name whose value may be different in states *b.state* and *b.state′*, if *b.state*$(n) \neq$ *b.state′*(n) then $n \in$ names(r). By the FIFO property of *b.outQueue* and the definition of *snd-seq-*$r_{bn}(I_b)$, *snd-seq-*$r_{bn}(I_b)$ $\preceq$ *resp-seq-*$r_n(I_b)$. By Lemma 6.6, *rcv-seq-*$r_{an}(I_a) \preceq$ *snd-seq-*$r_{bn}(I_b)$, so *rcv-seq-*$r_{an}(I_a)$ $\preceq$ *snd-seq-*$r_{bn}(I_b) \preceq$ *resp-seq-*$r_n(I_b)$ $\square$

We can now show that the sequence of values assigned to the names in a cache mirrors the sequence of values assigned to the names on the original server. We show that the projection of a cache on its synched names equals the same projection on a recent or current state of the server.

Lemma 6.8 *(dynamic cache synchronization): For any state t_a of any $X \in$ execs(DCS), $a \in$ APPS, $b \in$ DATAS, let synched$_b$ = a.synched $\cap$ NAMES$_b$ in t_a and let t_b be the state in X immediately after the e_b event enqueueing the latest response from b that has been received by a. Then a.cache | synched$_b$ in state t_a = b.state | synched$_b$ in state t_b.*

Proof: We proceed by induction on the prefixes of X. As our base case, the invariant trivially holds in the start state of X since *a.synched* is initially empty. We claim that each action e extending a prefix X' into another prefix of X preserves the invariant.

As a preliminary, since *a.synched* $\subseteq$ *a.inSub* then by Lemma 6.5 there exists for the final state t_a of X' and each $n \in$ *a.synched* at t_a a suffix I_{an} = *sub-interval*(a,n,t_a) of X' where for each such I_{an}, by Lemma 6.7, *rcv-seq-*$r_{an}(I_{an}) \preceq$ *resp-seq-*$r_n(I_{bn})$, where I_{bn} = *pub-interval*(a, b, n, t_a). In other words, a is receiving every response affecting any name in *a.synched*, in the order of the corresponding state changes.

Let e_b be the event that enqueues the latest response message sent from b that has been received by a at t_a. By the inductive assumption, in the final state t_a of X', s_{ab} = *a.cache* | *synched*$_b$ equals s_b = *b.state* | *synched*$_b$ in the state t_b after e_b.

1. If e = receive$(sr \in SR)_{d,a}$ then e may remove set N = *unsub*(sr) from *a.synched*, which preserves the invariant.

2. If e = receive($vr \in VR)_{d,a}$ then e may add a set $N = (a.synched' \setminus a.synched) \cap a.inSub$ of names to $a.synched$. Let e_v be the value($N \subseteq NAMES_b)_b$ event of DynamicDataServerHost$_b$ that enqueues vr. Since $a.synched' = (a.synched \cup N) \subseteq a.inSub$, a receives all responses r such that $names(r) \cap a.synched' \neq \emptyset$, and by Lemma 6.4, it receives them in the order sent by b. Therefore, b cannot have sent a response r' affecting N between e_b and e_v. Since e_v does not modify $b.state$, $b.state \mid synched_b$ at $e_v = s_b$. Since e updates set N of names in $a.cache$ with values from vr, we have that $s'_{ab} = a.cache \mid a.synched'$ in the state t'_a following e equals s_b in the state t'_b following e_v. Since e_b is still the event enqueueing the latest response message from b that has been received by a in t'_a and $s_b = s'_{ab}$, the invariant holds after e.

3. If e = receive($r \in R \setminus SR \setminus VR)_{d,a}$ then e updates $a.cache$ with $TRANS_b(a.cache, r)$. If $names(r) \cap synched_b = \emptyset$ then e has no effect on $a.cache \mid synched_b$. Otherwise, let e'_b be the receive($q \in$ Q$)_{c,b}$ event of DynamicDataServerHost$_b$ that enqueues r. By our preliminary reasoning here above, a receives the response for each state change affecting names in $synched_b$ in the order of the state changes. Therefore, e'_b is the first event following e_b that modifies $b.state \mid synched_b$ and $b.state \mid synched_b$ at $e'_b = s_b$. By our inductive assumption, $s_{ab} = s_b$. Since response r is separable, $TRANS_b(s_b, r) \mid synched_b = TRANS_b(s_{ab}, r) \mid synched_b = s'_b \mid synched_b$, that is: both e'_b = receive($q \in Q)_{c,b}$ and e = receive($r \in R \setminus SR \setminus VR)_{c,a}$ compute the same values for the names in $synched_b$. Since e'_b sets $b.state$ to s'_b and e overrides $a.cache \mid synched_b$ with $s'_b \mid synched_b$, we have that $a.cache \mid synched_b$ in the state t'_a following $e = b.state \mid synched_b$ in the state t'_b following e'_b. Since e'_b is the event enqueuing the latest response message from b that has been received by a in t'_a, the invariant holds after e.

4. No other events affect the invariant □

Before we can prove the main Theorem, we must show the following:

Lemma 6.9 *DynamicAppHost$_i$ preserves ACK Well-Formedness.*

Proof: Immediate, since the only action in DynamicAppHost$_i$ that enqueues $ack(r)$ is receive$(r)_{c,i}$□

Lemma 6.10 *DynamicDataServerHost$_i$ preserves Responder Well-Formedness.*

Proof: Since DynamicDataServerHost$_i$ is identical to SimpleDataServerHost$_i$ except for the additional value($U \subseteq NAMES$)$_b$ action, the proof of Lemma 4.4 from Chapter 4 applies □

We can now show that if a dynamic server executes a request q then the response is among those possible given the dynamic application's cached state at the time the application issued q. Our proof is quite similar to the one in Chapter 4 for Theorem 4.1, showing that if this were not the case, then the intervening creation of a response that altered the state of the server would have caused a response that conflicts with q, and q would have been dropped.

Theorem 6.1 *Dynamic Cache Operation Atomicity: For any execution $X \in execs(DCS)$, application $a \in HOSTS$, server $b \in DATAS$ and channels d and c incident to a and b, respectively: send$(q \in Q)_{d,a} \rightarrow_X$ send$(r_q \in R)_{c,b} \Rightarrow r_q \in execute_b(s_a, q)$, where s_a = a.cache $|$ depends(q) at the createRequest event e_q that enqueues q.*

Proof: Assume for contradiction that there exists some execution $X \in execs(DCS)$ and server $b \in DATAS$ where send$(q)_{d,a} \rightarrow_X$ send$(r_q \in R)_{c,b}$ but $r_q \notin execute_b(s_a, q)$.

By inspecting the createRequest action, we see that *depends*$(q) \subseteq$ *a.synched* at e_q, so by Lemma 6.8, *b.state* $|$ *depends*$(q) = s_a$ at some earlier point in X, before the receive$(q)_{c,b}$ event that executes q. Therefore, there must be an event e_r at b preceding receive$(q)_{c,b}$ that assigns to *b.state* a value s_b such that $r_q \notin execute_b(s_b, q)$. If we let r be the response enqueued by e_r, then *conflicts-inv*(q, r), by the definition of *conflicts-inv*. Since *a.cache* $|$ *depends*$(q) = s_a$ at the createRequest$(q)_a$ event issuing request q, we have createRequest$(q)_a <_X$ receive$(r)_{d,a}$ and therefore by FIFO send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. This gives us $e_r <_X$ receive$(q)_{c,b} \wedge$ send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. Since by Lemma 6.9 a is ACK Well-Formed and by Lemma 6.10 b is Responder Well-Formed, then by Theorem 5.2 we have send$(q)_{d,a} \not\rightarrow_X$ send$(r_q)_{c,b}$, a contradiction □

6.7 Liveness

We show that DynamicAppHost$_a$ preserves Dynamic Subscription Well-Formedness, as in definition 5.6. Hence, liveness theorem 5.3 applies to DynamicAppHost$_a$.

Lemma 6.11 *(DynamicAppHost Preserves Well-Formedness): In any $X \in execs(DCS)$, DynamicAppHost$_a$ preserves Dynamic Subscription Well-Formedness for a.*

Proof: Fix any event $e = \text{receive}(q \in Q)_{c,i}$ in any $X \in execs(DAS)$, where i is the home switch of a and $c = (a, i)$. We must show that $subscribed_a(n, t)$ holds for each $n \in depends(q)$ in the state t preceding e. Inspecting DynamicAppHost$_a$, we see that e is caused by an earlier event $\text{send}(q \in Q)_{c,a}$, in turn due to an earlier event $e' = \text{createRequest}(q \in Q)_a$ that enqueues q. The precondition of createRequest requires that $depends(q) \subseteq synched$, so by Lemma 6.2, $depends(q) \subseteq inSub$ in the state t' at e'. By Lemma 6.3, therefore, $subscribed_a(n, t')$ holds for each $n \in depends(q)$, that is: $n \in i.outSub_c$ or $\exists\ cr \in i.responses_c \cap SR : n \in names(cr)$.

1. In the case where $n \in i.outSub_c$ in t', if also $n \in i.outSub_c$ in t then $subscribed_c(n,\ t)$ holds. On the other hand, if $n \notin i.outSub_c$ in t then name n is removed from $i.outSub_c$ by some intervening event e_n, where $e' <_X e_n <_X e$. Event e_n must be one of $\text{receive}(sr \in SR)_{d,i}$ with $n \in unsub(cr)$ and $d \in CHANNELS_i$, $\text{purge}(N \subseteq NAMES)_{c,i}$ with $n \in N$ *or* $\text{receive}(sq \in SQ)_{c,i}$ with $n \in unsub(sq)$. In each case, e_n adds a cancellation response cr_n to $i.outSub_c$, where $n \in effects(cr_n)$. By FIFO, the earliest $\text{receive}(ack(cr_n))_{c,i}$ event occurs after e, so $cr_n \in i.outSub_c$ at e and $subscribed_c(n,\ t)$ holds.

2. In the case where $\exists\ cr \in i.responses_c \cap CR : n \in names(cr)$ at t', then a has not yet received and ACKed cr at e', since that would imply $n \notin a.inSub$ at e'. Since an ACK for cr is sent after e' by FIFO the earliest $\text{receive}(ack(cr))_{c,i}$ event occurs after e, so $cr \in i.outSub_c$ at e and $subscribed_c(n, t)$ holds $\square$

6.8 Optimizations

Since applications can receive their value messages in an arbitrary order, it is possible for a and/or b to prioritize the order of value messages, as to minimize the synchronization delay of the most important requests or to refresh the values that are most highly visible to end-users first, for example. Application a could hint at the desired order when it requests the values and/or the server could use application-specific knowledge to govern the order in which is sends values. This is orthogonal to the state transfer protocol, though, and we do not explore the issue further here. We note that these kinds of optimization also work with the scalable cache protocol presented in Chapter 7.

Another orthogonal optimization would be to enable host a to receive values for only those names in some set $N_a \subseteq \mathit{NAMES}$ whose values may have changed since a last subscribed to N_a. In other words, the host could receive a (non-conflict checked) *delta update* that transforms its cache from that old state t' to some more recent state t. The host would then receive normal value update messages to complete the move from t to the current state. To sketch an example, server b could periodically tag its current state as a checkpoint state and simultaneously send a checkpoint marker message to its subscribers, who would tag their current cached states with the marker. Upon later re-subscribing, an application could send its latest marker with its value request, and receive delta and value messages only for state that has changed since the marker was created. The main requirement for any such algorithm is that it satisfy the consistency guarantees of Lemma 6.8.

6.9 Discussion and Related Work

This section discuss issues related to the abstraction level of responses as well as related work in optimistic concurrency control.

6.9.1 Responses and Abstraction Levels

As discussed in section 6.2, we make the simplifying assumption that each response $r \in R$ is *separable*, so it can be individually applied to each name in its *effects* set. Many operations that are useful for increasing concurrency, such as increment/decrement operations, are not affected by this restriction. However, it rules out responses that depend on one name to compute the effects on another name, such as assigning to one name the average of the values of a set of names. More generally it rules out some high-level course-granularity responses, such as executing the action associated with a certain menu item on a user interface, since it may depend on the values of arbitrarily many names.

The attraction of supporting such responses is that they permit low-bandwidth caching of application state during periods of low contention. During the time a user is exclusively editing a particular part of a CAD model or text document, for example, it suffices to send high-level operation request such as "combine the currently selected objects into one" or "cut the currently selected paragraph". Rather than create a response containing a large set of fine-grained operations on a large set of names, the server could send back a response that is very similar to the request, i.e. "combine set N of objects" or "cut paragraph X".

The problem with high-level responses, as mentioned before, is that subscribers cannot apply them unless they are synched with all the names they depend on. Furthermore, large-granularity operations imply large-granularity conflicts, at worst requiring total ordering of all operations in a system.

One approach we would like to investigate in the future is to structure a response as a tree, corresponding to its refinement from a top-level response into the sub-responses generated at successively lower levels of abstraction. For example, a "move text" operation might refine to a pair of "delete text" and "insert text" operations and ultimately to "rewrite the following disk blocks". This concept aligns nicely with nested [62] or multi-level transactions [157], where transactions are modeled as trees of nested sub-transactions. It would provide flexibility in trading bandwidth and server processor cycles off with subscription set sizes and conflict granularities. Some application would maintain a coarse-granularity subscription to a relatively large subset of data but receive only a low-bandwidth stream of high-level responses (the roots of response trees, for example). Other applications would maintain more specific, finer-granularity subscriptions to parts of that same data but receive higher-bandwidth streams of the lower levels of response trees, for example the leaves consisting of only separable responses. As an example of the two extremes, an end-user application might be cached at the end-user access device and replicated in whole by sending as requests and receiving as responses the low-bandwidth stream of high-level mouse and keyboard gestures input to it. On the opposite end of the spectrum, the device could receive high-bandwidth bitmap images of the application's user interface.

6.9.2 Caching and Optimistic Concurrency Control

Our AT-based Cache Consistency algorithm (AT-Cache, for short) is a form of Optimistic Concurrency Control [15], since application hosts first execute transactions in a local data space and then attempt to validate the execution at the server. Pessimistic methods such as locking [1], by contrast, detect conflicts between transactions as they execute and block (delay) later transactions until earlier transactions have completed. A key difference between these methods is that locking limits host utilization by blocking transactions while optimism limits it by wasting resources on transactions that are later aborted and restarted.

The usual performance metric for transactional information system is sustained throughput of transactions per second. Transaction response time is often a concern as well. Generalized comparison of concurrency control methods is difficult, as their performance depends substantially on multiple factors [158, 159], including:

1. The level of concurrency / multiprogramming.
2. The level of data contention, defined as the ratio of transactions that conflict or the number of conflicting accesses to any particular datum.
3. The amount of computing and I/O (e.g. disk) resources consumed by transactions and degree of resource contention.
4. The ratio between read-only and read/write transactions and operations.

5. Network bandwidth and latency.

6. Transaction access patterns and cache hit rates .

In centralized settings, simulations and analysis suggest that optimism outperforms locking in environments that are not significantly resource-constrained and where data contention is moderate to low [158, 160]. When conflicts are rare, optimism needs to restart few transactions. When resources are plentiful, the multiprogramming level can be kept higher than with blocking, even as restarted transactions waste resources. However, the same studies indicate locking performs better with hight contention in centralized settings. While OCC has been a research topic for more than 25 years, it has seen limited use in commercial database systems, which use locking almost exclusively.

However, optimism may be particularly well suited to client/server transactional cache systems [16]. Interest in such systems has coincided with the ascendancy of powerful workstations and commodity PCs over minicomputer and mainframes, and the subsequent proliferation of data centers, comprising tens or hundreds of thousands of hosts. An important reasons why client caching suits optimism is that aborted transactions affect a server much less than in the centralized case, where a server must maintain undo logs and actively roll back the changes of aborted transactions. In the case of client caching, an aborted transaction will only have changed the client's cached data and the client handles the undoing of those changes.

6.9.3 Adaptive Optimistic Concurrency Control (AOCC)

The most relevant work to AT-Cache are the OCC algorithms developed for the Thor distributed object database system [67, 14]. The initial algorithm was shown to outperform the best lock-based method for the same environment, Adaptive Callback Locking (ACBL) [161], with low to moderate contention. A later refinement, Adaptive Optimistic Concurrency Control (AOCC) [162], outperforms ACBL for most workloads. Another OCC algorithm designed for a similar client-caching environment also outperforms locking under a variety of workloads [134].

AOCC keeps track of recently committed transactions and validates incoming transactions against them. This is much faster than validating transactions directly against the data they depend on (or data versions numbers) since that might entail fetching the data from disk[14]. Simplifying a bit, for each data object x, an AOCC server tracks all clients that cache a copy of it. When x is modified, the server adds x to the *invalid set* for each client that caches x. As a part of validating a transaction t_c from a client c, AOCC checks that no object read by t_c is in the invalid set of c. When transactions commit, AOCC sends invalidation messages to clients to notify them about updates to objects. It removes objects from invalid sets upon receiving invalidation acknowledgements from clients.

AT-Cache, by comparison, offloads the responsibility for validating requests to the network, by asking switches to conflict-check responses. In effect, rather than having the server remember invalid sets for each application host on an end-to-end basis, the network paths remember the sets as they forward them to their receivers, and take care of processing acknowledgements on a hop-by-hop basis.

Let n be the number of requestes/clients actively using a data server. The main differences between AOCC an AT-Cache can be quantified as follows:

1. An AOCC server must maintain n invalid-sets, one per client. An AT-Cache server does not maintain invalid-sets, or rather: maintains exactly one, for its home server. Furthermore, an AOCC server must retain invalid-set information until it receives an end-to-end ACK from the corresponding client, while an AT-server only retains it until it gets a next-hop ACK from its home switch.
2. An AOCC server must send on the order of n invalidation messages for each request it executes[15] and process on the order of n ACKs sent back by clients. An AT-Cache server sends a single response message that is multicast to all subscribers.
3. An AOCC server must maintain "subscription" information for n clients, to track which objects each client is caching. It must also process $O(n)$ subscription changes, as clients add and remove objects from their cache. In Atomic

[14]This was a major cause for poor performance in some earlier OCC-based systems, as validation was serialized under a global lock.

[15]The precise number depends on the number of clients subscribing to modified objects.

Networks, by contrast, subscription processing can be distributed throughout the network.

4. An AOCC server must send $O(n)$ data object messages, as clients fetch objects into their cache. By comparison, Chapter 7 of this dissertation shows how Atomic Transfer can be used to distributed such reads across multiple caching hosts, limiting the load on the original server.

5. An AOCC server must receive and process all conflicting requests sent to it, and send a conflict notification message for each one. An AT-Cache server receives only a fraction of the conflicting requests sent to it (see Section 3.7.1) and does not need to send any conflict notifications.

This leads us to believe that AT-Cache scales better than AOCC and could outperform AOCC, especially during periods of heavy data contention. Some of the good properties of AT-Cache stem from the use of multicast, more than Atomic Transfer per se. Indeed, the scalability of AOCC might be improved by multicasting invalidation and conflict notifcation messages, instead of sending them individually. Still, an AOCC server would need to receive and process individual ACKs.

AT-Cache should preserve throughput during periods of high contention better than AOCC. Let n be defined as before. Let m_x be the average number of requests sent to a server per second during some period of time that are successfully executed. Let m_c be the average number of requests sent to the server per second during that same period that have a conflict and cannot be executed. The AOCC server must receive and process $m_x + m_c$ messages during that period and send $O(n \cdot m_x)$ invalidations and m_c conflict notifications. By comparison, the AT-Cache server receives on the order of m_x messages, since only a fraction of the m_c conflicting requests are received (see Section 3.7.1), and sends $O(m_x)$ responses. If message processing (receiving, detecting concurrency conflicts, sending and possibly and authenticating, decrypting and encrypting) comprises a significant part of a server's effort, then the utilization of an AOCC server may be limited to $m_x/(m_x + m_c)$, the fraction of requests that are non-conflicting. By comparison, an AT-Cache server should be able to maintain nearly full utilization, regardless of contention levels.

Chapter 7

Scalable Cache Synchronization

The subscription protocol of Chapter 5 is scalable, since additional switches allow a system to handle a greater number of application hosts. On the other hand, the cache synchronization protocol of Chapter 6 does not scale, since the server of a name n must handle the value read requests of all applications synchronizing with n. To alleviate this bottleneck, this Chapter develops an algorithm that enables applications to synchronize their caches using the caches of other application hosts. This allows growth in synchronization traffic to be met with additional host resources. The caching is transparent to applications, including applications serving as caches to other applications; they do not distinguish between value messages from application caches and original servers, in general. Caches and cache hierarchies use conflict detection on switches to synchronize state transfers and concurrent response messages, preserving atomicity.

7.1 Cacher Hosts and Cacher-Aware Switches

The protocol works roughly as follows. An application c subscribing to some set N_c of names on a server b may offer its services as a *cacher host* (cacher) for N_c. Host c notifies the switches on the forwarding path from itself to b about its intention to serve as a cacher, by sending protocol messages or piggybacking data on subscription requests, for example (we leave that detail unspecified in our model). A cacher-aware switch i notes for each channel $d \in \mathit{CHANNELS}_i$ the *cacher subset* of its subscription on d, that is: the subset of names forwarded on d for which one more hosts have declared themselves as cachers.

As illustrated in Figure 7.1, whenever switch i receives a read request vq for a set of names that falls in the cacher subset of some channel $d \in CHANNELS_i$, it may choose to *divert* the request away from its destination server and onto d. The request is converted into a diverted value request dvq, tagged as being diverted by i so that other switches continue the diversion, forwarding it (non-deterministically) until it arrives at some cacher host. The cacher host responds by sending back one or more diverted value message(s) dvr to satisfy the read request. The cacher host tags its diverted value messages with index i and switches forward them towards the home server(s) of the names they contain, instead of towards subscribers like normal value messages[16]. When switch i receives a value message with diversion tag i on it, it removes the tag and forwards the value message towards subscribers as a normal value message vr, same as if vr had just arrived from the original server.

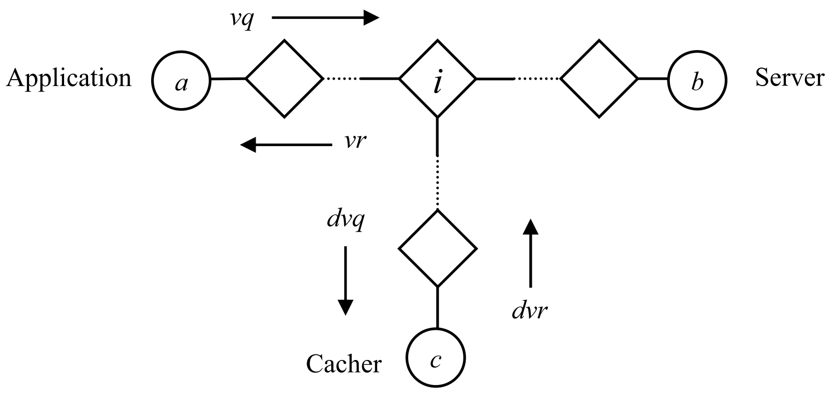


Figure 7.1: Diversion of value read request to a Cacher Host

The protocol outlined can both serve to increase the number of applications that can be handled by a particular data server and/or to directly reduce application synchronization delay, as applications can (transparently) peruse multiple caches in parallel. We note that an application host can serve as a cacher alongside its primary duties as an application host, improving system utilization.

[16]That is to say: they forward them like requests.

7.2 Protocol Race Condition and Solution

The protocol sketched in the prior Section has a race condition, which occurs when the server $b = host(n)$ of a name n sends a response r updating n at the same time some cacher $c \in APPS$ is sending a value message vr containing an older value for n. While c is by definition subscribed to n and will eventually receive response r, r may reach the diverting switch s_i connecting b and c before vr does. Response r hence *overtakes* vr at s_i, and a subscribing application a that is in the process of synchronizing with n will receive r ahead of vr. Since n will not yet be be synchronized at a when it receives r, a will ignore r and miss its effects, but install the stale value from vr as the current cached value of n.

We prevent this race condition using Atomic Transfer, by defining a diverted value message as conflicting with any response whose effect overlaps the message's name set. Dropping conflicting diverted value messages would ensure that any value message reaching an application would be guaranteed to be fresh and up to date, since it encountered no conflicting response along the way. However, this straightforward approach could lead to livelock, with caches repeatedly sending out new value messages that conflict with the latest responses affecting some frequently updated name(s) and are consequently dropped.

Instead of dropping a conflicting value message, therefore, we make the switch detecting the conflict *freshen* the diverted message, by appending the conflicting response to a list of responses contained in the value message. An application host a receiving a value message vr applies the response list of vr immediately after applying vr's update (but atomically with the application of vr), resulting in correct synchronization of the names.

As it happens, switches do not actually need to add conflicting responses to a value message's list. We observe that an application a only sends a value request for names it is subscribed to, so a will generally have received the responses on the list already. In our approach we let a buffer all responses affecting a name n from the time it sends a value request for n until the time it has synchronized n. Then it suffices to include in value messages a list of the identifiers of any conflicting responses, which a uses to look up the corresponding responses in its own stored sequence of recently received responses. These identifiers could be implemented as message sequence numbers or, alternatively, as digests of the corresponding responses. In either case, the identifiers would be smaller than the response messages themselves, preserving bandwidth.

We can in fact reduce the size of the response identifier list in messages to unity, by noting that an application a receives its responses in the same order as they would appear on a identifier list. It therefore suffices to include the identifier of the *first* conflicting response r_f in the list, when the list is non-empty. Application a can then infer the rest of the list by storing and checking each response in the sequence of responses received after r_f but before vr for conflicts with vr. This means additional work for a but simpler processing in the network, since only a fixed-size message field needs to be updated.

This does require that responses be unique, so that a particular response is sent at most once in any execution. Otherwise the first conflicting response might not unambiguously correspond to a list of conflicting responses. This assumption is likely to hold in practice, as implementations of transactional systems usually include a sequence number with requests to ensure exactly-once execution semantics and to match responses to requests in the presence of failures and partitions[17].

Since subscriptions may be added and removed at any time, a must take care that it always infers a complete list of responses for a value message, with no responses missing. For example, if a's subscription to a set of names C is canceled after a requests a set of names $N \supseteq C$ (due to a purge on some switch, for example) but a re-establishes the subscription in time to receive the value message(s) for C, then a may have missed one or more responses affecting C during the subscription outage and could infer an erroneous response list from value messages updating names in C. To prevent this, subscription response are included in stored response message sequences, enabling applications to detect any invalid lists.

[17]Systems where this assumption does not hold can use the version of the protocol based on lists of message identifiers instead.

7.3 CacherAppHost$_a$ and CacherAwareSwitch$_i$

We refine the dynamic application and switch I/O Automata of Chapter 6 to become the CacherAppHost and CacheAwareSwitch I/O Automata, respectively.

To model the freshening of value messages, let set VR now comprise, for any update $u \in \mathit{UPDATES}$ and any $r_f \in R$ a unique element $vr = VR(u,\, r_f)$ of VR, such that function *update*: $VR \to \mathit{UPDATES}$ maps vr to u and function *first-resp*: $VR \to (R \cup \{\perp\})$ maps vr to r_f. We let $VR(u)$ now denote $VR(u,\, \perp)$, the value message for u with an empty response list, so *first-resp* yields $\perp$ for the message. An implementation would include a response identifier or digest in value messages instead of actual response messages, but that detail is unimportant for our models.

We define the sets DVQ, $DVR \subseteq Q$ of *diverted value read request messages* and *diverted value messages*, respectively, disjoint with other message subtypes of Q. Note that the latter really are defined as requests, not responses, since they are conflict-checked against responses much like requests. Requests in DVQ do not conflict with any response; in fact they travel in the same direction as responses. Diverted value messages, however, conflict with all responses that they have a name in common with, including subscription responses. Formally:

Definition 7.1 *For all* $dvr \in DVR$, $r \in R$: $\mathit{names}(dvr) \cap \mathit{names}(r) \neq \emptyset \Rightarrow \mathit{conflicts\text{-}inv}(dvr,\, r)$.

Function $\mathit{diverted}(v,\, j)$ maps any pair of a read request or value message v and switch $j \in \mathit{SWITCHES}$ to the message dv in DVR or DVQ, respectively, such that functions *message*: $DVR \to VR$ and *message*: $DVQ \to VQ$ map dv to d and functions *divert-switch*: $DVR \to \mathit{SWITCHES}$ and *divert-switch*: $DVQ \to \mathit{SWITCHES}$, respectively, map dv to j. Also, for any $dv \in DVR \cup DVQ$ let $\mathit{names}(dv) = \mathit{names}(\mathit{message}(dv))$.

CacherAppHost$_a$

A refinement of DynamicAppHost that serves value messages from its cache on behalf of the servers (or lower-level cacher hosts) hosting the corresponding state. Infers the list of responses that overtake a value message at a diverting switch from a stored sequence of responses in the *synchResp* field. It stores responses that conflict with names for which the application awaits value messages, which is sufficient. The list is inferred by computing the subsequence of stored responses that affects the value message's name set, starting at the first-conflicting response from the value message, if present in the response sequence. We capture this logic in the *resp-list* function as follows:

Definition 7.2 *Let $R*$ denote the set of sequences over R. For any $\rho \in R*$ and any $r_f \in (R \cup \{\perp\})$, let resp-list($\rho$, r_f) denote:*

- *λ if $r_f = \perp$, or else:*
- *$\perp$ if $r_f \notin \rho$, or else:*
- *the suffix of ρ that begins with r_f.*

CacherAppHost$_a$ also adds any subscription response messages received to its response sequence. A value message vr with a first-conflicting response r_f can only be used if a is storing r_f and has been continuously subscribed to each name in *names*(vr) since receiving r_f, so there are no subscription response messages following r_f in *synchResp* that affect a name in vr. Formally:

Definition 7.3 *For any $\rho \in R*$, $N \subseteq NAMES$ and $r_f \in R \cup \{\perp\}$, let predicate usable-list(ρ, N, r_f) be satisfied exactly if $r_f = \perp$ or else if $r_f \in \rho \wedge \nexists sr \in$ resp-list(ρ, r_f) : $(sr \in SR \wedge sub(sr) \cap N \neq \emptyset)$.*

Additional State:

requested$_s$: set of names being requested after a diversion by switch s, initially $\emptyset$.
awaiting: set of names awaiting a value response message, initially $\emptyset$.
synchResp: sequence of responses overlapping with *awaiting*, initially λ.

Modified Input actions:

```
receive($q \in Q \setminus SQ \setminus VQ \setminus DVQ \setminus DVR$)$_{c,a}$
    replaces receive($q \in Q \setminus SQ \setminus VQ$)$_{c,a}$, but behaves identically to it.

receive($r \in R \setminus SR \setminus VR$)$_{c,a}$
Effect:
    $cache' = cache \oplus TRANS_b\ (cache,\ r)$, where $b = sender(r)$
    $outQueue' = outQueue \cdot ack(r)$
    // if this response affects a name we're waiting for
    if $names(r) \cap awaiting \neq \emptyset$
        // add to synch response sequence
        $synchResp' = synchResp \cdot r$

receive($vr \in VR$)$_{c,a}$
Effect:
    // set $N_{vr}$ contains the names now becoming synched
    let $N = names(vr)$, $N_{vr} = (N \cap awaiting) \setminus synched$, $r_f = first\text{-}resp(vr)$ in
        // if we can safely infer the responses list
        if $usable\text{-}list(synchResp,\ N_{vr},\ r_f)$ then
            // these are synched now
            $synched' = synched \cup N_{vr}$
            // no longer awaiting these
            $awaiting' = awaiting \setminus N$
            let $s_{vr} = TRANS_b(cache \oplus vr,\ resp\text{-}list(synchResp \mid N_{vr}, r_f))$ in
                // apply changes to $N_{vr}$ only
                $cache' = cache \oplus (s_{vr} \mid N_{vr})$
    // $remove$ is the set of responses not containing any names that we're waiting for
    let $remove = \{\ r \in synchResp\ \setminus SR :\ names(r) \cap awaiting' = \emptyset\ \}$ in
        let $removed\text{-}resp = synchResp \setminus remove$ in
            $synchResp'$ = longest suffix of $removed\text{-}resp$ beginning with a response $r \notin SR$

receive($sr \in SR$)$_{c,a}$
Effect:
    $outQueue' = outQueue \cdot ack(sr)$
    $\forall n \in NAMES$: $inNames'(n) =$
        $inNames(n)$, if $n \notin \text{names}(sr)$,
        $sub$, if $n \in sub(sr)$,
        $pend$, if $n \in pend(sr)$,
        $unsub$, if $n \in unsub(sr)$.
    $synched' = synched \setminus unsub(sr)$
    $//awaiting \subseteq inSub$
    $awaiting' = awaiting \setminus unsub(sr)$
    $//requested_s \subseteq synched$
    for each $s \in SWITCHES$
        $requested'_s = requested_s \setminus unsub(sr)$
    // remember names we're becoming subscribed to, for *usable-list* testing
    if $sub(sr) \neq \emptyset$
        $synchResp' = synchResp \cdot sr$
```

Additional Input actions:

```
receive($dvq \in DVQ$)$_{c,a}$
*Effect:*
    // note that we should send value messages for the names
    let $s = divert\text{-}switch(dvq)$ in
        $requested'_s = requested_s \cup (\text{names}(dvq) \cap synched)$
```

Additional Internal actions:

```
request($U \subseteq NAMES$)$_a$
*Precondition:*
    // request subscribed names we're not already requesting
    $U \subseteq inSub \setminus synched \setminus awaiting$
*Effect:*
    // send a value request message
    $outQueue' = outQueue \cdot VQ(U)$
    // note that we're waiting for value messages
```

$awaiting' = awaiting \cup U$

resend$(vq \in VQ)_a$
Precondition:
$names(vq) \subseteq awaiting$
Effect:
// retransmit a value request
$outQueue' = outQueue \cdot vq$

value$(U \subseteq NAMES)_a$
Precondition:
$U \subseteq requested_s$, for some $s \in SWITCHES$, and $U \neq \emptyset$
Effect:
// send a diverted value message, using values from our cache
$outQueue' = outQueue \cdot diverted(VR(a.cache \, \Delta \, U), \, s)$
// this set of names is done
$requested_s{}' = requested_s \setminus U$

The receive$(r \in R \backslash SR \backslash VR \backslash DVR)_{c,a}$ action is identical to the one in DynamicAppHost$_a$, except that if the response affects a name that the application is requesting, the response is appended to the *synchResp* sequence.

The receive$(vr \in VR)_{c,a}$ action applies a value message to the application's cache, the same as in DynamicAppHost$_a$, except it now applies the (possibly empty) sequence of responses listed in a value message after applying the value message's update. If the response list cannot be used, the value message is ignored. This can happen, for example, if the application is receiving a value message it did not request or if it has recently lost and re-established a subscription to some of the message's names. In any case, the action removes the message names from the *awaiting* set. It also garbage collects from *synchResp* responses that do not affect any names the application is waiting for. It removes subscription responses only if no request response precedes them, that is: it removes the prefix of the sequence that consists only of subscription responses. They can be garbage collected because they will never be needed for a *usable-list* test.

The receive($sr \in SR)_{c,a}$ action is the same as in *DynamicAppHost* except it also removes unsubscribed names from all $requested_s$ sets, to maintain the invariant that $requested_s \subseteq synched$, which ensure that the application never transmits stale value messages. It also removes unsubscribed names from *awaiting* to maintain the *awaiting* $\subseteq$ *inSub* invariant.

The receive($dvq \in DVQ)_{c,a}$ action receives a diverted read request. It simply notes the set of names requested, that is: the subset of the requested names it has synched. The value action will gradually send out value messages for that set until each name has been handled by a value message. Atomically enqueuing the response(s) here is impractical for requests for large name sets, and is not needed for correctness.

The request($U \subseteq NAMES)_a$ action non-deterministically sends a value request message for some non-synched subset of its subscription, to get these names synched. It adds the set of names requested to the *awaiting* set. The presence of a name in *awaiting* results in each response or cancellation response affecting that name being added to the *synchResp* response sequence.

The resend($vq \in VQ)_a$ action non-deterministically sends a value request message for some set of names for which the switch is waiting to receive value messages. This action captures the time-out and retransmission mechanism an application implementation would use to overcome loss of value request messages.

The value($vr \in VR)_a$ action non-deterministically chooses some subset of the names that have been requested by diverted read requests. It sends back a diverted value message for the subset, while removing it from the set of pending requested names.

CacheAwareSwitch$_i$

A refinement of *DynamicSwitch* that can divert read requests off the path to their server(s) and onto the path towards a cacher host or hosts. Since we leave out the details of how cachers are added or removed from consideration, there are no actions that modify $cachers_c$ fields. We simply assume that the fields correctly identify names cached by cacher hosts, throughout any execution.

When forwarding a diverted value message, CacheAwareSwitch sets the *first-resp* of the message to the first conflicting response buffered on the switch or else leaves it unchanged, if the message already has a *first-resp* or if the switch is not buffering a conflicting response. We capture this formally as follows, recalling that $R*$ denotes the set of all sequences over R:

Definition 7.4 *Let first-resp-for be the function:* $VR \times R* \rightarrow (R \cup \{\perp\})$ *such that:*

- *first-resp-for*$(vr,\ \rho)$ = *first-resp*(vr) *if first-resp*$(vr) \neq \perp \vee \rho = \lambda$*, or else:*
- *first-resp-for*$(vr,\ r \cdot \rho)$ $= r$

Additional State:

for each $c \in$ *CHANNELS*$_i$:
    $cachers_c$: the subset of $outNames_c$ that one or more cachers serve

Modified Input Actions:

receive$(q \in Q \setminus SQ \setminus VQ \setminus DVQ \setminus DVR)_{c,i}$
    replaces receive$(q \in Q \setminus SQ \setminus VQ)_{c,i}$, but behaves identically to it.

```
receive$(vq \in VQ)_{c,i}$
*Effect:*
    // may divert some names of $vq$ but forwards others normally
    let $d_1$, $d_2$, ..., $d_n$ represent some ordering of $CHANNELS_i$ in
        let $\{ N_0, N_{d_1}, ..., N_{d_n}\}$ represent a partitioning of $names(vq)$
        such that for each $d_i \in [1,\ n] : N_{d_i} \subseteq cachers_{d_i} \cap outSub_{d_i}$, in
            for each $d_k \in \{d_1, d_2, ..., d_n\}$ where $N_{d_k} \neq \emptyset$
                // forward as diverted by $i$
                $outQueue'_{d_k} = outQueue_{d_k} \cdot diverted(VQ(N_{d_k}),i)$
        for each $d \in CHANNELS_i$ such that $N_0 \cap nHop_i(d) \neq \emptyset$
            // forward rest normally, non-diverted
            $outQueue'_d = outQueue_d \cdot VQ(N_0) \cap nHop_i(d)$
```

Additional Input Actions:

```
receive($dvq \in DVQ)_{c,i}$
*Effect:*
    // forward the names of $dvq$ to arbitrary caches for the names
    let $d_1$, $d_2$, ..., $d_n$ represent some ordering of $CHANNELS_i$ in
        let { $N_0$, $N_{d_1}$, ...,$N_{d_n}$} represent a partitioning of $names(dvq)$
        such that for each $d_i \in [1, n] : N_{d_i} \subseteq caches_{d_i} \cap outSub_{d_i}$, in
            for each $d_k \in \{d_1, d_2, ..., d_n\}$ where $N_{d_k} \neq \emptyset$
                // forward with original switch tag
                $outQueue'_{d_k} = outQueue_{d_k} \cdot diverted(VQ(N_{d_k}), divert\text{-}switch(dvq))$

receive($dvr \in DVR)_{c,i}$
*Effect:*
    // forward diverted value messages back towards diverting switch
    let $vr = message(dvr)$ in
        // if the value messages doesn't conflict with a cancellation response
        if $\nexists sr \in responses_c \cap SR$ where $conflicts\text{-}inv(q, sr)$
            // $\rho_c$ has all conflicting responses, in received order
            let $\rho_c$ be the sequence of elements of $\{ r \in responses_c \mid conflicts(vr, r) \}$
            in the order they were added to $outQueue_c$ in
                // $vr_\rho$ is new value message with $first\text{-}resp\ updated$
                let $vr_\rho = VR(update(vr), first\text{-}resp\text{-}for(vr, \rho_c))$ in
                    // if this switch diverted the request
                    if $divert\text{-}switch(dvr) = i$
                        // forward it as a normal value message
                        for each $d$ such that $names(vr_\rho) \cap outSub_d \neq \emptyset$
                            $outQueue'_d = outQueue_d \cdot vr_\rho$
                    else
                        // forward as still diverted
                        let $d$ be the channel such that names$(vr_\rho) \subseteq nHop_i(d)$ in
                            $outQueue'_d = outQueue_d \cdot diverted(vr_\rho, divert\text{-}switch(vr_\rho))$
```

The receive($vq \in VQ)_{c,i}$ receives a read request message and forwards different parts of it to the appropriate server(s) and / or cacher host(s). The action's non-determinism stems from the way the action partitions the names of the request, with the only constraint that the set of names diverted onto a channel must be a non-empty subset of the cacher-subset of that channel's subscription. Note that a request's name is only diverted to a cacher for that name if the corresponding channel is subscribed to the name, otherwise the name is not synched at any cacher reachable through that channel. Partition N_0 represents the part of the request that is not diverted. As a practical matter, given a choice of cachers, an implementation might perform static or dynamic load balancing to efficiently distributed the effort of serving read requests. Also, an implementation might forward messages unchanged to all relevant destinations, rather than incur the processing cost of splitting up messages. Our model shows the name sets that must be forwarded at a minimum.

The receive($dvq \in DVQ)_{c,i}$ action receives a diverted read request, that is: a request already diverted by another switch. It forwards it the same way as receive($vq \in VQ)_{c,i}$ does when diverting original read requests.

The receive($dvr \in DVR)_{c,i}$ action receives a diverted value message. If the message is in response to a request diverted by this switch, then the action converts it to an ordinary value message and forwards it as if it had been received directly from the original server. If not, the message is still traveling back towards its diverting switch so it is forwarded essentially as if it were a request. Note that the switch never has to split the message in the latter case, since all its names are bound for the same diverting switch.

The key to the correctness of the scalable caching scheme is that the action also conflict-checks the value messages against the responses buffered in $responses_c$. Rather than dropping a conflicting value message, it tags the message with the first (earliest) conflicting response, if it doesn't have such a tag already. The receive($vr \in VR)_{c,i}$ action in *CacherAppHost* uses this information to infer the list of all responses that the value message encountered on its way to its diverting switch, which as our proofs will show, is the same as the list of responses that overtook the value message at the diverting switch.

The action drops the diverted value message if it conflicts with a subscription response. Since the subscription to a name in *dvr* has been changed (the name has been canceled, most likely) *dvr* may miss a conflict with a response and the responses list inferred for it by its receiver may no longer correspond to the responses overtaking it at the diverting switch.

7.4 Properties and Proofs

Let *SDCS* be the I/O automaton composed of a CacheAwareSwitch$_i$ automaton for each switch $s_i \in$ *NODES,* a CacherAppHost$_a$ automaton for each application host $a \in$ *APPS, a* DynamicDataServerHost$_b$ automaton for each data server host $b \in$ *DATAS* and a Channel$_c$ automaton for each channel $c = (i, j) \in$ *CHANNELS*.

We show that execution correctness Theorem 6.1 for Dynamic Atomic Cache Systems still holds in *SDCS* even though applications may synchronize their caches using other caches. Lemmas 6.1 through 6.7 and lemmas 6.9 and 6.10 of Chapter 6 are easily shown to still hold in *SDCS*. Lemma 6.8 applies in SDCS only to value messages sent directly by original servers. We will show that it also holds for messages sent by cachers, and thus that Theorem 6.1 still holds.

Lemmas 6.2 and 6.3 hold in SDCS because CacherAwareSwitch$_i$ and CacherAppHost$_a$ handle subscriptions, synchronization and request/response processing exactly the same as DynamicSwitch$_i$ and DynamicAppHost$_a$, respectively. CacherAppHost has slightly different receive($vr \in VR$)$_{c,i}$ and receive($sr \in SR$)$_{c,i}$ actions, but their behavior with respect to *inSub* and *synched* sets is the same. Lemma 6.4 concerns response order along a particular path between two switches and still holds, since the behavior of CacherAwareSwitch$_i$ for responses is the same as that of DynamicSwitch$_i$. Lemma 6.5 concerns subscription and publishing intervals and holds for the same reason. Similarly, Lemmas 6.6 and 6.7 concern response messages and still hold for CacherAwareSwitch$_i$.

Lemma 6.8 holds if all value messages received are sent by original servers and have not been diverted. CacherAwareSwitch$_i$ behaves the same as DynamicSwitch$_i$ for those read requests it doesn't divert and CacherAppHost$_a$ behaves the same as DynamicAppHost$_a$ for non-diverted value response messages, since they have an empty responses list. However, Lemma 6.8 does not hold for value messages sent by cacher hosts, as message diversion can disrupt the consistent total order that exists between value messages and responses sent from the same original server. What we show is that when a value message vr from a cacher is received at a host, prepending vr's list of conflicting responses to the sequence of responses received after vr yields a consistent state. Lemma 7.2 shows that Lemma 6.9 holds for CacherAppHosts, and we observe that Lemma 6.10 still holds as it only concerns DynamicDataServerHost.

We extend the $\rightarrow_E$ causes relation, with each send$(dvr' \in DVR)_{d,j}$ event caused by a receive$(dvr \in DVR)_{c,j}$ event, where d and c are channels incident to a switch $j \in SWITCHES$. Also, add to $\rightarrow_E$ each send$(vr \in VR)_{d,j}$ event caused by a receive$(dvr \in DVR)_{c,j}$ event, i.e. when a switch j receives back a diverted value message sent in response to a value request that j diverted or forwarded onto channel c.

As mentioned before, we do not model how cachers add themselves to the $cachers_c$ fields of switches, but we will assume that a cacher's value request is never diverted back to the cacher. This should be straightforward to ensure in practice, for example if a cacher never register itself as a cacher for a name until after it is synched with that name. But we begin with two quick lemmas.

Lemma 7.1 *(Requested implies Synched): In any state of CacherAppHost$_a$: $a.requested_s \subseteq a.synched$, for each $s \in SWITCHES$.*

Proof: Both fields are empty in the start state. Inspecting the actions of CacherAppHost$_a$, we readily see that all changes to *a.requested* fields and *a.synched* preserve the invariant □

Lemma 7.2 *(CacherAppHost$_i$ preserves ACK Well-Formedness).*

Proof: Immediate, since the only action in CacherAppHost_i that enqueues $ack(r)$ is $\text{receive}(r)_{c,i}$ □

We now show a key lemma, saying that at the point when a diverted value message dvr is received at its diverting switch i, the application of dvr and its response list produces the state for $names(dvr)$ that existed on the original server b immediately after b enqueued the latest response affecting $names(dvr)$ that has been received at i. Hence, dvr can be safely "inserted" into the sequence of messages bound for a subscriber a, since the state it produces will match up seamlessly with any later responses from b.

This Lemma holds only if the cacher host generating dvr receives its value messages directly from original server hosts. We later extend our results to multi-level caches, so a cacher host can receive value messages from another cacher, as well as from original servers.

We make the assumption that a cacher has exactly one network path connecting it to any particular switch diverting value read requests to it. This assumption can be relaxed, since (value) responses may be forwarded on multiple paths while preserving Conflict Locality, but the assumption simplifies our proofs. However, the proofs are somewhat complicated by the fact that a host may cache state from multiple servers, and the name sets of value requests and responses may span multiple server

Figure 7.2 illustrates the main entities of Lemma 7.3, and shows an example of the correspondence between response prefixes at servers. Server b has sent out a sequence of response messages ending with $abcdef$ (so each letter is one response). Diverting switch i and cacher c have received prefixes of that response sequence, i.e. ending with $abcd$ and ab, respectively.

We introduce a new operator for sequences of responses, that filters out those responses that do not affect a particular name set.

Definition 7.5 *For any sequence of responses $\rho \in R*$ and $N \subseteq NAMES$, let $\rho \lhd N$ denote the maximal subsequence of ρ such that for every $r \in \rho \lhd N$: $names(r) \cap N \neq \emptyset$.*

We break out a part of the lemma's premise into a separate predicate.

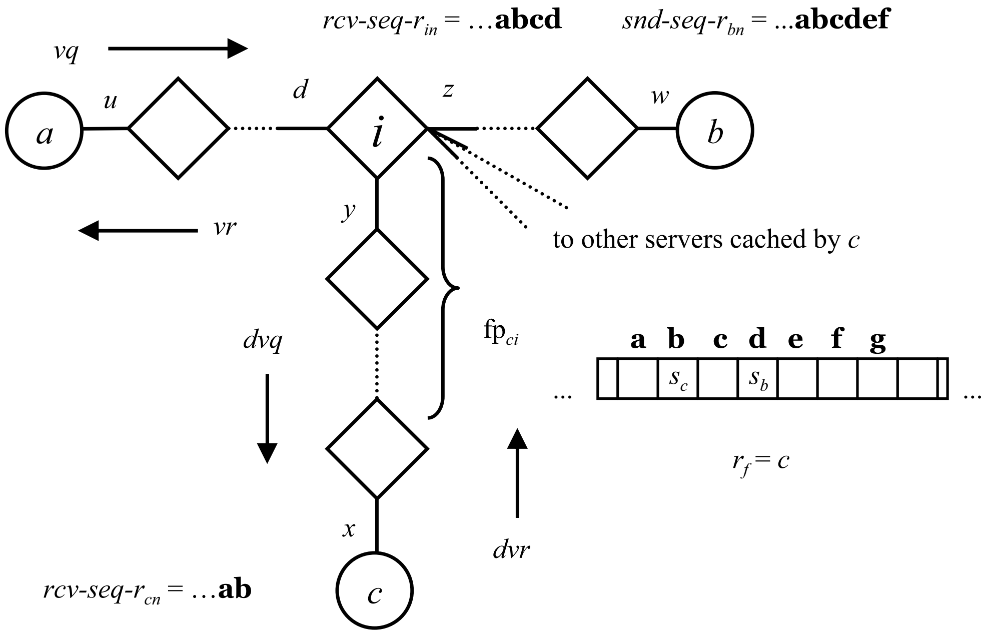


Figure 7.2: Value message diversion, and naming for Lemma 7.3

Definition 7.6 *For any execution $X \in$ execs(SDCS), cacher host $c \in APPS$ and server $b \in DATAS$, let cache-cond(c,b,X) be true exactly if cacher c only receives value messages from VR containing a name from $NAMES_b$ directly from b in X.*

Lemma 7.3 *(Freshness of diverted value messages base case): Let X be any execution fragment of SDCS with final event e_{idv} = receive($dvr \in DVR$)$_{y,i}$ at a switch i = divert-switch(dvr) caused by a send(dvr)$_{x,c}$ event at a cacher host $c \in APPS$, with $y \in CHANNELS_i$ and x incident to c. Let vr = message(dvr). Let N = names(vr) and let $N_b = N \cap NAMES_b$, for a server $b \in DATAS$ where $N_b \neq \emptyset$. Let responses$_{dvr}$ = resp-list(rcv-seq-$r_{iN_b}(X)$, first-resp(vr)). Let s_b be the value of b.state immediately after the e_r event enqueuing the response r leading to the send(r)$_{w,b}$ event causing the latest event e_{ir} = receive($r \in R$)$_{z,i}$ in X, for channels w incident to b and $z \in CHANNELS_i$, or let s_b = start$_b$ if X has no such event. Then for any $s \in STATES$ and $d \in CHANNELS_i$, $e_{idv} \rightarrow_E$ send($vr \in VR$)$_{d,i}$ $\wedge$ cache-cond(c, b, X) $\Rightarrow$ TRANS$_b$($s \oplus vr$, responses$_{dvr}$) $\mid N_b = s_b \mid N_b$.*

Proof: Let e_{idv} be an event fulfilling the lemma's premise where $e_{idv} \rightarrow_E$ send($vr \in VR$)$_{d,i}$. Let e_c be the value($N \subseteq NAMES$)$_c$ event at cacher host c that enqueues the

dvr message leading to the send$(dvr)_{x,c}$ event causing e_{idv}. From the precondition of the value$(N \subseteq NAMES)_c$ action we see that $N_b \subseteq N \subseteq c.synched$ at e_c, since $c.requested_i \subseteq c.synched$ by Lemma 7.1. Let s_c be the value of $c.cache$ at e_c, so $(s \oplus vr) \mid N_b = s_c \mid N_b$ for any $s \in STATES$.

Since *cache-cond*(c,b,X), we have by Lemma 6.8 of Chapter 6 that $s_c \mid N_b = b.state \mid N_b$ in the earlier state t_b of X immediately after the e_b event enqueuing the latest response from b that has been received by c.

Let fp_{ci} be the prefix s_1, s_2, ..., s_n of the forwarding path from c to b, up to and including switch s_i, so $s_i = s_n$. By our assumption, each response forwarded from i to c is forwarded via fp_{ci}, as well as each diverted value message sent from c to i. Since $N \subseteq c.synched$ at e_c and dvr is not dropped due to a conflict with a cancellation response, no response conflicting with dvr is purged on the relevant part of path fp_{ci} (ahead of dvr on fp_{ci}) during the interval when dvr travels along fp_{ci} to i. Inspecting the case in receive$(dvr \in DVR)_{c,j}$ action at each $j \in fp_{ci}$, where dvr is not dropped, we see that j detects each conflict between dvr and a response in $responses_c$ exactly like the receive$(q \in Q \setminus SQ \setminus VQ \setminus DVQ \setminus DVR)_{c,j}$ action does for requests (recall that $DVR \subseteq Q$). The crucial difference is that while j may alter dvr due to a conflict, dvr is not dropped. By the reasoning in Theorems 5.1 and 5.2, each response r received at i in X that conflicts with q for which c has yet to send an ACK is detected as a conflict with dvr. In particular, the first response r_f detected as conflicting with dvr, if there is such a r_f, is the first response updating any $n \in N$ that is received after e_c at c, since c applies every prior response r to $c.cache$ while enqueueing an ACK, placing the $ack(r)$ ahead of dvr in $c.outQueue$.

In the example of Figure 7.2, $r_f = c$ and c has received responses ...*ab* while i has received responses ...*abcd*. Hence, cached state s_c is the same as $b.state$ immediately after server b enqueues response b.

If there is a first-conflicting response r_f, it is received at switch i before it is received at c. Let X_{r_f} denote the suffix of X beginning with the receive$(r_f)_{i,z}$ event at i and let *rcv-seq-fp*$_{ic}$ denote *rcv-seq-r*$_{iN_b}(X_{r_f})$, that is: each response affecting N_b that has been received at i in X but has not been received by c at e_c. If there is no such r_f, then c has received every response in *rcv-seq-r*$_{iN}(X)$ at e_c and we define *rcv-seq-fp*$_{ic}$ to be the empty sequence λ. Observe that immediately after e_{idv}, dvr has conflicted with every response in *rcv-seq-fp*$_{ic}$. Furthermore, since *dvr* is not dropped, there are no cancellation responses in *rcv-seq-fp*$_{ic}$ affecting N. Hence, $N_b \subseteq N \subseteq i.inSub_z$ at e_{idv} and for each $n \in N_b$, *sub-interval*$(i, n, t_X) = I_{in}$ exists by Lemma 6.5 and contains X_{r_f}, where t_X is the final state of X. Therefore, by Lemma 6.6 *rcv-seq-r*$_{in}(I_{in}) \preceq$ *snd-seq-r*$_{bn}(I_{bn})$, where $I_{bn} =$ *pub-interval*(b, i, n, t_X). This establishes that i has received a continuous sequence of responses from b, that is: *rcv-seq-fp*$_{ic} \preceq$ *snd-seq-r*$_{bN_b}(I_{N_b})$, where I_{N_b} is any suffix of X that includes I_{bn} for each $n \in N_b$.

We claim that *rcv-seq-fp*$_{ci}$ = *responses*$_{dvr}$, that is: the response list computed for *dvr* at e_{dvi} is precisely the sequence of conflicting responses that have been received at i immediately after e_{idv} but have not been received by c at e_c, as c enqueues *dvr*. In the case where *dvr* does not have any conflicts, by definition *rcv-seq-fp*$_{ci} = \lambda$. Then, since *dvr* never conflicts, *first-resp*$(vr) = \perp$, so *resp-list*$(\rho,$ *first-resp*$(vr))$ = *responses*$_{dvr}$ = λ, for any response sequence ρ. On the other hand, if *dvr* does have conflicts then the first one is detected in an event $e_j = \text{receive}(dvr \in DVR)_{(j-1,j),j}$ on some switch $s_j \in$ *fp*$_{ci}$, where $1 \leq j \leq n$. Since *first-resp*$(vr) = \perp$ at e_j, event e_j (via *first-resp-for*) assigns response r_f as the *first-resp* of the modified diverted value message forwarded, where r_f is the earliest received response among those in *j.responses*$_{(j-1,j)}$ that conflict with *dvr*. If *dvr* (or more precisely: a message caused by *dvr*) is detected as a conflict at a switch s_k where $j < k \leq n$, the corresponding $\text{receive}(dvr \in DVR)_{c,k}$ event does not change *first-resp*(vr), by the definition of *first-resp-for*. At e_{idv}, therefore, *first-resp*(vr) is the first response from *rcv-seq-r*$_{iN}(X)$ that has not been received by c at e_c, which is r_f. Hence *responses*$_{dvr}$ and *rcv-seq-fp*$_{ic}$ are both the maximal subsequences of the suffix of *rcv-seq-r*$_{iN}(X)$ starting with r_f consisting of responses that affect names in N_b, and it follows that *rcv-seq-fp*$_{ic}$ = *responses*$_{dvr}$.

We've established that *responses*$_{dvr}$ = *rcv-seq-fp*$_{ci} \preceq$ *snd-seq-r*$_{bN_b}(I_{N_b})$. In the example of Figure 7.2, *responses*$_{dvr}$ is the sequence *cd* of responses enqueued by i but not yet received at c, and s_b is the state immediately after response d was enqueued, since that's the latest response r of b received at i. If *responses*$_{dvr} = \lambda$ (so there is no first-conflicting response r_f) then $s_b \mid N_b = c.cache \mid N_b = s_c \mid N_b$, and for any $s \in$ *STATES*, *TRANS*$_b(s \oplus vr, \lambda) \mid N_b = s_b \mid N_b$ and the theorem holds. If *responses*$_{dvr}$ is not empty, observe that receive event $e_{ir} = \text{receive}(r)_{z,i}$ is the event providing the last message r of *responses*$_{dvr}$. Also, the first message r_f of *responses*$_{dvr}$ is the first response affecting a name in N_b after e_c, when c enqueues *dvr*. Since we've shown hat $\textit{responses}_{dvr}$ is a (contiguous) prefix of $\textit{snd-seq-r}_{bN_b}(I_{N_b})$, $\textit{responses}_{dvr}$ is precisely the sequence r_f, r_2, r_3,..., r of responses that transforms $s_c \mid N_b = c.cache \mid N_b$ at e_c to $s_b \mid N_b$, the value of $b.state \mid N_b$ immediately after r was enqueued. In conclusion, in the case where $\textit{responses}_{dvr} \neq \lambda$, $\textit{TRANS}_b(s \oplus vr,\ \textit{responses}_{dvr}) \mid N_b = s_b \mid N_b$, for any $s \in \textit{STATES}$ $\Box$

We now show that if the $a.synchResp$ sequence of CacherAppHost$_a$ is usable with respect to a name set N_b and a first-conflict response r_f, then it contains a (contiguous) sequence containing each response from b that affect N_b, at least from r_f on. The Lemma is generalized to talk about the responses sent by any node on the forwarding path to b. The key idea is to show that in these circumstances, a has been subscribed to each name in N at least since receiving r_f.

Lemma 7.4 *(completeness of responses sequences) For any state of any $X \in \textit{execs}(SDCS)$, $a \in APPS$, $b \in DATAS$, name set $N_b \subseteq a.awaiting \cap NAMES_b$, $r_f \in (R \cup \{\perp\})$ and node $o \in \textit{fp}_{ab} \cdot b$: usable-list$(a.synchResp, N_b,\ r_f) \Rightarrow$ there exist a suffix I_{oN_b} of X such that resp-list$(a.synchResp \lhd N_b,\ r_f)$ = rcv-seq-r$_{aN_b}(I_{aN_b}) \preceq$ snd-seq-r$_{oN_b}(I_{oN_b})$, where I_{aN_b} is the suffix of X beginning immediately before event receive$(r_f)_{c,a}$ in X.*

Proof: As a preliminary, we claim that in every state of X, $a.awaiting \subseteq a.inSub$. This trivially holds in the initial state where $a.awaiting = a.inSub = \emptyset$, and it is easily verified by inspecting the actions of CacherAppHost$_a$ that modify $a.awaiting$ or $a.inSub$ that this invariant is preserved.

Let $N_b \subseteq a.awaiting \cap NAMES_b$ and $r_f \in R$ be respectively a name set and response fulfilling the lemma's premise, so usable-list$(a.synchResp, N_b,\ r_f)$ holds. If $r_f = \perp$ then resp-list$(a.synchResp,\ r_f) = \lambda$ and the lemma trivially holds. Otherwise, $r_f \in a.synchResp$ and there must be an event e_{r_f} = receive$(r_f)_{c,a}$ in X, where c is the channel incident to a, since that's the only event that can add r_f to $a.synchResp$. Since $N_b \subseteq a.awaiting \subseteq a.inSub$, by Lemma 6.6 of Chapter 6 we have for each $n \in N_b$ that rcv-seq-r$_{an}(I_{an}) \preceq$ snd-seq-r$_{bn}(I_{on})$, with I_{an} and I_{on} the corresponding subscription and publishing intervals of n. We claim that a has been contiguously subscribed to each name of N_b since before receiving r_f. More precisely, we claim that suffix I_{aNb} is contained in I_{an}, for each $n \in N_b$. Assume, for contradiction, that I_{aNb} is not contained by I_{an}, for some $n \in N_b$. Then the last receive$(sr \in SR)_{c,a}$ event in

X with $n \in sub(sr)$ occurs after e_{r_f}, and sr follows r_f in *a.synchResp* (and therefore cannot have been garbage collected in X). But then $\neg$*usable-list*(*a.synchResp*, N_b, r_f), which is a contradiction.

Hence, suffix I_{aNb} must be contained by each of the I_{an} intervals, and since for each of them we have *rcv-seq-r*$_{an}(I_{an}) \preceq$ *snd-seq-r*$_{on}(I_{on})$ it follows that there exists a suffix I_{oN_b} of X (contained by each I_{on}) such that *rcv-seq-r*$_{aN_b}(I_{aN_b}) \preceq$ *snd-seq-r*$_{oN_b}(I_{oN_b})$. Note that I_{oN_b} begins with the send$(r_f)_{o,d}$ event causing e_{r_f}, where d is some channel incident to o. Since a appends every response r affecting a name in $a.awaiting$ $\supseteq N_b$ to *a.synchResp*, and since *resp-list*(*a.synchResp* $\lhd N_b$, r_f) contains every response in *a.synchResp* after and including r_f that affects N_b, it follows that *resp-list*(*a.synchResp* $\lhd N_b$, r_f) $=$ *rcv-seq-r*$_{aB}(I_{aN_b}) \preceq$ *snd-seq-r*$_{on}(I_{oN_b})$ $\Box$

We can now show that Lemma 6.8 of Chapter 6 still holds in the scalable system, that is: the synched part of an application cache contains a consistent projections of an earlier (or current) server state. The key is to show that the list of responses an application infers from a value message is the same as the sequence of responses the corresponding diverted value message conflicted with, which are exactly the responses that the application missed as they overtook the value message at the diverting switch. Applying the response list therefore recreates the effect of these omitted responses and brings the value message's names to the state that existed on the original server just after it sent the last omitted response. We limit ourself, for the time being, to the case where each cacher receives its value messages directly from servers, and not from other caches.

We break out a part of the lemma's premise into a separate predicate.

Definition 7.7 *For any execution $X \in$ execs(SDCS), application host $a \in APPS$ and server $b \in DATAS$, let* app-cond(a,b,X) *be true exactly if a only receives value messages for names in NAMES$_b$ directly from b or from a cacher host c only if c receives its value messages for names in NAMES$_b$ directly from b in X., that is: if* cache-cond(c, b, X) *holds.*

Lemma 7.5 *(Scalable dynamic cache synchronization base case): For any state t_a of any $X \in$ execs(SDCS), $a \in APPS$ and $b \in DATAS$, let synched$_b$ $=$ a.synched $\cap$*

$NAMES_b$ in t_a and let t_b be the state after the e_b event enqueueing the latest response from b that has been received by a. Then app-cond$(a, b, X) \Rightarrow a.cache \mid synched_b$ in state $t_a = b.state \mid synched_b$ in state t_b.

Proof: We proceed by induction on the prefixes of X. As argued in the proof of Lemma 6.8, since $a.synched \subseteq a.inSub$ then by Lemma 6.5 each state of X is contained in a subscription interval I_{an} for each $n \in a.synched$ and by Lemma 6.7 $rcv\text{-}seq\text{-}r_{an}(I_{an}) \preceq resp\text{-}seq\text{-}r_n(I_{bn})$, for the corresponding publishing interval I_{bn}. In other words, a is receiving every response affecting any name in *a.synched*, in the order of the corresponding state changes.

As our base case, the invariant trivially holds in the start state of X since *a.synched* is initially empty. We claim that each action e extending a prefix X' into another prefix X'' of X preserves the invariant. Let e_b be the event that enqueues the latest response message sent from b that has been received by a at t_a, the final state of X'. By the inductive assumption, $s_{ab} = a.cache \mid synched_b$ at t_a equals $s_b = b.state \mid synched_b$ in the state t_b immediately after e_b.

1. If $e = \text{receive}(sr \in SR)_{d,a}$ or $e = \text{receive}(r \in R \setminus SR \setminus VR)_{d,a}$ then the invariant holds after e, by the same exact argument as in Lemma 6.8.

2. If $e = \text{receive}(vr \in VR)_{d,a}$ and e is caused by a $\text{send}(vr)_{b,d}$ event in data server $b \in DATAS$ then the invariant holds after e, by the same exact argument as in Lemma 6.8.

3. If $e = \text{receive}(vr \in VR)_{d,a}$ and e is caused by a $\text{send}(vr)_{i,d}$ event at a switch i caused in turn by an event $e_{idv} = \text{receive}(dvr \in DVR)_{y,i}$ event, where $d, y \in CHANNELS_i$, let $N = names(vr)$, $r_f = first\text{-}resp(vr)$, $N_{vr} = (N \cap a.awaiting) \setminus a.synched$ (the names becoming synched) and $N_b = N_{vr} \cap NAMES_b$. If $\neg usable\text{-}list(a.synchResp,\ N_{vr},\ r_f)$, then e modifies neither *a.cache* nor *a.synched* and the invariant holds with the same state s_b as before. Otherwise, let $responses_{dvr} = resp\text{-}list(rcv\text{-}seq\text{-}r_{iN_b}(X_{idv}), r_f)$, where X_{idv} is the prefix of X ending with the e_{idv} event that enqueues vr on $i.outQueue_d$, towards a.

   Since *app-cond*(a,b,X), the cacher c sending dvr gets its value messages for N_b directly from b so by Lemma 7.3, for any $s \in STATES$, $TRANS_b(s \oplus vr,$

responses$_{dvr}$) $\mid N_b = s_{bi} \mid N_b$, where s_{bi} is the value of *b.state* at the event e_{ri} when b enqueues the latest response r that has been received by i at e_{idv}. Note that r is the last response of *rcv-seq-r*$_{iN_b}(X_{idv})$.

Let *rlist-a* = *resp-list*(*a.synchResp* $\lhd N_b$, r_f). We claim that *responses*$_{dvr}$ = *rlist-a*. We observe that since $N_b \subseteq N_{vr}$, *usable-list*(*a.synchResp*, N_{vr}, r_f) implies *usable-list*(*a.synchResp*, N_b, r_f), by the definition of *usable-list*. Since furthermore $N_b \subseteq$ *a.awaiting* $\cap$ *NAMES*$_b$, by Lemma 7.4 there exist suffixes I_{aN_b} and I_{iN_b} of X such that *rlist-a* = *rcv-seq-r*$_{aN_b}(I_{aN_b}) \preceq$ *snd-seq-r*$_{iN_b}(I_{iN_b})$ where I_{aN_b} begins with receive$(r_f)_{u,a}$, so by Lemma 6.6 we have *rcv-seq-r*$_{aN_b}(I_{aN_b}) \preceq$ *snd-seq-r*$_{iN_b}(I_{iN_b}) \preceq$ *rcv-seq-r*$_{iN_b}(I_{iN_b})$. Since event e receives the vr that is enqueued by e_{idv} at the end of X_{idv}, by Lemma 6.6 each response of *rcv-seq-r*$_{iN_b}(X_{idv})$ has been received at a in X, up to and including response r. Hence, *rlist-a* = *rcv-seq-r*$_{aN_b}(I_{aN_b})$ = *resp-list*($< r_f, r_2, \ldots, r >$, r_f) = *resp-list*(*rcv-seq-r*$_{iN_b}(X_{idv})$,r_f) = *responses*$_{dvr}$, so the claim holds.

Event e applies vr and *rlist-a* = *responses*$_{dvr}$ to each name of N_{vr} (including N_b) in *a.cache,* so we get *a.cache*$' \mid N_b = s_{bi} \mid N_b$. Since we've shown that the latest response from b received at a is r (enqueued by e_{bi}) we have $e_{bi} = e_b$, so $s_{bi} = s_b$ and *a.cache*$' \mid N_b = s_b \mid N_b$. Since e does not modify any names in M = $(NAMES_b \setminus B) \cap$ *synched*$_b$ and since by the inductive assumption *cache.a* $\mid M = s_b \mid M$, we have that *a.cache*$' \mid M = s_b \mid M$. Therefore, *a.cache*$' \mid$ *synched*$_b$ equals s_b = *b.state* $\mid$ *synched*$_b$ and the invariant holds after e.

4. No other events affect the invariant □

Lemma 7.5 shows that an application's execution atomicity is ensured although some of the application's value messages come from one or more cacher hosts, if they in turn receive their value messages directly from the corresponding servers. We now show that the Lemma holds even if the cachers receive value messages from other cachers. We use structural induction on the length of chains of cachers to show that if Lemmas 7.3 and 7.5 hold for cacher chains of length n, they also hold for cacher chains of length $n + 1$. Formally:

Definition 7.8 *The cacher chain length* chain-len$_b$(a) *of an application (cacher) host* $a \in APPS$ *with respect to a server* $b \in DATAS$ *is:*

- *0 if a only receives value messages for names in $NAMES_b$ directly from server b, or more precisely: no cacher host connected to a switch on fp_{ab} sends diverted value responses for names in $NAMES_b$ to a.*
- $1 + \max\{\textit{chain-len}(c) : c \in C\}$*, where C is the set containing b and a non-empty set of cacher hosts from which a can receive value messages for names in $NAMES_b$.*

In other words, $\textit{chain-len}_b(a)$ is the maximum distance of a from b in the directed acyclic graph connecting b to application hosts (including cachers) subscribing to names in $NAMES_b$ and application hosts to other application hosts (including cachers) that cache names from $NAMES_b$. This network is a DAG because cachers for $NAMES_b$ only communicate via switches on the forwarding path to b. Note that Lemma 6.8 of Chapter 6 corresponds to the base case with a chain length of 0 and Lemma 7.5 corresponds to applications with cacher chain length of at most 1.

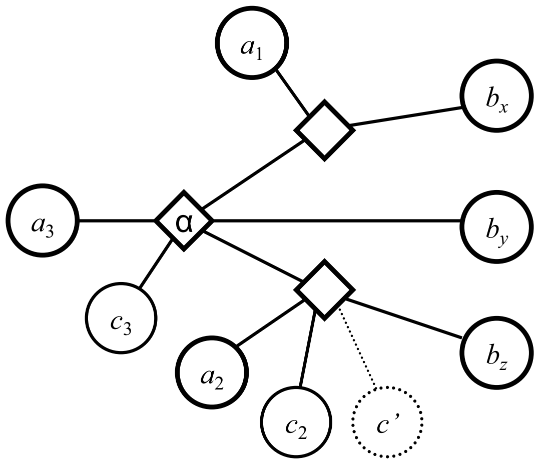


Figure 7.3: Cacher Chain Lengths Illustrated

Figure 7.3 shows an example of a small atomic network, where each host subscribes only to hosts at or below its own level in the hierarchy of switches rooted at switch α. We have $\textit{chain-len}_{b_x}(a_1) = 0$, since there are no cachers to which to divert value requests from a_1 to b_x. We also have $\textit{chain-len}_{b_z}(c_2) = 0$, since value requests from c_2 to b_x are never diverted back to c_2. Note that if there were another cacher host

c' attached to the switch and the set of names served by c_2 and c' overlapped, then their chain-lengths would be 1, as each could serve read requests from the other. We have *chain-len*$_{b_z}(a_2)$ = 1, since the value requests of a_2 may get diverted to cacher c_2, whose chain length is 0. We have *chain-len*$_{b_z}(c_3)$= *chain-len*$_{b_z}(c_2) + 1 = 1$, since c_3 may use lower-level cacher c_2 (by contrast, *chain-len*$_{b_x}(c_3)$ = *chain-len*$_{b_y}(c_3) = 0$) Finally, we have *chain-len*$_{b_z}(a_3)$= *chain-len*$_{b_z}(c_3) + 1 = 2$, and we have *chain-len*$_{b_x}(a_3)$ = *chain-len*$_{b_y}(a_3) = 1$.

Lemma 7.6 *(Scalable dynamic cache synchronization): For any state t_a of any $X \in$ execs(SDCS), $a \in$ APPS and $b \in$ DATAS, let synched$_b$ = a.synched $\cap$ NAMES$_b$ in t_a and let t_b be the state after the e_b event enqueueing the latest response from b that has been received by a. Then a.cache | synched$_b$ in state t_a = b.state | synched$_b$ in state t_b.*

Proof: Let $C \subseteq$ *HOSTS* be the set containing server b and the set of applications (including cachers) that receive (diverted) value messages for names in *NAMES*$_b$. We proceed by induction on the maximum length of cacher chains of hosts in C to show that the invariant holds for each host in C, including a. As our base cases, Lemma 7.5 and Lemma 7.3 show that the theorem holds for each cacher $c \in C$ where *chain-len*$_b(c)$ = 0 and *chain-len*$_b(c)$ = 1, respectively. As our inductive assumption, the invariant holds for each cacher c' such that *chain-len*$_b(c') \leq n$, for some $n \geq 1$. Let c be any application host such that *chain-len*$_b(c) = n + 1$. We observe that since *chain-len*$_b(c)$ = $n + 1$, it follows by the definition of c and *chain-len*$_b$ that any host c' from which c receives value messages has *chain-len*$_b(c') \leq n$. We make two claims.

1. Consider Lemma 7.3, with the new definition that *cache-cond*(c, b, X) is true exactly if cacher c only receives value messages for names in *NAMES*$_b$ from a host c' if *chain-len*$_b(c') \leq n$. Call this modified lemma Lemma 7.3n. We claim that this Lemma 7.3n holds for each diverted value message sent by c. The proof is the same as for Lemma 7.3, except instead of invoking Lemma 6.8 we invoke the inductive assumption, that is: we change the wording in the proof to “Since *cache-cond*(c, b, X), we have by the inductive assumption that ... ”. Hence, Lemma 7.3n holds.

2. Consider Lemma 7.5, with the new definition that *app-cond*(a, b, X) is true exactly if a only receives value messages for names in $NAMES_b$ from a host c' if *chain-len*$_b(c') \leq n$. Call this modified lemma Lemma 7.5n. We claim that this modified lemma holds in each state of X, with $a = c$. The proof is the same as for Lemma 7.5, except instead of invoking Lemma 7.3 we invoke the inductive assumption and Lemma 7.3n, that is: we change the wording in the proof to "Since by the inductive assumption *app-cond*(c', b, X) holds for the cacher c' sending dvr, by Lemma 7.3n, $TRANS_b(s\oplus\ vr,\ responses)_{dvr} \mid N_b = \ldots$".

Lemma 7.5n shows that the invariant holds for c, and hence any c with *chain-len*$_b(c)$ $\leq n + 1$, completing the induction □

We can finally prove (for the last time!) a theorem corresponding to Theorem 4.1 of Chapter 4, saying that if a server in a scalable dynamic system executes a request q then the response is among those possible given the dynamic application's cached state at the time the application issued q. Our proof is essentially the same as the one for Theorem 6.1 (and Theorem 4.1), showing that if this were not the case, then the intervening execution of some request that altered the state of the server would have caused a response that conflicted with q, and q would have been dropped.

Theorem 7.1 *(Scalable Dynamic Cache Operation Atomicity): For any execution $X \in$ execs(SDCS), application $a \in$ HOSTS, server $b \in$ DATAS and channels d and c incident to a and b, respectively: send$(q \in Q)_{d,a} \rightarrow$ send$(r_q \in R)_{c,b} \Rightarrow r_q \in$ execute$_b(s_a, q)$, where s_a = a.cache $\mid$ depends(q) at the createRequest event e_q that enqueues q.*

Proof: Assume for contradiction that there exists some execution $X \in execs(SDCS)$ and server $b \in DATAS$ where send$(q)_{d,a} \rightarrow$ send$(r_q \in R)_{c,b}$ but $r_q \notin$ *execute*$_b(s_a, q)$.

By inspecting the createRequest action, we see that *depends*$(q) \subseteq$ *a.synched* at e_q, so by Lemma 7.6, *b.state* $\mid$ *depends*$(q) = s_a$ at some earlier point in X, before the receive$(q)_{c,b}$ event that executes q. Therefore, there must be an event e_r at b preceding receive$(q)_{c,b}$ that assigns to *b.state* a value s_b such that $r_q \notin$ *execute*$_b(s_b, q)$. If we let r be the response enqueued by e_r, then *conflicts-inv*(q, r), by the definition of *conflicts-inv*. Since *a.cache* $\mid$ *depends*$(q) = s_a$ at the createRequest$(q)_a$ event issuing request

q, we have createRequest$(q)_a <_X$ receive$(r)_{d,a}$ and therefore by FIFO send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. This gives us $e_r <_X$ receive$(q)_{c,b} \wedge$ send$(q)_{d,a} <_X$ send$(ack(r))_{d,a}$. Since by Lemma 7.2 a is ACK Well-Formed and by Lemma 6.10 b is Responder Well-Formed, then by Theorem 5.2 we have send$(q)_{d,a} \not\to_X$ send$(r_q)_{c,b}$, a contradiction $\square$

7.5 Discussion and Related Work

This section discuss the performance of our caching protocol and related work, as well as suggesting some further uses for cacher hosts, including supporting the reliable multicast needed for our model to apply in the first place.

7.5.1 Caching and Performance

The main purpose of cachers is to alleviate servers from handling value read requests, conserving their processing and storage bandwidth for execution of requests and persisting of their effects. Database transactions typically read between five and ten values for every value they write [6]. Moreover, the relative benefit increases as data contention rises, since many applications will be subscribing to and synchronizing with the same names, raising cache hit-ratios.

Cacher hosts make limited demands on servers. Except for their initial value requests to synchronize the names in their caches, they can provide service indefinitely without any involvement from the server, by applying responses to their cached state. Servers do not need to keep track of cachers nor send additional messages for their benefit. Similarly, clients do not need to keep track of cachers and are essentially unaware of caching.

Hierarchical caches are scalable, since growing demand for value messages and synchronization can be met with additional cacher resources. A system based on our model should be able to support large numbers of application hosts maintaining a consistent view of a set of names. The underlying multicast provides efficient update dissemination, while cacher hosts take care of bringing newly added applications up to date with the latest state. This could be a good fit for applications such as large-scale trading and auctions, where many applications observe state and then occasionally issue bid requests. It might also be a good fit with massively multi-user online role-playing games (MMORPGs), where each player is subscribed to an area of interest surrounding the player's current location in the game but must frequently (un)subscribe to parts of the world at the area's edge, as the player moves across the world.

While outside our scope, coordination of cache contents [74] could be used to increase cache-hit rates and dynamically adjust caching strategies. For example, as a server nears full utilization, it could send out a "cry for help," calling for cachers to cache certain name ranges. A cache nearing saturation could similarly cry to cachers at the next chain-length level above, and so forth. The intent is that names should get cached / replicated to a degree commensurate with the number of applications synchronizing with them.

Our design is unique, as far as we know, in providing scalable, consistent caching irrespective of data update rates while not adding latency. Since switches know from their subscriptions which cachers are currently available for a given name, they divert requests only to cachers that can service the value request for that name, with high probability. Requests do not bounce between hierarchy levels, which would add latency [163]. If cachers can be placed close to clients, the round-trip time to a cacher can be made shorter than the round-trip-time to the original server. For example, a cache placed near a site's gateway to the Internet could significantly lower access delays for data commonly used by applications in that site.

In chapter 6 we compared our caching algorithm to AOCC [162], an efficient conflict-based concurrency control algorithm bearing many similarities to ours but operating exclusively end-to-end. We do not see a straightforward way to extend AOCC to support cacher hosts, since an AOCC server is solely responsible for detecting all conflicts and must be explicitly aware of all subscriptions of all applications, so that it can maintain and propagate invalid sets. In our caching scheme, by comparison, servers are oblivious to clients and cachers, but the switches along a diverting path ensure that stale value messages are freshened through conflict-detection with concurrent response messages.

We observe that the distance of a cache's diverting switch from a server has no effect on the expected lengths of value message response list, which depends only on the length of the path from the diverting switch to the cacher. We also observe that the response time for value read request is essentially independent[18] from a server's data update rate. Higher rates only increases response list lengths and create additional work for application hosts. Assuming applications have enough bandwidth and resources to process the response stream in the first place, this additional burden should be negligible.

The price for this performance, though, is the assumption of efficient yet fine-grained name subscriptions on switches. As discussed in Section 5.10.1, we do not address the feasibility of such subscription processing in this dissertation.

Internet caching has received considerable interest in recent years [164], including approaches based on multicast [76]. Ours is an *en-route caching* [165] scheme, since it places caches along the forwarding path towards a data source. While methods for optimizing the placement [165] and coordination [166] of such caches are mostly orthogonal to our algorithm, they are applicable.

7.5.2 Cachers as Service Providers

Efficient, transactionally coherent cacher hosts may have other uses besides serving value requests. Their location to the side of the "pathway" of requests and responses traveling back and forth to a server makes them well placed to provide services related to that server. We sketch a few ideas here, leaving proper consideration to future work. The common theme is to use cachers to provide certain services without encumbering data servers, whose performance may be the bottleneck for one or more applications. This is more attractive from a scalability standpoint than requiring each such service to execute only on original data servers.

A cacher host c could *scrub* a "dirty" request q from an application a that is not fully synchronized with the $depends(q)$ set of q. Application a would send the values[19] it has cached for the subset of $depends(q)$ that is not synched at a and switches

[18] Independent insofar as the updates do not require a large share of the available bandwidth.
[19] Or achieve the same effect through the use of timestamps or other version information.

would preferentially try to divert such requests to cachers for those names. If a cacher c receiving q found that each of these dirty values matched its own (synched) values for the names, it would re-issue q on the behalf of a as a normal, "clean" request. This request would get conflict checked and handled just like any other request, and a would receive its response directly from the server. On the other hand, if some of the dirty values did not match, c would discard q. It could also go ahead and send value messages for the stale names, to hasten their synchronization at a. This scheme would allow an application to issue a request without waiting for its synchronization processing to complete. Alternatively, the application could attempt to issue a request without synchronizing at all. This may yield an overall gain over performing synchronization when an application issues requests infrequently and/or the state in question changes infrequently. Note that it essentially emulates classic optimistic concurrency control algorithms based on server-side validation of transaction reads before commit. The difference is that it does it in a scalable manner, as the validation effort can be distributed onto multiple cacher hosts.

Cachers could serve the role of "data warehouses," by storing one or more snapshots of entire data subsets in a read-optimized format, with extensive indexing for efficient query processing. This practice is common today, as it is very difficult to efficiently combine long-running read transactions with short-lived update transactions. A fundamental reason why is that the probability of transaction conflicts grows quadratically with transaction sizes [12].

In general, cachers could serve as entry-point "switches" for clients of distributed applications, providing services as well as aggregating clients for the purposes of increasing the scalability of fine-grained multicast subscriptions. An application host would use cachers as a bridge from a legacy network protocol into the atomic network, translating back and forth between atomic multicast and a unicast connection to the host, for example. The cacher would handle conflict-checking on the "last mile" between end-host applications and an atomic multicast network, treating each host as if connected by a separate channel. A cacher that is trusted by an organization could perform services such as usage metering of the organization's data, as well fine-grained access control and encryption of response streams for individual hosts. While commodity hosts cannot match the performance of dedicated switches, they can still handle considerable traffic and manage numerous connections [115].

7.5.3 Caches, Response Synch and Reliable Multicast

The cache synchronization algorithms we have described are based on applications receiving state updates that replace existing cached values regardless of their previous values. An alternative would be for an applications to receive a sequence of old responses that roll forward its cache to a current state. The obvious advantage of the state transfer method is that it "truncates" the response history, providing an upper bound on the message complexity of synchronizing a cache that depends on the size of its state, not the number of responses that have affected the state in the past. Also, response sequences require some form of versioning for states and responses, so that the appropriate subsequence of responses can be applied to the old version of the state. However, the response synchronization method may have a sigificant advantage in the case where hosts lose their subscription only briefly. In that case, transmitting the sequence of responses missed during a subscription outage may require fewer messages than explicit state synchronization. Cachers could readily offer this alternative type of service in addition to the state-based one we described, since they receive all responses affecting their cached state anyway.

In fact, cachers are ideally situated to play the role of message loggers in receiver-based methods for reliable multicast [148, 149], discussed in Section 5.10.1. We could take an approach similar to LMS [167], for example, where Negative ACKs (NACKs) are diverted off the path towards their destination server and towards some switch termed the NACK's *turnaround point*, which directs it towards some *replier* receiver host that can supply the missing message. The replier unicasts the missing message to the turnaround switch, which multicasts it back to the receivers. In our model, cachers would play the role of repliers. We might even improve upon LMS by initiating repairs immediately at the (sending or receiving) switch where responses are dropped, sending repair requests to cachers for affected names along with identifiers for the dropped responses. Cachers would, at their discretion, retransmit the missing responses (or value messages) for the names affected, possibly after getting a "congestion cleared" notification from the switch. This could lower repair latency while largely eliminating the need for applications to issue repair requests. While this approach seems promising, we have yet to develop it. Fine-grained, name-based multicast subscriptions and forwarding present a problem for reliable multicast, but they might be part of the solution as well.

Chapter 8

Prototype Implementation of Conflict Checking

The feasibility of Atomic Transfer depends critically on how effectively requests and responses can be conflict-checked. In particular, is is essential that conflicts on switches can be checked at line rates, since otherwise packet queues can build up and overflow. This chapter describes a data structure intended to address that challenge, dubbed Tree Intersection Bitmap (TIB). It is a succinctly encoded trie, based on the Tree Bitmap [168] trie datastructure. TIBs are designed to rapidly detect whether two sets of names overlap, where names are variable length bit strings. We also sketch a high-level design for switches and how they could use TIBs to achieve good conflict checking performance.

While TIBs are designed with Network Processors [169] in mind, we implement and test them on the Intel x86 architecture, with an implementation written in the C programming language. Our primary objective is to get a "ballpark" figure for the performance with which conflicts can be checked, in best as well as worst-case scenarios.

8.1 Hierarchical Naming Scheme

Our prototype assumes a naming scheme similar to the one suggested in Section 5.10.1. More concretely, a name $n \in \mathit{NAMES}$ is encoded as a variable-length bit string $m \cdot n$, whose first $\mid m \mid$ bits represent the name's authoritative owner, as a 64 or 128-bit cryptographic digest of the name owner's public key, for example. The remaining $| n |$ bits identify a particular name in the "name space" defined by m. Management of that name space is the owner's responsibility, but the means by which owners delegate responsibility for names to the public keys of host and data center principals [170, 171] are largely orthogonal to Atomic Transfer and will not be considered here.

We make the important assumption that *names give certain clues about their physical whereabouts*. This is not to say that a name corresponds to a particular network address or host: rather, we assume that the longer the shared prefix of two names (the more *similar* the names, as we term it), the fewer network hops generally separate their hosts. In particular, two names of equal length that differ only in their last bit should be highly likely to reside on the same host. The reason for this assumption is that it dramatically facilitates name-based forwarding and conflict checking. It enables hierarchical routing, as in the Internet Protocol, that is: switches can aggregate forwarding information for a large set of similar names by associating it with their shared prefix. Similarly, we will assume that a request usually refers to a set of similar names, which can therefore be succinctly encoded in tries and efficiently checked for conflicts with the names in similarly encoded responses. While our model would work correctly with arbitrary, meaningless identifiers, performance of conflict checking would suffer considerably.

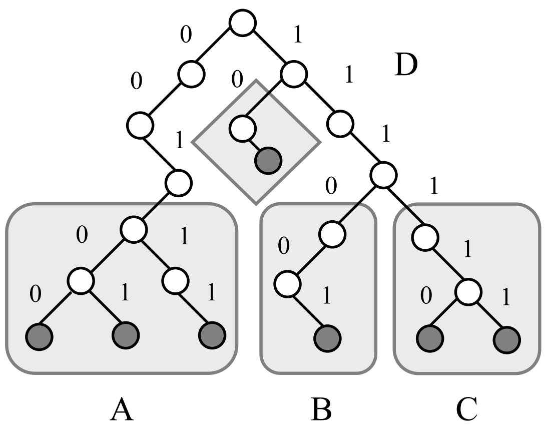


Figure 8.1: Example mapping of names to nodes

Figure 8.1 shows an example of the mapping of a tiny name hierarchy to nodes in a network. The names with prefix "0" map to server A, while prefixes "1110" and “1111” map to servers B and C, respectively. Prefix “10” maps to switch D, suggesting that switch configuration data can be presented using global names and updated using atomic actions. As an aside, we conjecture that this could simplify network management and promote routing flexibility [172].

We see the data names used at the Atomic Transfer protocol level as occupying a middle ground between network addresses and human-readable and meaningful identifiers. A name by itself gives no information about its host and can be transparently re-mapped from one host to another. However, a name’s binding to a human-readable moniker or higher-level abstraction or semantic happens through some form of indirection, for example through interpretation by a program or a cryptographically verifiable chain of logical statements [170]. Hence, a name serves a dual purpose: it uniquely identifies a datum and also encodes its physical location relative to other names, to some degree.

A hierarchical naming scheme does represent a trade-off, though, as naming things hierarchically can be restrictive and changes in data access patterns usually mean that any “clustering” of data into hierarchies must be intermittently adjusted to maintain performance. Load re-balancing is achieved by migrating data between servers while simultaneously updating forwarding tables, so that names still refer to the same values. Renaming data items and otherwise reconfiguring the name hierarchy requires even more coordination. However, dynamic reconfiguration [70, 173] is greatly facilitated by atomic execution, since changes to application data, application software and configuration “metadata” (such as human-readable monikers) can be performed in atomic steps. The impact of large-scale changes can be mitigated by dividing them into a preparation phase and a subsequent installation phase, that moves in a wave across the system [71]. So on a balance, we feel that the benefits of hierarchical naming outweigh its costs.

For concreteness we sketch how a few common data abstractions can be mapped into a hierarchical names in the name/value state abstraction of the Atomic caching systems.

- A hierarchical file system can be mapped directly by encoding the full pathname of each file into a bit string name, terminated with a designated character that is not a part of any filename. The name's value is the file's data. Alternatively, the bits following the name terminator could be used as a binary-tree like subdivider for the file's data, allowing block level or even byte-level access to parts of the file.

- A collection of Java-like objects in a heap can be mapped by assigning a code to each of the object class' fields and storing the value of each field under name $o \cdot c$, where o is a unique identifier for the object and c is a code identifying the field. Objects created as a part of a larger object assembly or aggregation could be given similar names, to increase the likelihood of them being co-located.

- A database relation r can be mapped by storing each row as name $r \cdot k$ with value v, where k is a concatenation of the row's primary key column and v is the concatenation of the other colums. The absence of a row is indicated by the absence of the name corresponding to it. Alternatively, each non-key column could be assigned a unique code c and the value of column c stored under key $r \cdot k \cdot c$. Note that in the latter case, the whole row can still be obtained by a value read request for prefix $r \cdot k$.

- Say a relation r' is predominantly accessed through a join with relation r from the prior example, that uses column c of r to look up a row of r' by the primary key of r'. Then we might store each row of r' by name $r \cdot k \cdot c_{r'}$, where $c_{r'}$ is a code not denoting a column in r. This would increase the probability of rows of r' being hosted close to the rows of r through which they are accessed and permit convenient access to a row k and its join dependencies, by subscription to prefix $r \cdot k$.

We note that these mappings would generally be transparent to end-users and programmers. A database programmer, for example, can still work in terms of tables and queries while (re)mappings of data take place behind the scenes. The pre-joining of rows could be designed by a clustering algorithm that infers access patterns from execution traces.

We assume that name/value pairs are mapped onto hosts along name hierarchies, so in general names with prefix p are assigned to the same host or next-hop switch. Note that servers are free to choose their internal data representations, e.g. hash tables or hierarchical file systems. Application hosts can independently choose the data representations best suited to their needs. The naming scheme does not dictate internal data formats, it merely creates common ground for effective access to data and, as we discuss in the next section: efficient conflict checking.

8.2 Organizing Packets for Fast Conflict Detection

This section discusses our prototype design and implementation for the encoding of request and response name sets into network packets. We focus on the encoding of name sets and those aspects of operations that pertain to conflict relations, ignoring how other message information is encoded. For example, in our empirical evaluation we use simple read/write operations and encode the operation type along with the name to which it is applied, but leave unspecified the encoding of the values read or written. Also, we limit our discussion to messages that fit in a single packet. We note that atomic messages can be broken into multiple packet, as we discuss in Chapter 9 on future work. Furthermore, many high-performance transactional systems, such as On-Line Transaction Processing (OLTP) applications, predominantly create small transactions that access only a few data items each. The widely used TPC-C transaction performance benchmark [174] in fact uses only such small transactions.

Our encoding is based on the trie [175, 176], a fundamental data structure for storing a set of strings over a fixed alphabet. A trie for an alphabet Σ with $\mid \Sigma \mid = b$ is an ordered tree of degree b, where each child of a node corresponds to a letter of the alphabet. The search procedure for a string s begins at the root of the tree. For each character of s, from first to last, the search navigates to the corresponding child until a leaf is reached or no child is present for a character. Tries and prefixes have a direct correspondence, since all strings sharing a prefix reside in the same subtree. The longer the prefixes shared by a set of string, the more compact their trie representation. On the other hand, a straightforward trie implementation where each node has an array of b child references is relatively space inefficient, especially for larger alphabets and sparser tries. Furthermore, for a low b such as a binary $b = 2$, a naive implementation is relatively slow, requiring $O(n)$ steps to search for an n-bit string.

8.2.1 Multi-bit Tries and Tree Bitmaps

What makes tries a candidate for high-speed set intersection checking is recent work on using tries for fast IP lookup on routers [177, 178, 168], which has yielded very efficient trie variants on which we base our approach. The first insight behind this work is that the search needs to proceed in strides of several bits at a time, for speed. While several multi-bit trie data structures have been proposed[20], one of the simplest yet best performing is the Tree Bitmap [168], shown in Figure 8.2. Starting at a node in a regular trie, it is converted into Tree Bitmap form by encoding the next n levels below the node into a single Tree Bitmap node, which we shall call a *block*. Since each block corresponds to a small binary trie of height n we can succinctly encode the set of n-bit strings (or seen in a different light, the n-bit characters) present in a block using a bitmap of length 2^n, whose k-th bit is set only if the k-th string is present in the block, that is: the string whose path from the top node in the block is labeled with the binary representation of k. We will refer to a string as being an *entry* of a block. An entry can either represent a *member* of the original trie or a *prefix* that extends into a child block.

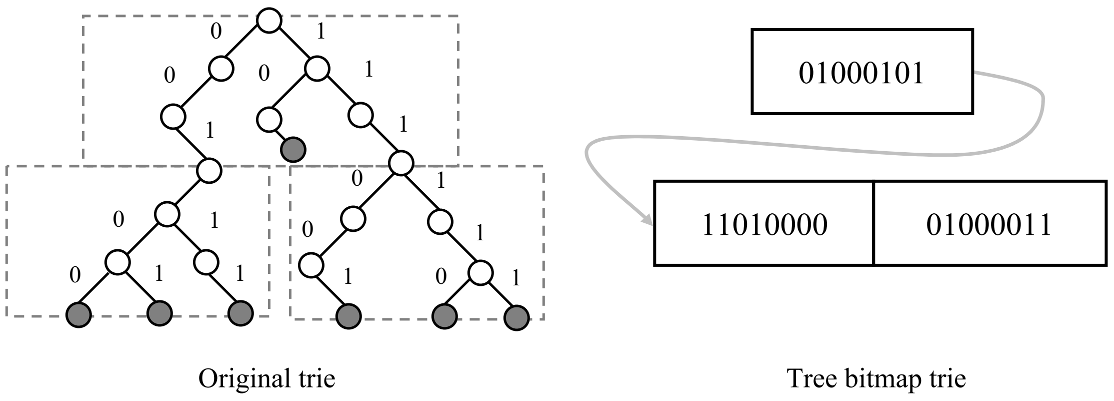


Figure 8.2: Example tree bitmap encoding with stride $n = 3$

Figure 8.2 shows how a trie is converted into three Tree Bitmaps blocks with stride three. The trie contains the string "101" of length 3 and six other strings each of length 6. The top-level block has a 1 bit in positions 1, 5 and 7, since prefixes "001" and "111" are present in the block, as well as member "101". We describe later how the encoding tells members from prefixes. We make the assumption that names are

[20]Reference [168] presents an overview.

prefix-free, that is: no name is the prefix of another name. When the need for prefixed names arises, such as in the example in Section 8.1 of mapping a file system to a name space, the standard trick of terminating names with special symbols not used as a part of any name can be employed.

In the case of the original Tree Bitmaps, another bitmap is needed to account for the fact that IP forwarding prefixes may have arbitrary length and that IP forwarding databases are not necessarily prefix-free. We simplify our implementation by stipulating that the length of all names be a multiple of n. In fact, we will choose $n = 4$ as our stride, since it yields bitmaps of reasonable size (16 bits) that are convenient for software implementation, being a power of 2. TIBs add 16 bits of information to round out a 32-bit block which suits architectures requiring 32-bit memory access alignment.

Tree Bitmaps save space by storing sibling blocks contiguously in memory, so a single pointer suffices to navigate to all the child blocks of a parent block. In Figure 8.2 the gray arrow represents the pointer from the top-level block to its first child block. This convention makes blocks highly uniform: each block has a bitmap and a pointer to its first child block, if any.

The Tree Bitmap encoding is very concise. Moreover, it is well suited to computing name set intersections. When comparing the two top-level blocks of a Tree Bitmap trie, for example, we can determine the set of 4-bit entries they have in common by performing a bitwise AND operation on their respective bitmaps, computing the set of up to 16 possible common entries at a stroke. This is the key to the efficiency of the data structure we adapt from Tree Bitmaps, the Tree Intersection Bitmap.

8.3 Tree Intersection Bitmaps

Our Tree Intersection Bitmap is essentially a Tree Bitmap less the encoding of arbitrary length strings plus the encoding of operation information. We discuss how child pointers are encoded at the end of this section.

We could easily use the k-th bit of the remaining 16 bits of a block to indicate whether the k-th entry in the block is a member or a prefix. However, that encoding would be fairly inefficient as most nodes below the uppermost levels of a trie are sparsely populated. This is true for tries containing random strings [179], and we expect it to be true for request and response name tries as well. Therefore, we store the additional information as a list of consecutive j-bit codes in the remaining 16 bits of a block, which we will call a block's *list*.

This scheme is clearly more complex to decode than a straight bitmap. However, the space efficiency gains are important because they reduce the number of memory fetches a processor must make during intersection checking. The speed differential between Dynamic RAM (DRAM) memory chips and processors has become such that a processor can execute tens or hundreds of instructions in the time it takes to fetch a chunk of data from DRAM, although that chunk can be relatively large (e.g. 32-64 bytes). General-purpose processors mask this latency to a large degree using several levels of on-chip cache memories, but packet data streaming through a network device has negligible temporal locality, limiting the effectiveness of caching. As a result, the key to good performance is to fit as many TIB-blocks into each chunk's worth of memory as possible.

The choice of j depends on the number of distinct operations or operation equivalence classes that must be encoded. For our empirical evaluation we assume simple read/write operations, similar to table 4.1 of Chapter 4, and use $j = 2$. We will assume all write operations read too, as "blind" writes without reads are rare in practice. Also, our conflict relation ignores the values being read or written. The only case that benefits from comparing the values is when a read does not conflict because a write is writing that same value, which mainly occurs if requesters generate unnecessary writes. Our conflict relation is shown in Table 8.1. While the rest of our discussion is framed in terms of our chosen stride and operation encoding, we discuss generalizations later in the chapter.

Figure 8.3 shows the format of an example block containing entries 1, 8 and 9 (corresponding to bitstrings "0001", "0100" and "0101"). The block's list contains the respective codes for the entries, "01", "00" and "01". The meaning of list codes is as follows:

	read()/v	write(v)/OK
read()/v'		true
write(v')/OK	true	true

Table 8.1: Conflict relation for a simple read / write register.

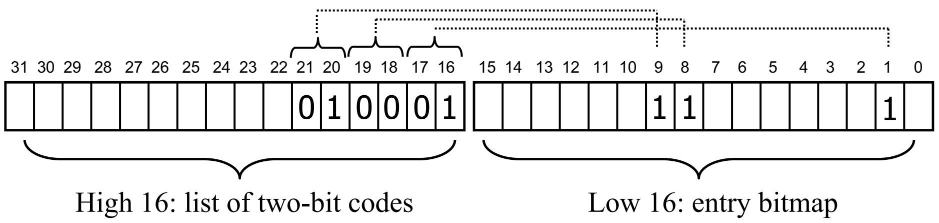


Figure 8.3: Tree Intersection Bitmap Block with three entries

- If the lower bit is set, the entry is a prefix and the higher bit is unused.
- If the lower bit is clear, the entry is a member, and:
  - If the high bit is set (clear) the member is a write (read) operation.

In the example of Figure 8.3, members 1 and 9 are prefixes while members 8 is read operation.

To navigate from a parent block to the block corresponding to the parent's n-th prefix entry, we count the number of prefix bits to the right of the nth entry in the list (Tree Bitmaps use a similar technique). For example, the list of Figure 8.3 contains codes for entries 1, 8 and 9, from right to left. There is one prefix code to the right of the code of entry 9 (the code of prefix 1) so the child block corresponding to prefix 9 is the second block from the first-child pointer, which points to the block for prefix 1. The number of prefixes to the right of a prefix can be computed in constant time, by masking off all even bits and bits to the left of the prefix in the list and the counting the population of the bits remaining.

Since we only have 16 bits in which to encode up to 16 2-bit entries, a block's bitmap overflows if it has more than 8 entries. If any children among siblings overflow, we capture the remainders of their lists in *overflow blocks* following the last sibling block in memory. The first overflow block, if any, has a 16-bit *index bitmap* with bit k

set exactly if the k-th sibling overflows. The sequence of 16-bit overflow lists follows, alternatively in the high and low-order halfs of the 32 bit overflow blocks. Figure 8.4 shows an example with four TIB-blocks where block 0 and block 2 overflow. The lower half of overflow block 4 contains the index, while the upper half of overflow block 4 and lower half of overflow block 5 contain the remainder of the lists of blocks 0 and 2, respectively.

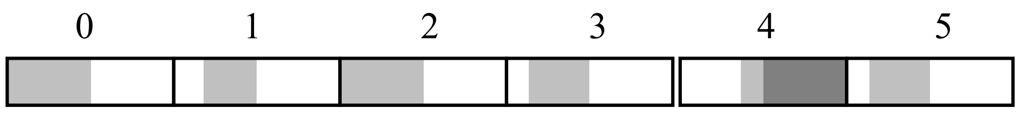


Figure 8.4: TIB-blocks with overflowing lists and overflow blocks

Overflows should be relatively rare, occurring mainly at the top of densely populated tries and when a collection of leave nodes with the same name lengths differ only in their last two or three bits. But even when they occur, the overflow list can be extracted in about 20 instructions.

Figure 8.5 shows the high-level algorithm for checking whether the TIBs rooted at two blocks intersect, where o.list(n) denotes the n-th two-bit code from the list of o. It first computes c, the bitwise AND of the block bitmaps. Each set bit of c corresponds to a prefix or member that the blocks have in common. For each such bit, the algorithm either recurses into the child blocks of the common prefixes or else checks whether the member operations conflict. In the case of our conflict relation, operations conflict if either is a write, so the n-th members of blocks a and b conflict exactly if (a.list(n) OR b.list(n)) AND $2 \neq 0$.

We next describe the most important details and optimizations of our implementation.

We avoid recursive procedure calls by maintaining an explicit stack. Each stack entry records information about two blocks, including how far along their respective lists the algorithm has progressed. Avoiding explicit recursion is faster and allows our code to be compiled for architectures lacking a hardware stack, such as IXP microengines. [112]. We skip stack pushes and pops whenever possible, e.g. if two blocks have only a single entry in common then we replace the top of the stack with their children instead of pushing them. This also bounds the size of the stack to $\lg n$, where n is

```
**intersect**(block $a$, block $b$)
    let $c = a$.bitmap AND $b$.bitmap
    for each bit $n$ of $c$ that is set: // for each common entry
        if ($a$.list($n$) AND $b$.list($n$)) AND 1
            // both entries are prefixes
            recursively **intersect**() their child blocks ...
        else if not ($a$.list($n$) OR $b$.list($n$)) AND 1
            // both entries are members
            if operations $a$.list($n$) and $b$.list($n$) conflict
                note and / or handle conflict ...
        else
            // error, one is a member, the other a prefix!
```

Figure 8.5: Block intersection algorithm at a high level.

the number of block pairs compared during an intersection, since new blocks are only pushed onto the stack when the current blocks have at least 2 prefixes in common.

We use constant-time bitwise operations to advance from one common bit of c to the next. We use the expression (c AND $-c$) to isolate c_r, the rightmost (least significant) bit of c [180], so $c_r - 1$ yields a mask for the bits to the right of c_r. We use that mask when counting[21] the number of bits to the right of c_r in a block's bitmap, yielding the index of the corresponding entry in the block's list. Hence, the main loop proceeds in time $O(\mid a \cap b \mid)$, where we take a and b to denote the set of entries in blocks a and b, respectively.

The last point is important because we expect request tries to be "skinny" at the top, branching into "bushes" primarily near the bottom. Assuming name hierarchies are designed to achieve high locality data accesses, most names in a request will be highly similar and have a long shared prefix. This results in a low $\mid a \cap b \mid$ for upper-level blocks and fast intersection checking for the long "stem" of request tries. Other things being equal, longer stems increase the probability of intersection checks terminating before reaching the "bushy" parts of tries, in the common case where tries do not intersect.

[21]Many architectures have a "population count" instruction that can perform this count in a cycle or two. Since the x86 architecture only recently gained such an instruction (with Core i7 and SSE4.2) our tests simulate it with a slower table lookup and additions.

As described, TIBs do not encode long stems particularly efficiently, requiring $8n$ bits to encode an n bit long prefix stem, e.g. 32 bytes for a 32-bit long stem. If “skinny” request with long stems are indeed common, much of the first DRAM fetch would be spent on the stem, necessitating further fetches.

In the special case, therefore, when a block has a single prefix entry and continues to form a single stem for one or more levels, we use the block’s list to encode up to $3 \cdot 4 = 12$ bits of that stem. Following the entry’s operation code in the list we place a 2-bit count of the number of 4-bit prefix extensions present in the list (from 0 to 3) with 4, 8 or 12 bits of next-level prefixes in the remaining bits.

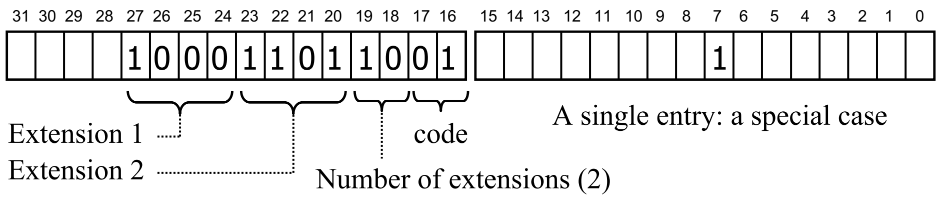


Figure 8.6: A TIB-block with two prefix extensions

Figure 8.6 shows an example, where the prefix “010110001101” is encoded with prefix extension in a single block, instead of being encoded into three one-entry blocks containing prefixes “0101”, “0001” and “1101”, respectively. Note that extensions appear in reverse order on the list, with the first extension in the most significant bits. This makes decoding a bit faster. The prefix extension optimization can encode an n-bit stem in as little as $2n$ bits, allowing the 32-bit stem we used as an example to fit in 8 bytes.

What remains to be discussed is the encoding of the pointers from a block to its first child block. A straightforward way is to keep the pointers in a separate array, so the first-child pointer of block n is the n-th element of the array. For example, 8-bit block indices would work for tries of up to 256 blocks, fitting a small Ethernet Maximum Transmission Unit packet. The space overhead for this method is a relatively modest 25%. A disadvantage of this method is that it requires two chunks of DRAM to be fetched at each time: a chunk of TIB-blocks and the corresponding pointers. This might be remedied by interleaving pointers with blocks, but such schemes are complicated by variable-length sibling blocks and the internal fragmentation resulting from the mismatch between 8-bit indices and 32-bit memory alignment requirements.

We do not develop such a scheme for our experiments but use an array of pointers. However, a reasonable direction for future work is to investigate the use of the HEXA hashing technique [181], where a prefix, from the root though to a prefix in the current block, is hashed to yield the index of its first child block. Collisions are avoided by including a few freely varying *discriminator bits* from nodes in the hash, where n discriminator bits give a choice of 2^n hashed locations for the first-child node. In our case, unused list bits could serve as our discriminator bits. Since most blocks will have a few unused list bits, choosing discriminators that hash to a free block should generally not be a problem. However, we must consider the run-time costs of constructing such tries and traversing them.

On the other hand, using explicitly encoded pointers affords us full freedom to lay out subtries in efficient ways, for example as close as possible to the cache-oblivious "Emde van Boas" layout [182], to minimize the number of DRAM fetches needed during conflict checking. Achieving a combination of HEXA-style hashing and cache-obliviousness could be an interesting problem for future work.

Spending some effort on the careful encoding of requests on the application side seems justified, if it accelerates network and server-side processing. This effort may be less welcome on the server side. We note that the cost of encoding packets is less of an issue with the atomic caching systems presented earlier in this thesis, since applications can include responses with their requests, obviating servers from encoding response packets.

8.4 Switches and Responses Sets

TIB intersection is performed on switches, when the TIB of a request inbound on a switch port p is compared with the set of responses that have been enqueued for p. This set of responses corresponds directly to the $\textit{responses}_p$ set for a channel p in our I/O Automata model of switches (see Chapter 3). While the details of an implementation will depend on the hardware architecture of the switch for which it is intended, we sketch a generic design for a switch that buffers packets at outbound ports. Routers and switches typically buffer packets in DRAM memory, while queues and other data structures are often kept in faster (and more expensive) Static RAM (SRAM) memory.

High performance packet switching does not allow for much processing per-packet. A 1Gbps port, for example, sending a steady stream of 100-byte packets must forward roughly a million packets per second to keep up with line rates, that is: a packet every microsecond, on average. As a rule of thumb, a switch can execute around 10 instructions per byte of packet data [183]. Clearly, a switch cannot perform extensive data structure maintenance as a part of processing a packet. Yet, it must somehow maintain the *responses* set for each port.

The size of a port's *responses* set depends on its response sending rate and the residency times of responses on the switch, which is upper-bounded by the amount of packet memory available. Switches are designed to minimize packet residency time, since it corresponds directly to latency. An average delay of $100\mu s$ for our 1Gbps port sending nothing but 100 byte responses would by Little's Formula have around 1.000.000 responses/sec * 0.0001 sec = 100 responses in its responses set at any one time. Clearly, checking a request against each in turn would be time consuming. Hence, we do not store *responses* sets as lists but rather as a trie, so that a request can be intersected with a single *responses* trie instead of many response tries.

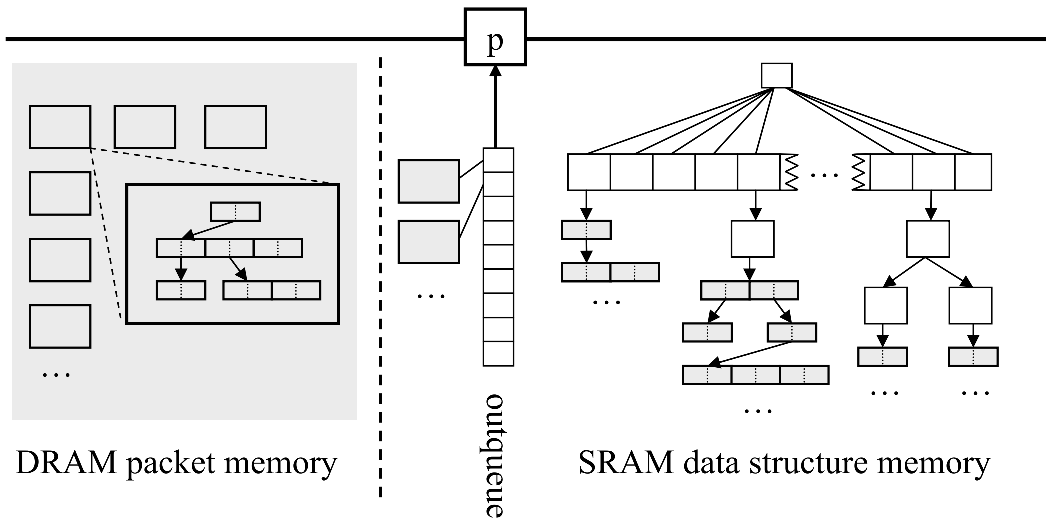


Figure 8.7: High-level schematic of conflict-checking switch port p

Our basic approach to maintaining switch *responses* tries is to incorporate TIB-encoded packets into them wholesale. Instead of adding request names individually to a switch's trie, which would be hopelessly slow, we propose to "wire" a response packet into the switch trie directly, by adding a pointer from the appropriate block in the switch trie to the appropriate TIB-block in the response's TIB trie in packet memory. Figure 8.7 shows a schematic for this design, for a port p on an atomic switch.

White boxes represent data in SRAM while gray boxes represent data in packet DRAM memory. The schematic shows a "blown up" view of one response packet, containing a trie made of TIB-blocks. We assume the standard optimization of having an *initial array* of trie blocks as the first level in the switch trie, indexed by the first n bits of the stems of packet TIBs, e.g. $n = 16$ for an array of 64K blocks (the names in the domain rooted at the switch are then restricted to being at least n bit long). The top-level blocks may point to packet TIB-blocks or other switch-trie blocks, shown in white. If we let i be the index encoded by the first n bits of a response r being forwarded onto p, then the i-th block in the initial array would be updated to point to the TIB-block of r that continues the trie of r past the first n bits. Upon receiving the ACK for r, the switch removes the pointer to r before reclaiming its memory.

We note that the format of switch-blocks would be different from packet TIB-blocks. They cannot easily use the Tree Bitmap single child-pointer optimization due to the dynamic updates the switch trie must support. Furthermore, it might be beneficial to use larger switch-blocks with a simpler, more direct encoding, so they can be quickly updated. Getting many blocks in each fetch matters less with SRAM than DRAM, since SRAM is faster and it is hard for switch to keep related trie blocks close in memory anyway.

With $n = 16$, the 1Gbps port from our earlier example containing 100 responses in its *responses* set on average, the probability of a response's slot in the initial array being occupied can be approximated as $100/2^{16} \approx 0.15\%$[22], assuming uniformly random n-bit stems in responses. This is a best-case scenario though, since accesses will be

[22]Since $100 << 2^{16}$ we simplify by treating the additions of entries to the array as independent events.

more clustered in practice, yielding higher hit ratios, e.g. 100% in the worst case where all messages have the same n-bit prefix.

When a new incoming response finds one of its prefix slots in the initial array is occupied, the insertion proceeds down the response's TIB and the trie of existing responses on the switch, until a slot is found. Packet TIBs cannot be modified, of course, since they will eventually be transmitted to the next-hop node. Hence, the insertion adds new blocks to the switch trie as it recurses down the TIB of r, performing a "copy on write" instead of modifying blocks. A newly created switch-block will point to the same child blocks in packet memory as the original old-TIB-block. Once the branches of the old-TIB and the new-TIB diverge, the last switch-block added or traversed is updated to point to a child (or children) of the current new-TIB-block.

To recap, the insertion of a response into a *responses* set starts with a switch-block and the top-level TIB-block from the response and then proceeds in two phases (for each trie prefix branch): first the insertion proceeds down switch-blocks, until an empty slot is found in a switch-block or the search "falls off" the switch trie and into an old, existing response's TIB trie. The search then proceeds down the old and new TIBs, adding new switch trie blocks along the way, until the search "falls off" the old-TIB and the current new-TIB-block is inserted as a child of the latest switch trie block added.

When an acknowledgement is received for a response packet, a similar traversal is needed to remove the packet's TIBs from the switch-trie. Alternatively, the switch could store a list of the switch-trie blocks modified by each packet, to accelerate the packet's removal. Once the bitset of a switch-trie block becomes empty, it can be removed from its parent block (and so on recursively) and its memory reclaimed.

While following a long chain of switch-blocks to insert a response is expensive, it cannot be avoided in the worst case. On the flip side, if a particular prefix is very "hot", with many messages containing it, the cost of building a path of switch-blocks for it will be amortized over many responses. In fact, the search may go faster while traversing switch-blocks, as memory chunks for new-TIB and switch-blocks can be fetched in parallel from DRAM and SRAM. A potential problem remains that even if the initial array keeps average response insertion times low, the variance could be quite high due to those responses that lead to long trie traversals. Subsection 8.6.2 sketches a possible solution to this problem.

The performance implications of these trade-offs are best evaluated with a prototype software implementation and simulation, followed by experimentation on a programmable switch. We have not implemented the switch trie or the insertion procedure for responses, but we have implemented packet TIB tries and section 8.5 evaluates the performance of intersecting a packet trie with a large, static TIB trie playing the role of the switch trie.

8.4.1 Trie and Set Asymptotics

The asymptotic behavior of trie intersection does not yield itself to easy analysis. Trabb Pardo's thesis [184] yields a "hopelessly inscrutable answer" for the general case of trie intersections, to quote the author, but a probabilistic analysis yields an average time (number of comparisons) of $O(m \lg(n/m))$ for sets N and M of size m and n, respectively, where $m \leq n$[23]. This is the same as the general worst-case bound for merging two ordered sets. However, it has been shown [185] that if ordered sets N and M can be divided into k blocks $N_1, N_2, \dots, N_k$ and $M_1, M_2, \dots, M_l$ (where $k = l \pm 1$) such that their merged set is an alternation of these blocks, then the sets can be me merged in time $\Omega(k \lg((n+m)/k))$. The lower the k, the "easier" the problem of merging the sets.

In our case, if requests have high locality, accessing mostly similar names, then the names of a new response (set M) are likely to form a single contiguous block, so $k \approx 1$ for the purposes of merging the names of M into the names of the responses set (set N) and the bound becomes $\Omega(\lg(n+m))$, or assuming $n + m \approx n$, $\Omega(\lg n)$. The worst-case of $k = n$ similarly yields $\Omega(n)$. This tells us that conflict checking is highly sensitive to the blocking factor k, and that the effort required to check two messages of similar size against a *responses* set can vary by a factor of $n/\lg n$ depending on how their names interleave with the set.

[23]In our case, m would correspond to the number of names in a message while n would be the number of names in a *responses* set

It also tells us that it is beneficial to keep the size n of *responses* sets as small as possible, to reduce variability. This is something most designs would strive for anyway, since it corresponds to lowering packet residency times and latency. However, this variability is fundamentally problematic in the context of switches and routers, which are usually designed with an eye towards worst-case performance, in order to keep traffic flowing smoothly. Bounding worst-case performance is difficult when the amount of processing for an incoming message can vary by an order of magnitude or two. To avoid wild performance fluctuations and mitigate the potential for denial-of-service attacks, a switch may have to bound the amount of processing it performs for a requests or response, dropping it if the bound is exceeded. Since dropping responses is expensive, the switch might shirk responsibility instead, as discussed in Section 3.6.3.

The main victims of such bounding policies would be large, complex requests (and, consequently, responses) concurrently accessing many names interleaved at a fine granularity. While dropping such requests as contention rises might sometimes be appropriate to protect servers from overload, it might also lead to liveness problems when applications are unable to get complex yet non-conflicting requests through to servers. We briefly discuss an approach for mitigating such problems in chapter 9 on future work.

8.5 Conflict Checking Performance Evaluation

The main objective of our experiments is to get a rough estimate for how many intersection checks a single processor core can perform per second.

8.5.1 Experimental setup

As discussed in the previous section, one of the most important factors for conflict checking difficulty is the *locality* of tries, that is: whether they have highly localized accesses to blocks of similar names or whether the accesses are diffusely spread across dissimilar names. Figure 8.8 shows examples of tries with low, medium and high localities, corresponding to a high, medium and low block factor k, as discussed in the preceding section.

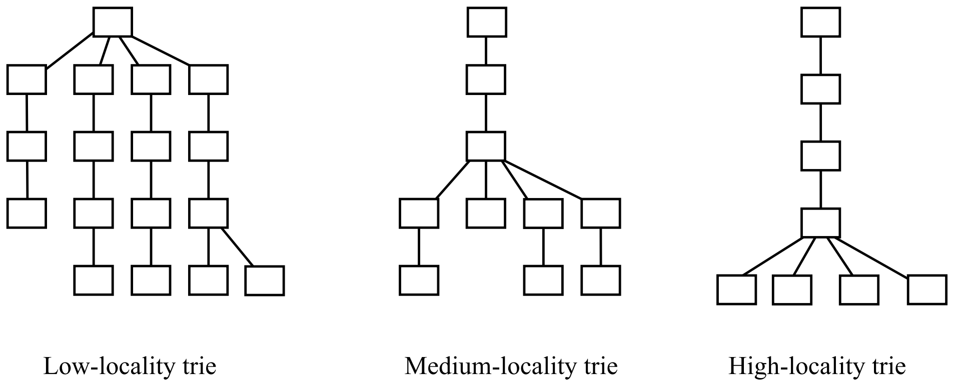


Figure 8.8: Three tries with different localities

We will quantify locality in terms of the distribution of branches across trie levels. For example, the "low-locality" trie of Figure 8.8, has all but one of its branches at the top level. The "medium" locality trie has its branches at its median level, while the "high" locality trie branches at its lowest level only. When generating tries for our experiments we use a parameter $l \in [0, 1]$ to shape locality, as follows. We distribute the branches a trie we generate onto its levels according to a normal distribution centered at $maxl \cdot l$, where $maxl$ is the maximum number of levels in the trie. We use a standard deviation of one-eighth the maximum number of levels, i.e. 2 for $maxl = 16$. We normalize the area of the curve between 0 and $maxl$ to 1 and move some of the uppermost branches down to lower levels if needed, since the top-level can accommodate at most 16 branches, the second level 256 branches and so on. Hence, $l = 1$ gives a highly local trie with most of its branches at the bottom, while $l = 0$ gives a trie with most branches at the top and low locality. Our test routines can generate tries with a given number of nodes and locality factor.

We wish to evaluate our performance for a relatively large namespace, with names of up to 64 bits. Choosing names at random from such as large name space is futile, as the probability of tries overlapping beyond the uppermost levels dwindles rapidly. Instead, we define a *data trie* containing a set of names and then randomly draw names from that trie to create packet tries. The data trie is defined by a recursive pseudo-random generator function and its nodes are only instantiated on-demand as names are drawn from the trie. The generation takes as input the minimum and maximum number of bits in names as well as the average density of child nodes at different levels in the trie. To generate a trie, a random walk is made down the branches of the data trie, choosing names and prefixes in a way that satisfies the specified branching distribution. Our method is carefully designed as not to introduce any bias to the selection. We use an implementation of the Mersenne Twister [186] algorithm from its authors to supply the pseudo-random numbers that drive our experiments.

In our experiments, the length of names in the data trie varies between 16 and 64 bits, with an average length of roughly 40 bits. The data trie is highly dense at the top levels, relatively sparse at the middle levels (around 6-15% block occupancy) but highly dense at the bottom levels, since that's where most of the actual data items would reside. For each locality factor in our experiments, we scale the data trie densities by trial and error to achieve a median conflict ratio of 5%, that is: until roughly 5% of packet tries have at least one member in common with the switch trie. The expected number of names in the data trie ranges from roughly a trillion to 10 trillions, corresponding to the amount of data stored in a small data center or sizeable cluster of servers.

When running an experiment we first generate a switch trie by generating the desired number of (response) tries with member counts varying uniformly between 10 and 500 and then merging the tries into a single trie. For each request trie size that is to be evaluated, we generate a few thousand (request) tries according to the method described here above. We then conflict-check each trie against the switch trie several thousand times, measuring the time elapsed using the highest-resolution timer offered by the host operating system (Windows XP SP 2). We ran our experiments in a single thread on a 3.20GHz Pentium 4 with 1GB of RAM. Note that this is a more powerful processor than some of the network processors we're targeting. Intel IXP2800 Microengines, for example, are clocked at 1.4GHz and have a simple, non-superscalar architecture. On the other hand, a single IXP2800 chip can achieve greater throughput as it comprises 16 such engines.

In all our experiments, we measure our implementation's conflict-checking throughput, measured in tries conflict-checked per second.

8.5.2 Varying Conflict Locality

In our first set of experiments, we vary between 5 locality factors ranging from 1.0 to 0.3. For each factor, we generate a switch TIB trie as described in the prior section, containing 100 "response" tries. We then create packet TIB tries of varying sizes, from 5 to 250 names, and intersect with the switch trie as described in the prior section. The switch trie contains around 25.000 names, in each experiment.

Figure 8.9 shows our main performance results, showing the number of names in tries on the horizontal axis and the millions of conflict checks per second on the vertical axis. For tries with relatively high locality, the performance is pretty good. At 11.03 million checks per second, each check takes about 90 nanoseconds or 290 CPU cycles on average. With 250 names per trie, the performance is still above 2 million checks per second, corresponding to roughly 500 nanoseconds or 1600 cycles. Beyond a certain threshold for trie locality, however, performance tapers off very quickly.

For concreteness, we also plot the approximate conflict checking throughput required to sustain conflict checking at rates of 1Gbps and 10Gbps, using the formula that each packet has 40 bytes plus 5 bytes per name.

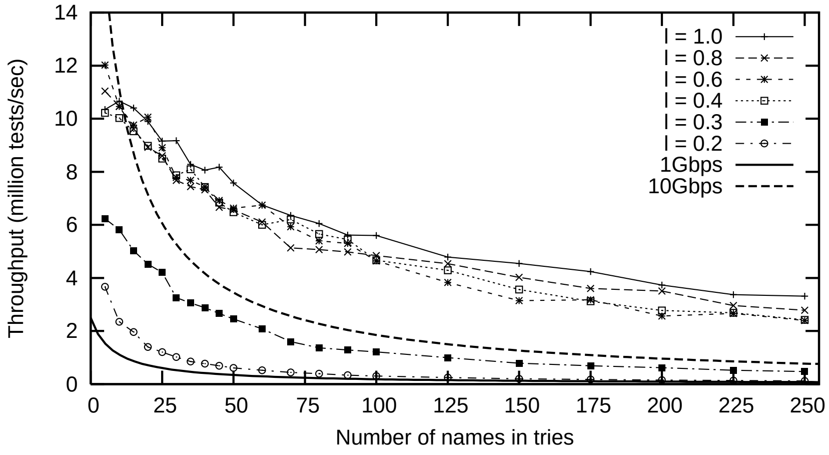


Figure 8.9: Performance as trie localities are varied.

We note that since our experiments repeatedly intersect the same tries, most of the memory accesses are served from the processor's L1 cache, so the results represent mainly processing time, not stalls for memory fetches. We feel that leaving out the effects of the memory system is more instructive in our context[24]. For completeness, we did test the reverse, worst-case order of testing a different trie every time, and this reduces performance steadily from around 55% of what is shown here down to 15%, for the largest tries.

In other respects, our results are conservative. For example, we do not use an initial array in our experiments. An initial array for the first 16 bits (4 levels) removes the need for 3 block comparisons, which would mostly remove the need to look beyond the initial array in the case of high-locality tries of up to 50-100 names. Also, a conflict ratio of 5% is on the high side of the 1-5% range considered typical [6]. Lower conflict rations yield faster checks, as demonstrated by Figure 8.19 on page 193. Our experiments also correspond to the worst-case traffic consisting solely of requests and responses. A more typical scenario would have a significant part of the traffic consist of non-conflict checked value requests and responses.

While our performance results do not transfer directly to network processor settings, due to divergent processor and memory subsystem architectures, they indicate that conflict checking at high rates may be feasible, as long as tries have high locality. Continuing with our 1Gbps port example, it needs to handle roughly a million checks per second to keep up with line rates with small packets of 100 bytes. This seems to be within reach for a processor core of comparable power to a Pentium 4. Note, though, that we do not include the effort required to update response tries upon reception of response packets, which would likely cut their conflict checking rates by at least half.

It is not as clear that a single core could sustain 10Gbps rates using our implementation, once other overheads have been factored in. But if multiple processors and memories can be applied to the checking in parallel, that level of performance may be achievable. For single-name, 45 byte packets, for example, a switch would need to sustain roughly 22 million checks / second. The initial-array optimization or TCAMs (see Section 8.6.2) would help, but sustaining these rates likely requires multiple processors/cores working in parallel.

[24]A tuned implementation on a network processor would mask memory latency to a large degree, using simultaneous multithreading to achieve a similar throughput but with higher latency.

Our conflict checking cannot sustain high rates when tries have low locality, and the analysis of Section 8.4.1 suggests that high performance in that case is fundamentally hard to achieve. Yet, tries corresponding to localities below 0.5 seem unlikely to occur in practical name hierarchies, or can and should be avoided in most cases. A request with significant branching in the uppermost levels of the name hierarchy would likely be forwarded to multiple hosts, requiring an expensive two-phase atomic commit which is expensive compared to requests executing on a single host. Hence, name spaces and host allocation will aim to concentrate related data on the same hosts as much as possible, leading to high trie localities. Generally, most of a name's prefix will correspond to some relatively high-level collection, such as a database relation or a set of related objects. The last, least-significant bits of the name discriminate between individual data items in that collection. Still, our algorithm might be able to sustain 1Gbps of purely low-locality requests.

The performance difference between the high-locality and low-locality cases is largely explained by the number of block comparisons performed, that is: the number of times the intersection algorithm looks at a new block from each trie. While that number remains relatively low for high-locality tries, it grows at a significantly faster pace for low-locality tries. We believe the main reason is that low-locality tries "saturate" their top levels with prefixes, and must therefore initiate searches down many branches, comparing many blocks. High-locality tries, by comparison, have a thin "stem" that "slices" through the dense switch trie top levels, since the maximum number of bits their top-level bitmaps can have in common with the tree switch bitmap is 1.

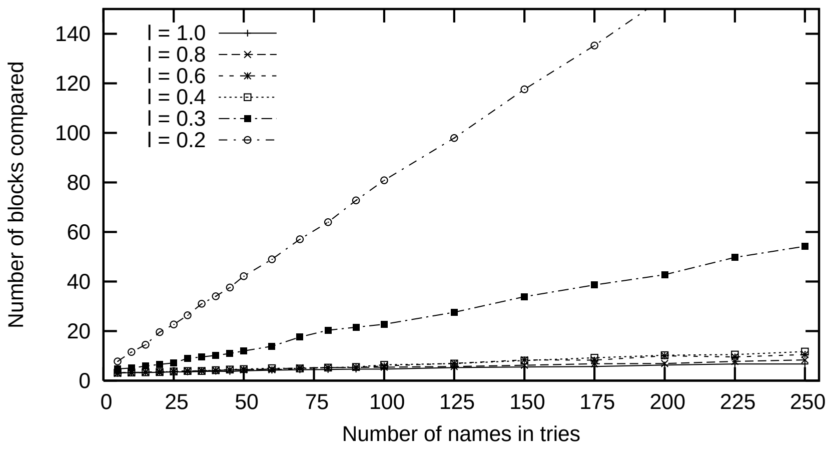


Figure 8.10: Number of blocks compared.

Some of the jitter in Figure 8.9 can be explained by variations in the conflict ratio obtained for different trie sizes. Controlling for conflict ratio variance completely is hard without introducing bias, and in the case of the low-locality tries it grows significantly with trie sizes. Figure 8.11 shows the conflict ratios occurring during the experiment. Since conflict checking performance is very sensitive to the degree of overlap between tries, the variations in conflict ratios show through in our other results.

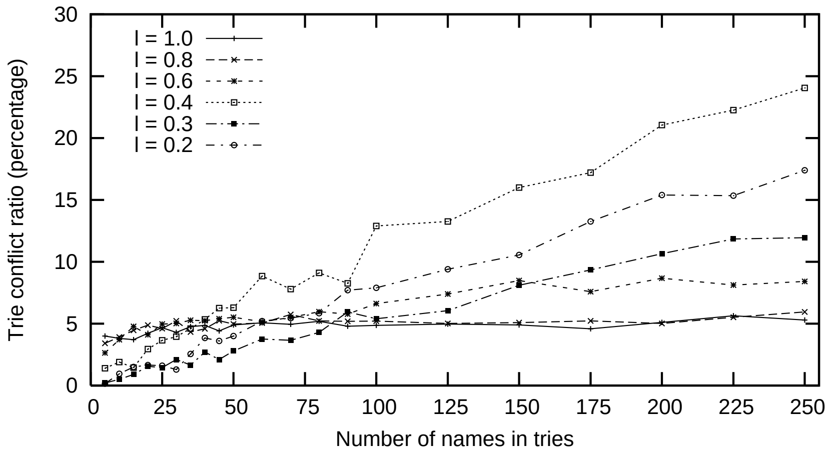


Figure 8.11: Ratio of tries that conflict.

To evaluate the number of DRAM fetches that our implementation would incur, we instrumented our implementation to note from which 32-byte section of memory each block from the trie being checked against the switch trie was "loaded". This number corresponds to the number of fetches a network processor would have to make to conflict check an incoming request or response, assuming 32-byte DRAM bursts. Figure 8.12 shows the average number of chunks per check, leaving out the low-locality cases to reduce clutter. We plot the number of chunk fetched per block compared, in Figure 8.13. We are not sure why the extreme low-locality case has such a low ratio. We suspect it is because these lowest-locality "fill in" the upper levels underlying data trie, resulting in packet tries and the switch trie taking on the same shape and ending up with the same arrangement of blocks into chunks.

Aside from performance, the size of the tries is of interest. We report the size of the blocks alone, without including any overhead for child pointers. Figure 8.14 shows the number of blocks for tries, while Figure 8.15 shows the average number of bytes

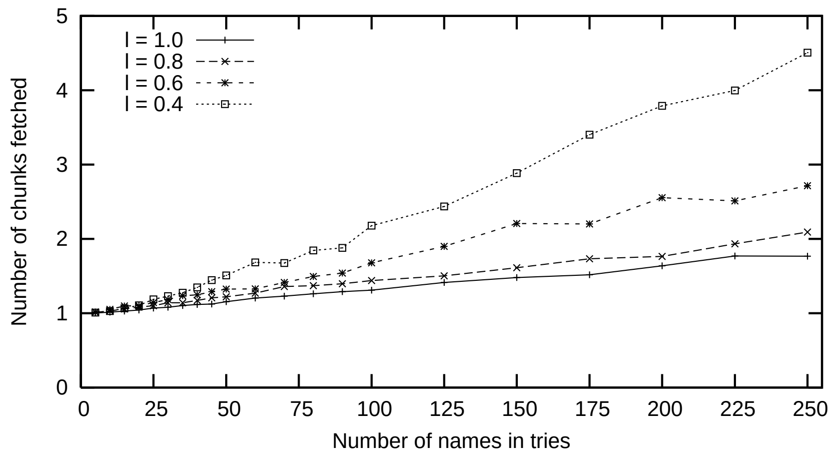


Figure 8.12: Number of memory chunks "fetched" in tries.

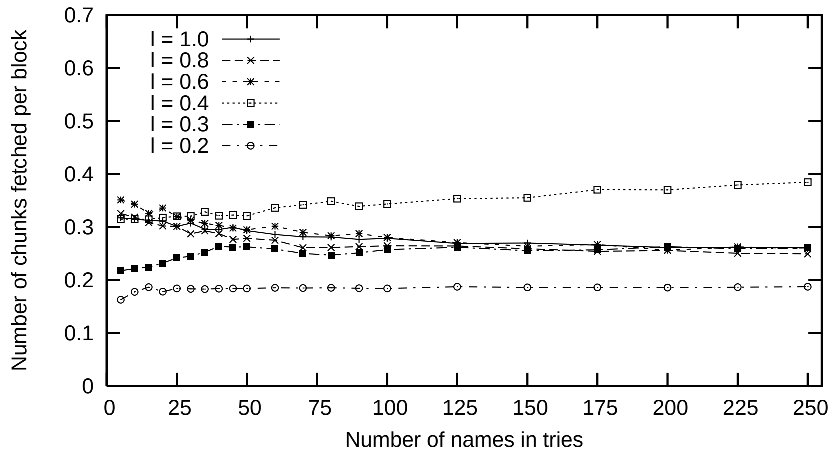


Figure 8.13: Number of memory chunks "fetched" per block.

per name encoded in a trie. Unsurprisingly, the higher the trie locality, the more efficient the encoding.

The results presented so far correspond to the case where a response is added to the switch trie, since we make the intersection test find and report all intersections. However, when conflict checking requests the search can be terminated as soon as the first conflict is found. Figures 8.16 shows the performance in this case.

These results probably overstate the benefits for high-locality tries, as in our experiments conflicting requests tend to have very many names in common when they do

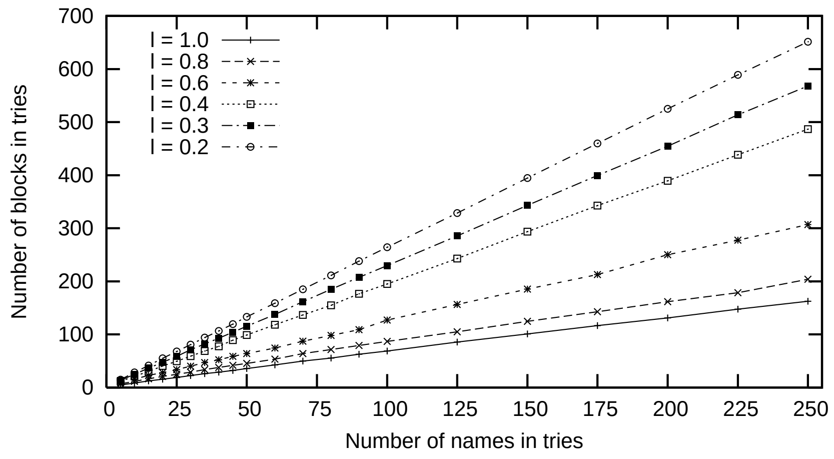


Figure 8.14: Number of blocks in tries.

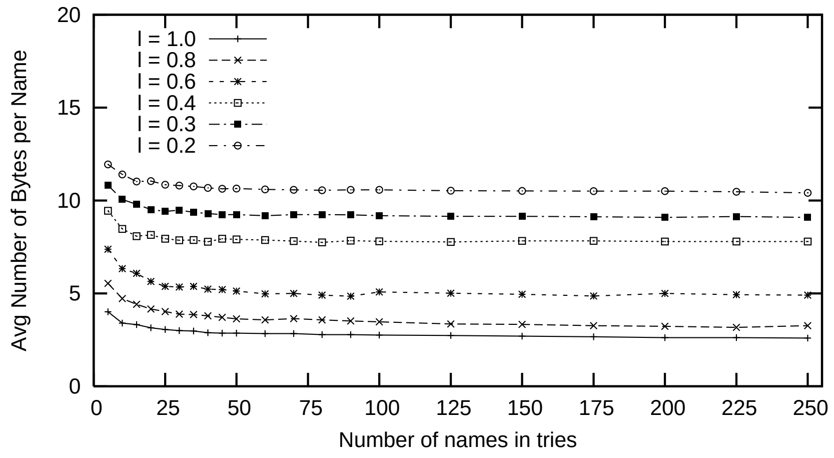


Figure 8.15: Number of bytes per name encoded in tries.

conflict, as shown in figure 8.17. In a system where conflicting requests have fewer names in common, the performance benefit would not be as significant.

8.5.3 Varying the Switch Trie Size

We ran our experiments again, this time varying the number of tries from which the switch trie is built, from 100 to 10.000 tries. Figure 8.18 shows the performance results with locality factor 1.0. The performance is relatively slightly affected by this hundredfold increase in the size of the switch trie. Most intersection checks only compare between 2 and 10 blocks near the upper levels of the trie, that are near

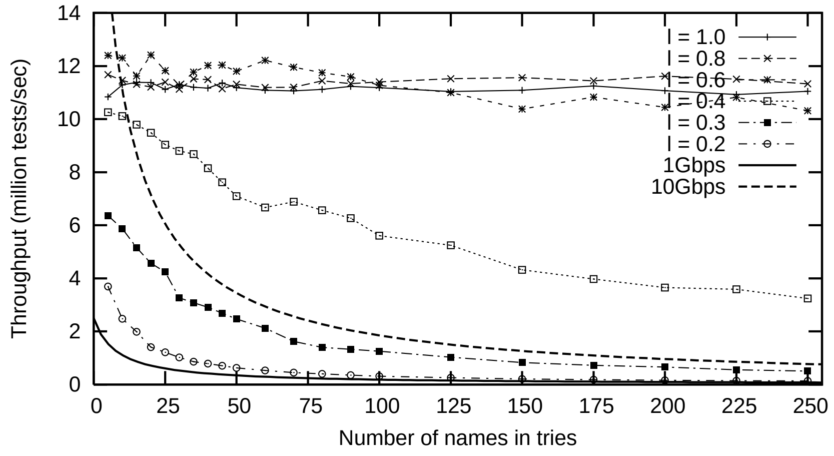


Figure 8.16: Performance as trie localities are varied, first-conflict only.

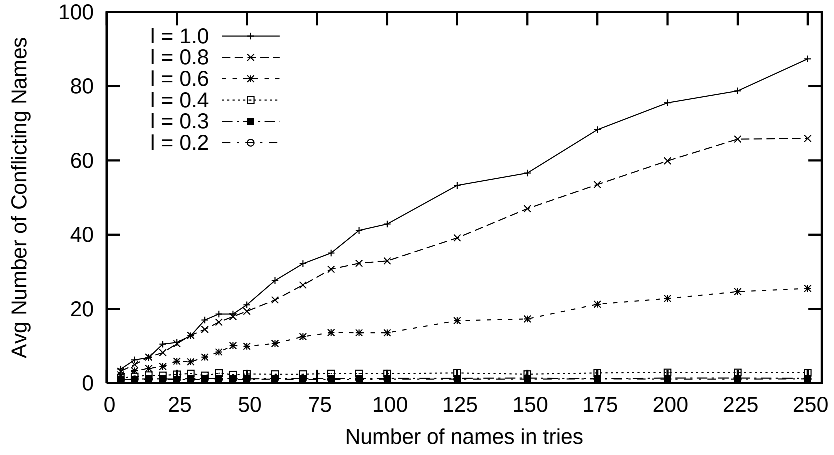


Figure 8.17: Average number of names in common to conflicting tries.

full density for all trie sizes (which is one of the reasons why it makes sense to use an initial array). As soon as the "stem" of a packet trie gets beyond these dense levels, the probability of it coinciding with switch tree stems decreases rapidly. With reference to the analysis in Section 8.4.1, the effort clearly grows closer to $\lg n$ than n.

8.5.4 Varying Conflict Ratios

In the final set of experiments, we vary conflict ratios, from 0 to 40%. We do it the same way as we controlled for conflict ratios in the other experiments, by scaling the

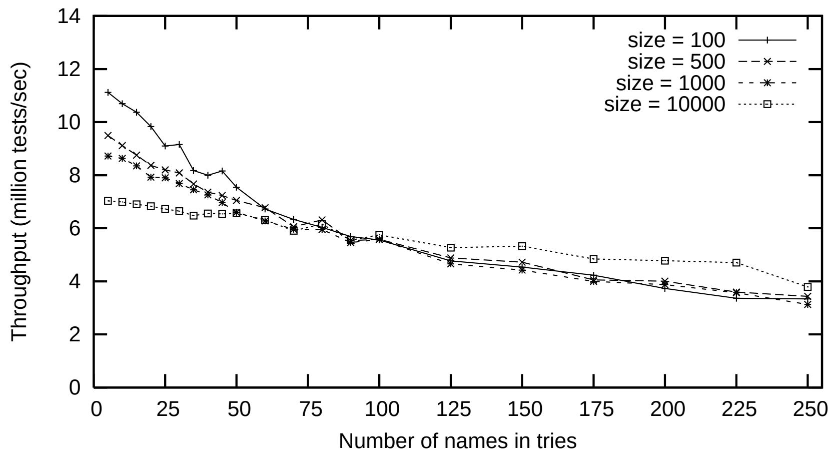


Figure 8.18: Performance as switch trie sizes varies.

block density factors of the data trie to increase or decrease the number of names in it, thus increasing or decreasing the probability of conflicts. Figure 8.19 shows the results for trie locality 1.0. The performance is quite sensitive to conflict ratios, due to the higher number of blocks that must be compared, as Figure 8.20 shows. This poses a problem; if routers are to help protect servers against contention overload, they must be able to filter out conflicting requests rapidly enough. A server that is becoming overloaded due to high conflict ratios might simply drop requests whose processing exceeds some bound, assuming that the request probably conflicts. But again, this graph represents the worst-case scenarios, as those requests that conflict have many names in common and the intersection routine is set to find them all. Figure 8.21 shows the best-case scenario, where the conflict routine stops after finding the first conflict. Real-world scenarios probably fall somewhere in between these extremes.

8.6 Possible Optimizations

This section discusses some optimizations that might enhance the performance of conflict checking and atomic switches.

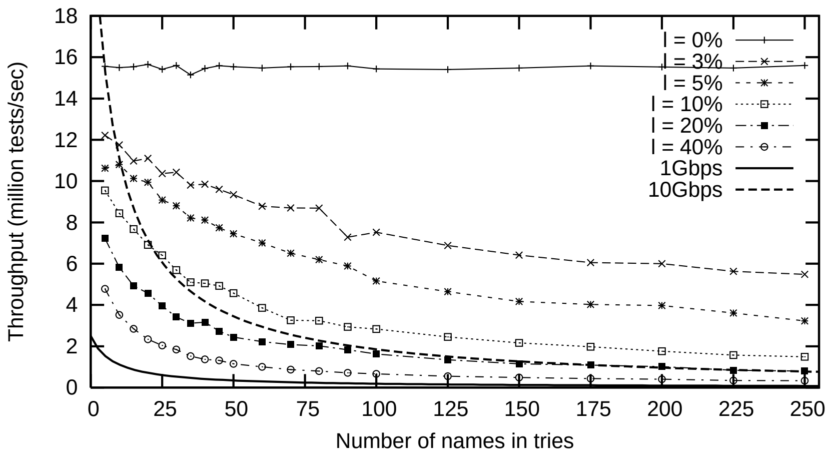


Figure 8.19: Performance as conflict ratios vary (all conflicts).

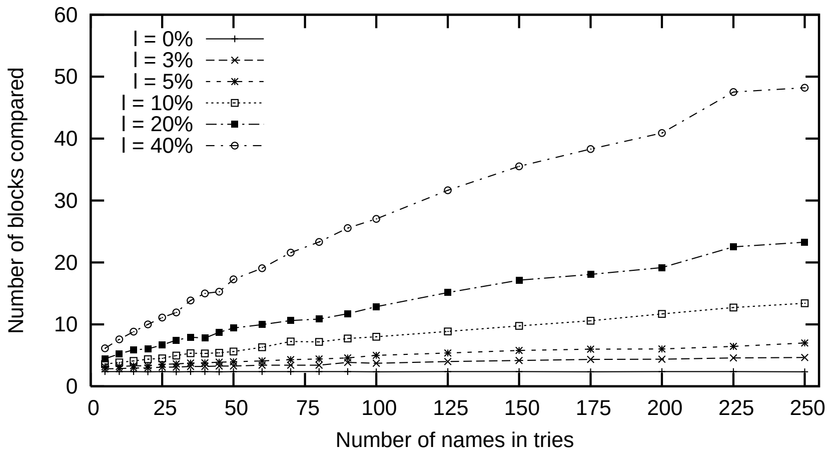


Figure 8.20: Number of blocks compared as conflict ratios vary.

8.6.1 Combining detection with Forwarding

It may be possible to amortize the cost of traversing the switch trie by storing forwarding information in it, looking up next-hop information while detecting conflicts. For example, a tree switch-block could have associated with it the set of ports subscribing to requests or responses, respectively, that contain a name beginning with the block's prefix. After all, this is essentially how Tree Bitmaps are used for high-performance IP lookup; a block is associated with next-hop information for the prefixes terminating in the block. Even if the switch's entire multicast subscription database does

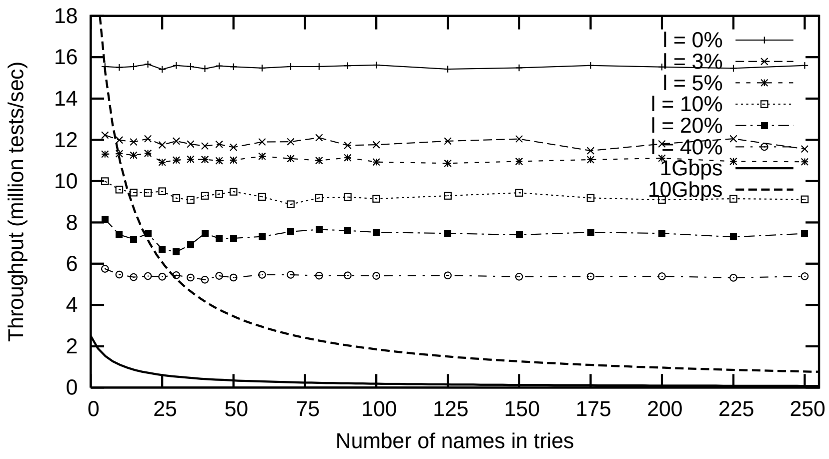


Figure 8.21: Performance as conflict ratios vary (first conflict only).

not fit into the switch trie, parts of it could be cached there since switch trie entries correspond to “active” prefixes for which packets have recently been forwarded.

While this is an attractive idea, it must be judged in the context of the overall multicast subscription management design. But within an autonomous subnetwork, where the naming hierarchy can be made to correlate with the network topology and forwarding tables can be kept compact, combining detection with forwarding could provide conflict checking with minimal additional cost, at least in the case of high-locality, non-conflicting requests. In Chapter 5 we do indeed model fine-grained multicast subscription as a set-intersection problem, that is: the switch must determine whether the intersection of a channel's subscription and a packet is non-empty. switch-blocks could possibly serve the dual purposes of storing subscription information and determining forwarding ports as well as detecting conflicts, increasing performance while reducing implementation complexity.

8.6.2 Hardware, TCAMs and Parallelism

The operation of comparing two TIB-blocks to find conflicts and/or the list of common child prefixes may be amenable to hardware implementation, in an FPGA (Field Programmable Gate Array) or ASIC, for example. A hardware implementation could potentially perform a block comparison in a few cycles. We observe that Tree Bitmaps have been implemented in FPGAs as well as an ASIC in a commercial router [178, 168], achieving excellent lookup performance bounded mainly by memory bandwidth and latency. We anticipate that decoding the TIB-block list-structure may pose a challenge, since it is geared towards iterative processing rather than combinational logic.

Another way to boost conflict checking performance would be to utilize the Ternary Content-Addressable Memories (TCAMs) available in many Network Processor architectures, such as the Intel IXP [112]. A TCAM is an associative memory supporting constant-time lookup of the value associated with a key. The sizes of keys and values can range from dozens to hundreds of bits and TCAM capacities can range from thousands to hundreds of thousands of key/value entries. A TCAM achieves constant-time lookup by searching its entries in parallel, using multiple comparator circuits. This enables TCAMs to sustain tens to hundreds of millions of lookups per second. A bitmask specifying *don't-care* bits is included with each key, permitting matching on a subset of a key's bits. If multiple entries match a search key, the highest-priority entry is returned. While TCAMs were designed for fast next-hop lookup of variable-length IP address prefixes, their function is quite general and TCAMs have been used to accelerate other types of packet classification tasks [187]. Their main drawbacks are high cost and energy consumption, as compared to ordinary SRAMs.

An straightforward use of a TCAM would be to store an initial-array in it. One could map each n-bit prefix present in the switch-trie to the TIB-block continuing that prefix. While up to 2^n TCAM entries might be needed in theory, only a few hundred or thousands of entries would be needed in practice, depending on switch-trie sizes and prefix distributions. In the best case when none of an incoming request packet's n-bit prefixes are present in the TCAM, the packet can be forwarded without stalling for switch-trie memory fetches. Furthermore, conflict-free response packets can be added to TCAM without stores to switch-trie memory.

This simple approach is effective when the n-bit prefixes of packet names are uniformly distributed across the space of n-bit prefixes. But as data contention increases[25], a larger proportion of incoming packets will have an n-bit prefix match in the TCAM. Also, using the TCAM as a high-speed memory does not take full advantage of its capabilities. We now sketch a different solution that addresses the issue of biased

[25]Or if the name space is biased towards a relatively small set of n-bit prefixes.

names, using don't-care bits for longest prefix matching. It essentially uses the TCAM as an index into the switch-trie, enabling many conflict-checks to skip over its topmost levels.

As an incoming response packet p is processed, a *covering set* of prefixes from p are chosen for insertion into the TCAM, that is: a set of non-overlapping prefixes K_p such that every name in p has some member of K_p as a prefix. The heuristics for choosing the prefixes could be something as simple as adding the first few extended prefixes encountered in p in a depth-first traversal[26]. The heuristics must be deterministic, though, as the packet will be traversed again to remove it from the TCAM and switch-trie, upon reception of its acknowledgement.

Prefixes are kept sorted by length in the TCAM from longest to shortest, so a single TCAM lookup returns the longest matching prefix. A simple way to allow fast insertion of prefixes in proper order is to partition the TCAM entries into "blocks", corresponding to different prefix lengths. Since name lengths are multiples of four, $m/4$ blocks would be needed for prefixes of length up to m, e.g. 20 for prefixes of length up to 80 bits. The TCAM is kept prefix-free like the switch-trie itself, that is: no entry in the TCAM is the prefix of another entry in the TCAM or switch-trie. The value stored into the TCAM for a prefix key is a trie-block, containing a member bit-set and the addresses of child-blocks in SRAM and/or DRAM.

Let p be an incoming response packet. For each prefix k from K_p, the switch looks k up in the TCAM. The main cases that can arise are:

1. There is no match. In this case, the switch trie has no names with k as a prefix, and the conflict-checking of k is complete. Prefix k is added to the TCAM, having as its value a trie-block pointing to the continuation of k in p's DRAM packet memory.

2. A matching prefix k_T is found, where the length of k_T is no greater than the length of k. In this case, the conflict-checking of k "backs off" to k_T and

[26]Applications could give "hints" by preferentially extension-encoding long top-level prefixes, and/or those known to experience contention.

continues with the trie-block of k_T. It may subsequently proceed to read switch-trie SRAM blocks and ultimately, packet DRAM blocks. At the end, if k turns out to be non-conflicting, it is added to TCAM, as in the first case.

3. A matching prefix k_T is found, where the length of k_T is greater than the length of k. In this case, k cannot be added to the TCAM, since it is a prefix of k_T. The lookup process therefore continues separately for each prefix $k \cdot k'$ of p, where k' is a child-prefix of k. Ultimately, these extensions of k will match an entry of equal length or have no match, leading to case 1 or 2 here above.

Request packets are processed similarly, except they are not added to the switch-trie and processing terminates immediately upon discovery of a conflict.

A single TCAM match can potentially skip a substantial number of trie-block comparisons. A 64-bit long match, for example, would skip sixteen 4-bit comparison steps, potentially obviating the need for several SRAM and DRAM reads. The TCAM optimization is more robust to biased name spaces and long names than a (reasonably sized) initial-array, as a greater number of prefix bits are be used to discriminate between conflicts and non-conflicts.

Since only prefixes of conflict-free names are added to the TCAM, a match on a prefix p implies that no TCAM entry for any proper prefix of p points to a switch-trie subtree containing names with prefix p. Note however that the TCAM entry for p may be shared by multiple response packets, whose prefixes continue in the switch-trie block pointed to by p and ultimately separate into non-overlapping branches. A TCAM entry is only removed once its bitmap becomes empty, meaning the entry no longer points to any trie-blocks. Hence, “hot” prefixes present in many responses tend to stay “cached” in TCAM, accelerating conflict-checking on these prefixes.

The TCAM optimization may be particularly effective in the case where acceleration is most needed: for hot prefixes affected mainly by small response packets, containing less than 10-20 names. Most of the names from such packets may be stored completely in TCAM, with long prefixes stored for the rest. Hence, many trie branches of incoming request packets can be processed entirely through TCAM lookups, with the rest requiring few memory fetches.

As a final, more general note, switches are inherently concurrent devices and conflict-checking is an inherently parallel process. Packets bound for different ports can obviously be conflict-checked in parallel, with multiprocessor cores assigned to sets of ports, e.g. Multiple messages bound for the same port can also be checked in parallel, although this requires careful coordination to ensure the conflict checking fulfills the AtomicSwitch automaton specification. Furthermore, a particular trie can be conflict-checked in parallel; when two blocks have multiple prefixes in common, each prefix can be processed independently of the others. They might be processed concurrently by different thread contexts on CPUs supporting hardware multi-threading, for example. Hence, there is considerable scope to parallelize conflict checking to scale its performance.

8.7 Discussion and Related Work

This section discusses related work on tries, as well as issues related to generalizing TIBs to other operations and conflict relations.

8.7.1 Related Work on Tries

Two well-known compressed trie variants are the Patricia Trie [188] and the Ternary Search Trie [189]. A trie can be converted into a Patricia Trie by removing each single child from the trie and appending its character to the label on its parent edge. Hence, edges are labeled with variable-length character strings, not characters, reducing the number of nodes. A Ternary Search Tree (TST) combines aspects of binary search trees with tries. A TST node's left (right) pointer points to a sub-TST continuing all strings that are lexicographically before (after) the node's prefix, while its middle pointer points to a sub-TST containing all strings that begin with the node's prefix.

Bagwell [190] describes a data structure similar to Tree Bitmaps, dubbing it an "Array Mapped Tree" and tracing the idea's lineage to Bird in 1977 [191]. Guy Jacobson [192] describes an asymptotically optimal way to encode static binary trees, using bit sequences to encode absence and presence of child nodes. The Shape Shifting Trie

[193] is an elegant trie variant based on Jacobson's encoding, where the underlying "shape" of a block is not a fixed, full binary trie but can vary according to the shape of the trie being encoded. This allows longer stems to be encoded within a single block. That data structure is geared explicitly towards hardware implementation, though, and intersections cannot be directly computed with bitwise AND unless the shapes of the underlying blocks conform.

8.7.2 Generalizing to Arbitrary Operations

The primary objective of our experimental evaluation was to get some empirical data about the set-intersection performance levels achievable with a Tree Bitmap-like data structure. But to use TIBs or some other suitable data structure for conflict checking in practice, the repertoire of operations must be expandable beyond simple reads and writes.

This issue could be addressed on an ad-hoc basis within isolated subnetworks, or Metanetwork [154], with application-specific packet formats installed into switches. However, the basic function of conflict detection is remains basically the same, regardless of application domain. Still generality must be weighed against performance. Given the performance constraints[27], it seems likely that switch conflict checking software would be a fixed, compiled binary, upgraded only as a part of system evolution. This may limit the number of available ADTs in an atomic network to a few dozen or hundreds, at best. On the other hand, availability of a good set of basic abstractions such as numbers, strings, lists, maps, queues, sets, objects and trees, for example, could facilitate interoperability between atomic applications.

Our experiments needed only a single bit to store operation information: whether the operations was a read or a read/write. For other ADTs, more operations may be available and some conflict relations may take operation arguments or return values into account. However, TIB-blocks can only accommodate so many bits in its list, e.g. up to 8 bits per entry. A reasonable compromise might be to split conflict detection into two parts: a detection based on up to 2^7 coarse-granularity operation conflict classes, detecting a superset of all conflicts. A conflict detected at this coarse level

[27]And in some system environments: code size constraints.

would lead to a second stage of conflict checking, using more detailed information about the operations in question. We note that the overflow block system can be easily extended to allow chaining of overflow blocks, supporting variable-length data in a relatively efficient way. Aso, a TCAM could possibly be used to quickly look up the conflict properties of a pair of operations. But the present dissertation does not further investigate these issue.

Chapter 9

Future Work

This section discusses future directions and extensions of the work presented in this dissertation, some of which are already in progress and others that are more speculative in nature. While concurrency control is one of the pillars of distributed atomic execution, it also rests on two other foundations: *persistent recoverability* and *atomic commit*.

Persistent recoverability refers to the fact that in an atomic execution a client should only receive an operation's response if the operation's effects have been persistently committed to the system's state. However, the persisting of operation effects in (slow, non-volatile) *stable storage* is often at odds with the desire to send back operation responses as quickly as possible. A common practical approach is to append operation records to a persistent *log* upon execution, while persisting their state effects in a background process. When a server recovers after a crash, it reconciles its log with its persistent state, completing the effects of committed operations while undoing the effects of aborted ones. Nonetheless, the logging delay can significantly increase response times in systems where persisting a log record takes a long time as compared to executing a request.

Atomic commit refers to the fact that operations must be atomic even if they span multiple servers. While this dissertation is intentionally restricted to systems where each request is executed on a single server host, transactional systems must generally be able to execute operations involving multiple hosts. An *atomic commit protocol* is required to ensure that a request is either successfully committed at each host it involves or else that it is aborted at each host. Atomic commit is an inherently expensive algorithm, as it generally requires at least two message rounds as well as locking of affected state while a request's fate is being determined. This determination is usually performed by a designated host called the request's *coordinator*. Unfortunately, failure of the coordinator can lead to the request being *hung* indefinitely, getting neither committed nor aborted.

This chapter sketches ideas for dealing with recoverability by using switch packet memory as stable storage, removing logging delays. It then sketches a new atomic commitment protocol that uses Atomic Transfer to achieve "stateless coordination", where switches act as coordinators for atomic commit without retaining any state for requests. The protocol assumes and retains the property that requests are addressed to variable names instead of servers and is furthermore transparent to requesters.

The chapter then sketches some ideas for optimizations aiming to further lower request response-times, motivated mainly by the desire to lower the rate of concurrency-control conflicts in cached systems such as those of Chapters 4, 6 and 7. These optimizations exploit the guarantees of Atomic Networks for aggressive, speculative execution that might otherwise not be practical. Finally, the chapter briefly discusses security and replication issues, which have otherwise been left outside scope.

9.1 Resilient Transfer

As outlined in the introduction, transactional systems commonly use operation logging to persist the execution of an operation before transmitting its response. If the response were sent any earlier and the server crashed between sending the response and persisting its effect, clients might erroneously receive the response for an operation that apparently never executed. Logging is commonly performed by *forcing* a log record to disk, waiting until the record has been persistently written to the disk's platter surface. This can be a significant source of latency, as even the fastest disks take several milliseconds to complete a write operation. Solid-state disks (SSDs) promise to lower this latency to tens or hundreds of microseconds, reducing the impact of logging delay. Yet, this corresponds to hundreds of thousands of processor cycles, so the delay remains significant in many cases.

There have been proposals for redundantly storing data in multiple volatile memory banks [127, 128] to create fast[28] stable storage. If banks fail independently and are backed by a battery and/or uninterruptible power supplies, the probability of all banks simultaneously losing their contents can be made arbitrarily small. One scheme would be for server hosts send their log records to other hosts and regard their records as stable upon receiving f acknowledgements of receipts, where f is the number of failed banks that should be tolerated. Still, this may result in longer delays than SSD-based logging, as sending and receiving logging messages and ACKs can take tens or hundreds of microseconds, besides consuming host resources.

Our observation is that the switches on a network path "naturally" create redundant copies of requests as they store and forward them. Hence, all that is required for switches to serve as redundant memory banks is for each switch to buffer each request packet until it knows that the f-th next-hop switch has received it. This ensures that a request is buffered on at least f switches at any instant in time. In the scheme we detail in the next section, a server can execute a request and send a response immediately without waiting for a log record to be written. Once the log record has been written in the background, the responder acknowledges the request to the switches, enabling them to purge their copies of it. If the responder fails before writing the log record or sending an ACK, the switches retain their copy of the request until they succeed in retransmitting it. Once the responder recovers, any non-acknowledged requests are thus *replayed* to the responder, allowing it to rebuild the state that existed following its last acknowledged request. Even if up to $f - 1$ consecutive switches on a forwarding path fail simultaneously, the non-faulty switch preceding that segment of the path will retransmit any lost packets, ensuring eventual delivery to the f-th switch on the path as long as the network's topology remains constant.

9.1.1 Resilient Transfer protocol sketch

This section sketches a protocol for fault-tolerant *resilient transfer* (RT) in some detail. The protocol adds limited overhead to switches, apart from increasing the amount of memory a switch requires to buffer packets. That amount increases roughly

[28]A store to DRAM can complete in tens of nanoseconds.

by a factor of f, where f is the maximum number of simultaneously failed switches tolerated. Since switches are generally designed for high reliability, $f = 2$ or $f = 3$ should suffice for most applications.

The core idea is that an ACK from a next-hop switch i acknowledges not only the reception of packets by i but also the reception of packets by switches up to $f - 1$ hops beyond i. Instead of sending a single sequence number a acknowledging receipt of all packets up to and including a, switch i sends an f-tuple $A = (a_0, a_1, \ldots, a_{f-1})$ of sequence numbers where a_n acknowledges that all switches n hops beyond i have received all packets up to and including a_n, for $n \in [0, f - 1]$. Note that a_n is a lower bound, as some switches n hops removed may have received packets with sequence numbers exceeding a_n.

Each packet p has encoded in it a *fault tolerance* $f_p \in [0, f]$, indicating the number of failed switches the packet's transmission must tolerate. Since fault-tolerance is assigned on a packet-by-packet basis, a resilient network can carry a mixture of unreliable ($f_p = 0$), reliable ($f_p = 1$, corresponding to hop-by-hop reliable transmission) and fault-tolerant ($f_p > 1$) packets. Let $\mathit{sseq}(p)_d$ denote the send sequence number of a packet p enqueued on channel $d \in CHANNELS_i$, that is: the number of packets enqueued ahead of p on d. A switch buffers a packet p enqueued on d until it receives an ACK tuple A via d where $a_{f_p} \geq \mathit{sseq}(p)_d$, since then it knows that the packet is buffered on the next f_p switches on every path that p is forwarded along.

Let $\mathit{rseq}(p)$ denote the reception sequence number of a packet p received on some channel $c \in CHANNELS_i$, that is: the number of packets received ahead of p on c. In a typical hop-by-hop reliable scheme, a switch would periodically send the latest rseq back on c in an ACK, to acknowledge the reception of packets. In the RT protocol, i must also infer and send back the corresponding rseqs of the $f - 1$ next-hop switches to which packets from c have been forwarded. To do this, i must "translate" incoming ACK rseqs from those channels to outgoing ACK rseqs for c.

Switch i can do this by tracking for each packet p enqueued onto a channel d the channel $c = \mathit{rcvChan}(p)$ on which p was received, as well as $\mathit{rseq}(p)$ and $\mathit{sseq}(p)_d$. It maintains for each pair of channels $c, d \in CHANNELS_i$ and fault-tolerance level $k \in [1, f]$ the latest (greatest) rseq number of a packet received on c and forwarded on d that has been acknowledged by all switches k hops removed from d, which we denote by $\#acked_{c,d,k}$. This information is updated as new acknowledgements are received. For example, if an acknowledgement arriving from a channel d allows a packet p with $p_f = 2$ to be purged, then $\#acked_{c,d,2}$ is increased to $rseq(p)$, where $c = rcvChan(p)$. When switch i sends an ACK $A = (a_0, \ldots, a_{f-1})$ on c, it uses as the value for a_k the lowest acknowledged sequence number among the channels forwarding to c, that is: $a_k = min(\ \{\#acked_{c,d,k} \mid d \in CHANNELS_i\}\ )$.

A problem can occurs, if packets temporarily cease to flow from some channel d' to c, halting the flow of ACKs for packets enqueued on d'. Then the ACKs sent on c will be stuck at the *rseq* of d', while $\#acked_{c,d,k}$ grows for $d \neq d'$, leading to unbounded buffering. To remedy this, switch i maintains a counter $\#rsMax_{c,d,k}$, containing the *rseq* of the latest packet from c with fault-tolerance k enqueued on d. Hence, $\#rsMax_{c,d,k}$ - $\#acked_{c,d,k}$ is the number of packets with redundancy degree k forwarded from c to d that have yet to be acknowledged back via d. When that number is zero, no packet from c with redundancy degree k needs to be retained for d, so we omit $\#acked_{c,d,k}$ from the computation of a_k. In the case where it is zero for all channels, we define a_k as $\#rseq_c$, the highest reception sequence number of any packet received on c[29]. Formally, we define the ACK A that should be sent on channel c as:

Definition 9.1 *For any $c \in CHANNELS_i$ on a switch i, the current ACK for c is the tuple $A = (a_0, a_1, \ldots, a_{f-1})$ where for each $k \in [0, f-1] : a_k = min(\ \{\#acked_{c,d,k} \mid d \in CHANNELS_i \wedge \#acked_{c,d,k} < \#rsMax_{c,d,k}\} \cup \{\#rseq_c\}\ )$.*

We must account for the fact that switches sometimes send their own packets, without any corresponding packet having been received, for example their own ACKs. Any such packet p is enqueued as having no associated channel and no *rseq*. It may have an arbitrary fault-tolerance and can be reliably transferred like other packets, but is ignored for the purposes of updating $\#acked_{c,d,k}$ fields.

Clearly, A is an underestimation of the number of packets buffered on neighboring switches, so switches generally buffer packets slightly longer than they have to. The overshoot increases linearly with the fault-tolerance level f, and the worst case occurs

[29]Here we mean the highest sequence number of a packet received in-order and without sequence number gaps.

when one of the paths to which a channel's packet are forwarded has a significantly slower transmission rate than the other paths. However, such rate differences would lead to overflow in any reliable transmission scheme, so RT does not introduce a new problem in that regard.

A host $a \in HOSTS$ also maintains send and receive counters, but sends ACKs of the special form $(\#\mathit{rseq}, \infty, ..., \infty)$, that is: a acts as if each received packet had been acknowledged by all "switches" on the non-existent path extending beyond a. This ensures that switches can purge packets that a has acknowledged, regardless of the packet redundancy degree or network path lengths. Hosts retransmit and purge outbound packets in a similar manner as switches do.

9.1.2 Recoverability through Resilient Transfer

RT is not intended as a general alternative to end-to-end reliable transfer, but rather as a mechanism for end-host recoverability without logging delays. The basic idea is that hosts store their send and receive packet counters, and use them to recover after failures and ensure exactly-once semantics for request execution. Switches maintain send and receive sequence counters persistently "forever", obtaining them back from neighboring switches upon recovering from failures. Hence, these sequences can be used as a part of end-host recovery.

A server host b executes requests and sends responses immediately, without waiting for a log record to be persisted. Host b persists log-records in the background, including with each record the *rseq* sequence number(s) of the request packet(s) causing the response. Crucially, b only acknowledges a request after b has persisted the corresponding log-record. Since request response-times are now independent of logging delay, b can persist groups of log records together, amortizing the write operation's latency over many records [30]. Upon recovering from a failure, the home switch i of b retransmits a superset of the requests that b had yet to persist at the time of failure. Host b recovers by replaying its log to obtain the state following the last request packet p present in the log, updating its *rseq* to $n = \mathit{rseq}(p)$ in the process. It then

[30] As log records are relatively small in general while the bandwidth of block-oriented storage devices is usually maximized for large, sequential writes, logging should be able to keep up with server throughput increases.

(re)executes received requests with receive-sequence numbers exceeding n. Switch i detects and drops any duplicate responses (those with sequence numbers less than i's *rseq* for the channel connecting i to b) so the first response forwarded onto the network represents the "continuation" of b's execution beyond the point where it failed. We note that b may alternatively skip logging completely, proceeding to directly update its persistent state, as long as it can accurately track the progress of these updates [125]. In any case, average response times can be reduced by a factor of up to ϕ, where ϕ is the fraction of request response times due to waiting for persisting of log record.

Our scheme, like any recovery scheme, is complicated by non-determinism in request executions, caused by concurrency and scheduling decisions, for example. The sequence of requests executed by a recovering server, in particular, must contain those already sent as a prefix. Non-determinism can be handled by allowing servers to include a *determinant* [194] in a response message r, that is: information that suffices for the server to reconstruct any non-deterministic choices it made as it originally executed q, the corresponding request. The determinant for q is carried with r up to f hops into the network[31]. When retransmitting requests, a switch includes any determinants received with responses to those requests. Upon recovery, a server therefore receives its determinants back with any retransmitted requests, allowing it to re-create their execution exactly. Determinants are garbage-collected on switches as the corresponding request packets are purged.

As an aside, a requester requiring exactly-once semantics for its requests (such as a work-flow processor atomically dequeuing and executing tasks from a task queue) could perform server-like recovery by including itself as a responder for its exactly-once requests, turning them into multi-server request[32] (see Section 9.2 here below), although we do not develop this idea further here.

A failed switch recovers by receiving re-transmitted packets from its neighbors, including sequence numbers. Neighboring switches detect and drop any duplicate packets forwarded by the recovering switch during this period. Hence, it is possible to avoid

[31]The determinant could also include *undo* information, to support undo/redo recovery [125]

[32]We prefer the term *multi-server transaction* instead of "distributed transaction", as single-server transactions are distributed despite being handled by a single server.

packet loss despite the simultaneous failure of up to f consecutive switches on a network path. Observe, though, that switches are inherently non-deterministic due to their scheduling of multiple packet flows onto an outbound channel. The order of packets sent can be re-created with the help of determinants. We leave these details to future work.

Our scheme is related to work on *replay recovery* for systems comprised of distributed processes sending and receiving messages [194, 195, 196]. However, these schemes are very general, dealing with arbitrary networks of stateful nodes performing arbitrary computations. They are therefore more complex and more expensive in terms of bandwidth and processing than our network and persistent-state-centric scheme. For example, processes must sometimes send auxiliary messages for recovery purposes, since redundancy is only achieved when multiple process nodes buffer messages. By contrast, our scheme relies on the "natural" redundancy occurring as switches forward packets, and assumes the specifics of store-and-forward processing, resulting in lower overhead on switches and hosts. The actual overhead is best evaluated using a prototype implementation, but this awaits future work.

As a final note, our scheme has some security implications. For example, a crashed server relies on switches to hold on to its non-persisted requests until the server recovers. This may be acceptable in practice, as the server chiefly depends on the last f switches on its inbound path(s) to store the requests, and one or more of these switches are likely a part of the server's trust domain. Also, all f switches would have to renege on their duty to store the request for it to be lost. We don't claim tolerance to Byzantine, arbitrary failures, but these issues would have to be considered in a robust implementation.

9.2 Multi-server Requests

As mentioned in the chapter's introduction, this dissertation limits its scope to single-server transactions, for simplicity and to keep separate the issues related to atomic commit. However, scalable systems need to able to execute atomic transactions across multiple hosts. While data name spaces can and should be designed to minimize the need for such transactions, they often cannot be avoided altogether.

This section sketches a novel approach for using atomic switches to accelerate the commit of multi-server transactions, showing that Atomic Transfer combines with atomic commit. In our approach, switches function as commit coordinators but in a stateless manner, exploiting name-based packet forwarding and atomic transfer to distribute most of the coordinating duties among network switches and server end-hosts. But first we must consider the issue of multi-packet requests and responses, which arises in practice even with single-server transactions.

9.2.1 Multi-packet Messages

The models of the preceding chapters assume that requests and response messages are forwarded as atomic units, effectively corresponding to network packets. In practice, some requests and responses may be too large to fit in a single packet of maximum size, and must therefore be broken up into multiple packets. A simple way to process such a *packetized* message m would be for each switch to buffer the packets of m and re-assemble m in memory before processing it. However, this could significantly increase latency, implementation complexity and the amount of packet buffer space needed on switches, in addition to placing a de facto limit on the maximum size of messages.

It is more attractive to retain packet-by-packet processing on switches. This is possible, by placing the following restriction on the way requests and responses are split into packets: if a request $q \in Q$ and response $r \in R$ where $\mathit{conflicts}(q, r)$ are split into respective sets of packets PQ and PR, then there exists a pair of packets $p_q \in PQ, p_r \in RQ$ such that $\mathit{conflicts}(p_q, p_r)$, where the domain of *conflicts* has been extended to packets. This ensures that if messages q and r cross in the network, packets p_q and q_r will cross too and be detected as a conflict. The naming scheme of Chapter 5 can easily be seen to have this property, if each name/operation pair of each packetized request or response can be decoded from a distinct packet.

Message name tries could be broken up into packets by introducing a new type of entry list code "11" in the TIB format of Chapter 8, denoting a prefix that continues in another packet. This encoding also enables a host to know when it has received its part of a request or response: when all such *continuation prefixes* have been "filled in" by sub-tries from other packets.

While packetization preserves the correctness of Atomic Transfer, it reduces the effectiveness of switches in shielding end-hosts from receiving conflicting request packets, as it cannot be known that two messages conflict until their first pair of conflicting packets is detected. In the worst case, where conflicting messages always have a single pair of conflicting packets, roughly half of the packets of a conflicting request q can be expected to be forwarded through the switch that detects the conflict. The rest of the packets of q, however, can be dropped by having the switch send a *request cancelation packet* c_q towards the sender of q, defined to conflict with all packets of q. As a compensating factor, multi-request packets will generally be large and therefore relatively cheap to receive and conflict-check on a per-byte basis, as indicated by the experiments of Chapter 8. Also, Section 9.3.1 discusses how conflict detection can be shifted from hosts to switches completely.

If the conflicting packet of q is not the first packet of q then a copy of it might be forwarded on to the responder, to notify about the dropping of q and prompt it to purge the packets of q it has already received. This is only be an optimization, though, as responders would time out incomplete requests to handle failed requesters, anyway.

We leave to future work the extension of the results of Section 3.7.1 to packetized messages, to evaluate the percentage of conflicting packets from packetized messages that can be expected to be detected on end-hosts as a function of contention. As a practical heuristic, though, we note that if requesters know which names are most likely to be contended, they can encode their operations for those names in the first packet(s) of any request involving the names.

9.2.2 Handling Multi-server Requests

The processing of multi-server requests must preserve two properties that trivially hold for serializable executions of single-server requests in our model:

1. *Global consistency*, meaning that a request is either committed at each of its *cohort* servers where it executes or committed at none of them. It cannot be committed at one cohort and aborted at another, for example.
2. *Global serializability*, meaning that there must exist a *global serialization order* for all committed transactions in a system of hosts, that is compatible with the *local serialization order* at each host.

As is well known, local serializability at each host does not imply global serializability across hosts, as two transactions may be ordered differently on different hosts. Consider, for example, mutually conflicting read/write requests q_1: "$a = 1, b := 2$" and q_2: "$b = 1, a := 2$". The first request assumes that a has value 1 and assigns value 2 to b, while the latter assumes b equals 1 and assigns 2 to a. If $host(a) \neq host(b)$ then $host(a)$ might receive and execute sub-requests "$a = 1$" of q_1 and "$a := 2$" of q_2 in that order, while $host(b)$ might receive and execute sub-request "$b = 1$" of q_2 and "$b := 2$" of q_1 in that order. Although the requests conflict globally and no global serialization order can include both requests, the servers cannot observe the conflict based on their local information.

The following two sections sketch approaches to ensuring global consistency and serializability for multi-server requests. These approaches preserve the naming abstraction of our models, and a request is issued the same way regardless of whether it will be served by one or multiple servers. A requester plays no role in the atomic commit protocol and remains oblivious to the mapping of names to hosts, promoting flexibility in the allocation of data to servers and load-balancing, etc.

9.2.3 Global Consistency with Atomic Transfer

Atomic commit of a multi-server request q generally requires a two-phase commit (2PC) [2, 6], which works roughly as follows. In the *prepare phase*, each cohort of q receives and executes its sub-request of q and sends back a *prepare message* for q to every other cohort, signalling its readiness to commit q. Upon learning that all the other cohorts of q have prepared q, a prepared cohort enters its *commit phase*, irrevocably committing the effects of its sub-request of q and sending a response. On the other hand, if some cohort sends an *abort message* in lieu of a prepare to indicate that it cannot process its part of q, then all cohorts abort q, ensuring that q has no effect on the state of any cohort.

Ensuring consistency is a challenge with failure-prone servers and communication channels, as the cohorts must agree whether to commit or abort the request even as some of them are crashed or partitioned. The standard solution is to let a central *co-ordinator* host oversee the 2PC processing and make the final abort/commit decision. The cohorts send their prepare or abort messages to the coordinator, which decides the request's fate and communicates its decision back to the cohorts in a commit or abort message. Since the central coordinator has full autonomy in making its decision (while the cohorts have none after sending their prepares) it can handle crashed cohorts and failed communication links by deciding to abort, after a time-out, for example.

Two-phase commit requires two rounds of messages between the coordinator and the cohorts, increasing request response time. Furthermore, after a cohort sends its prepare message for a request, the request is *in-doubt* at the cohort and any conflicting request must be delayed or dropped, pending the final commit or abort of the in-doubt request. Worse, if the coordinator crashes the cohorts remain blocked until the coordinator recovers and resumes processing. While there are non-blocking alternatives to 2PC [197], they add another round of messages, significantly increasing response times in the normal case.

Our proposed protocol does not fundamentally deviate from traditional 2PC, requiring two message rounds and blocking on coordinator failure. However, we attempt to minimize message latency and the probability of coordinator failure by deputizing the switch i_q that first splits a request q as the coordinator for q. Switch i_q stamps q with its identity, alerting subsequent switches that they are not the top-level coordinator for q. However, switch i_q retains no state for q and performs almost no special processing for multi-server requests beyond forwarding messages and performing Atomic Transfer, limiting additional switch complexity. Instead, the tracking of the progress of q is handled entirely by the cohorts. If a cohort suspects that progress has ceased due to another cohort failing, it *challenges* the request by sending a special packet to the coordinator switch. As described later, the switch resolves race conditions between a challenge and a final, missing prepare message using Atomic Transfer. Our protocol is interesting in that the cohorts communicate only in terms of names; they track neither the number nor identities of other cohorts.

We extend our model so that *qHop* is no longer a function, but rather a relation that forwards a request packet on each channel leading to a host of a name encoded in the packet. This somewhat complicates the definitions of conflict locality without fundamentally altering them. A server host b knows that the request q to which a packet p belongs is a multi-server request if at least one name or prefix in p (including continuation prefixes) is not mapped to b. In that case, b executes its part q_b of q only tentatively, but sends a *prepare* packet that contains prefixes that *cover* each name of q_b, indicating that the operations on those names are prepared. The prepare packet is sent back to coordinating switch i_q, which forwards it to the other cohorts. Once b has received a set of prepare packets covering the whole of $names(q)$, it knows that all other cohorts have prepared q and so commits q_b, sending out the response packet(s) for q_b.

Prepare packets could be routed back to coordinator switch i_q by addressing them to i_q directly (i_q having stamped its network identifier on the packet before forwarding it) or by sending packets "upwards" in hierarchical networks, e.g. The thornier question is how i_q can know where to forward prepare packets for q, and how a cohort b knows when all the names of q have been covered by a prepare, given that i_q maintains no state for q and that b only receives those packets of q that are relevant to b. A straightforward solution is for the issuer of a multi-packet request to always send a *survey packet* as the first packet of any request, containing continuation prefixes that cover q as far down q's name trie as there is space in a single packet. When host b is prepared, it converts the survey packet into a prepare packet by marking its own branch(es) of the packet's trie as prepared and sending it to i_q, which forwards / multicasts the packet normally to the other cohorts, as per the packet's prefixes. We note that the concise encoding of Chapter 8 can encode hundreds of prefixes in a single 1500-byte packet, but our method could be extended to allow multiple survey packets if needed.

A problem can occur if the survey is not "detailed" enough, so a switch would have to forward the survey packet based on a continuation prefix. The switch can preserve correctness in this case by aborting the request (see below), prompting the requester to retry with a more detailed survey. Alternatively, if the survey is being forwarded to a relatively small sub-network, the switch may simply flood the packet onto the subnetwork, which guarantees that all cohort hosts will receive it.

In the failure-free case, cohort b sends a prepare packet for its part of request q and adds prefixes from incoming prepare packets to the set of prefixes it knows to be prepared. Once all names of q are prepared, b commits q_b and sends response r_b, its part of the response r to q. On the other hand, if b decides to abort q instead of preparing it, it converts the survey packet into an abort packet and forwards to the other cohorts, via i_q. If b receives such an abort packet it aborts q_b immediately. Note that Atomic Transfer works the same as before, i.e. requests conflicting with r_b are dropped. If the conflicting request q' is a multi-server request detected below its coordinating switch $i_{q'}$, then q' might be forwarded on as an abort request, to expedite the aborting of q'.

The problem of crashed and partitioned cohorts is handled as follows. If a cohort b is dissatisfied with the progress of q, because b has not received any prepare messages for some duration of time, for example, it *challenges* q by converting the survey packet into a *challenge packet* p_c and sending to i_q. Note that it is *not* safe for b to send an abort packet, since b has already sent its prepare packet and some cohorts may already have committed q, so sending an abort could lead to violation of atomicity. Rather, p_c is converted into an abort packet p_a at i_q, but only if it is safe to do so. This requires handling the possible race condition between p_c and a remaining prepare packet p_p that may already be in transit. Our solution is to define challenge packets and prepare packets as being in conflict, so that prepare packets dominate challenge packets and cause them to be dropped. Hence, if a prepare packet p_p is received at i_q before p_c, then p_c is dropped and does not become an abort packet. On the other hand, if p_c is received before p_p, then p_c becomes an abort packet p_a that is forwarded to all cohorts. By defining aborts as conflicting with and dominating prepares, no cohort will receive p_p and safety is ensured. Note that this solution requires that a cohort b not commit a request before receiving its own prepare packet. Our scheme ensures that a request can be aborted and unblocked as long as a single cohort is alive and connected to the coordinating switch.

In traditional 2PC, failed and partitioned cohorts query the coordinator about the fate of a pending request q, upon recovering. This is not possible in our scheme, as the coordinating switch i_q does not maintain any state for q. Instead, a recovering cohort b sends a recovery packet to the other cohorts, which simply respond by retransmitting prepare or abort packets, as appropriate. Theoretically, this entails cohorts remembering the fate of each multi-server requests they execute "forever", since a recovering host may query about it at some arbitrary later time. In practice, this state (a bit per request) can be discarded using some garbage collection protocol or simply timed out, assuming an upper bound on the delay from when a server fails until it is brought back online or permanently removed.

It may appear that our protocol suffers from $O(n^2)$ message complexity, as each cohort sends it prepare to every other cohort. This compares unfavorably with the $O(n)$ complexity of sending n prepares to a stateful coordinator that sends n "commit" or "abort" messages back. However, since coordinating switch i_q multicasts messages, only a single prepare is received and sent over each network link in the forwarding tree of q (including the links incident to i_q) for each cohort, so from that perspective the complexity of our protocol is also $O(n)$, albeit with a higher constant. The number of round-trip message hops until a host can commit a request is also the same in both cases, but our protocol may yield an improvement in practice as switches can generally forward messages more rapidly than end-hosts. Furthermore, our protocol chooses coordinators efficiently, i.e. as close to the cohorts in the network as possible, localizing 2PC communication and reducing latency.

As stated before, this scheme blocks upon coordinator failure. However, relatively fixed-function switches can be engineered to achieve far higher availability than general application end-hosts, making coordinator blocking a commensurately rarer event. Furthermore, since the coordinator switch is stateless, it may be possible to devise a scheme whereupon a redundant backup switch takes over coordination duties for a failed switch, although we leave consideration of this to future work.

9.2.4 Global Serializability with Atomic Transfer

Atomic Transfer can be used to ensure local serializability of execution at a single responder host. However, as illustrated in by the example of Section 9.2.2, local serializability at each individual does not imply global serializability across all hosts, i.e. that there exists a sequential execution corresponding to the execution of all committed requests on all server hosts, including requests that span hosts. We propose two possible solutions for achieving global serialization for multi-server transactions: *bloated requests* and *gateway ordering*. The first is generally applicable, while the latter is more efficient but only applicable to certain network topologies, albeit common ones.

Global serialization violations stem from the fact that cohort servers of a request q have only a partial view of q; a cohort b only detects conflicts on variables that are hosted on b. A simple "brute-force" solution, therefore, is to ensure that each cohort receives entire requests, including the parts involving names not hosted on the cohort. This *bloating* of requests, as we term it, enables each cohort to detect all conflicts, the same as if requests were being sent to a single server. In the example of Section 9.2.2, if host a were to receive q_1: "$a = 1, b := 2$" first and q_2: "$b = 1, a := 2$" second, then a could detect the conflict on variable b even though b is not hosted on a, by performing conflict-checking processing on the whole of q_1 and q_2, including the sending of a response for a name not hosted on a! We note that such "bloated responses" do not have to be forwarded beyond the node splitting the corresponding request.

Bloating does lead to a higher cost in bandwidth and conflict-checking, as requests are sent in whole instead of getting split into sub-requests per cohort. However, bandwidth is an increasingly abundant resource [183], and the added conflict-checking burden is somewhat mitigated by the effectiveness of our conflict-checking methods and the network's participation in conflict detection. Less brute-force methods, such as having servers exchange explicit transaction dependency information, incur the more serious cost of additional message rounds

We face the problem of how to enable switches to forward all packets of a multi-server requests to each cohort, instead of forwarding to each cohort only the packets relevant to it, as dictated by the forwarding relations. One solution would be to use survey packets, as in Section 9.2.3, containing a high-level overview of the names of the packets yet to come. Upon receiving the survey, a switch i would remember a mapping $q \rightarrow C_q$ from (the identity of) q to the set $C_q \subset CHANNELS_i$ onto which prefixes from the survey packet of q may be forwarded, and forward any subsequent packets of q on the channels of C_q[33]. Switch i can discard the map entry for q after forwarding the final packet of q, or after a time-out, to handle failed requesters.

A cohort b receiving a bloated request q_2 conflict checks it with any other bloated request it has prepared. Upon detecting a conflict with another bloated request q_1, it could abort q_2 right away. However, this might lead to both requests getting aborted, if some other cohort receives q_2 first and aborts q_1. In the worst case, livelock may ensue, with the respective requesters repeatedly resubmitting the requests only to have them be aborted.

We resolve the issue as follows. Let b be a server receiving a pair of mutually conflicting bloated requests q_1 and q_2, say in that order. Rather than immediately aborting q_2, b computes the relative order of q_1 and q_2 according to some total order $<_Q$ on requests. An implementation could use the integer value of the (cryptographic) digests of the first packet of the two requests, for example. If $q_1 <_Q q_2$ then b immediately aborts q_2. On the other hand, if $q_2 <_Q q_1$ then b waits, delaying q_2 from getting committed. If q_1 is received before q_2 on all servers that receive both requests, then b will receive the prepares required to commit q_1, leading it to abort q_2. Otherwise, q_2 is received before q_1 on at least one server, which will immediately send an abort packet for q_1 since $q_2 <_Q q_1$. Upon receiving that abort, b aborts q_1 while sending its prepare for q_2, in effect resuming the 2PC processing of q_2. This algorithm ensures progress, since at least one out of a pair of conflicting requests will get committed. We propose a similar algorithm in [71], but leave a formal correctness proof to future work.

We observe though, that global serialization violations occur only if cohort servers receive their sub-requests for two conflicting requests in the opposite order. If they always receive their sub-requests in the same order then they simply conflict-check the response from earlier requests with those of latter requests as usual, which ensures

[33]The mapping could be stored in the switch's TCAM, in a high-performance implementation.

consistent conflict-checking across servers without the need for request bloating. In the example of section 9.2.2, if both servers process q_1 and q_2 in that order, then the server hosting b detects the conflicting operations on b and aborts q_2.

If all requests to a set of server hosts traverse a common *gateway switch* i_g, and two multi-server requests q_a and q_b are split at or below i_g, then any pair of these servers receives each pair of packets from q_a and q_b in the same order, when they do receive both packets. In the common hierarchical (star) network topology, for example, all packets to nodes below a switch in the hierarchy traverse that switch.

We can exploit this to derive a common agreed order for multi-server requests across servers in that subhierarchy, for example by defining the order of requests as the order in which their last packet is received. This works if requesters repeat a request's survey in the last packet of a request, ensuring this *trailer packet* is received at all cohorts. If a gateway switch is the coordinator for a request, it marks the survey packet as being gateway-ordered and does not bloat the request, forwarding its remaining packets normally. Hence, servers can process and conflict check a mix of gateway ordered and bloated requests. Gateway switches could also help to create an "official" total order on responses from their sub-hierarchies, by stamping response trailer packets with sequence numbers. Such orders can be used as the basis for delta-update processing, as sketched in Section 6.8.

As a parting thought, algorithms for concurrency control are subtle, and many erroneous ones have been published. We leave the modeling and correctness proving of these sketches to future work.

9.3 Optimizations for Cached Systems

While Atomic Transfer is a relatively general concept, its main application may lie in the construction of efficient atomic caching systems, such as those modeled in preceding chapters. The hallmark of such systems is that requesters can usually "predict" the response that results from a request and its effect on the state, since they cache the input and output data of each request they issue. In many cases, the only cause for mispredictions will be concurrent requests from other hosts. Also, for operations such as simple writes, the response to and effect of a request is always known. In such circumstances, the decision to commit a request and send its response can be decoupled in time and space from the act of updating persistent server state with its effect. This section outlines ideas for optimizations applicable to caching systems. Their common goal is to reduce the delay from when a request is issued until requesters are aware of its response, enhancing performance and reducing the scope for concurrency conflicts.

The optimizations depend on requesters embedding the expected reply to each request in the request itself. This could be achieved, for example, by encoding request packets in such a way that they can be easily converted into response packets by changing a packet type designation field. Note that embedding of responses is only possible if the request's execution is deterministic, or can be made deterministic by having the requester make and encode any non-deterministic choices along with the request.

9.3.1 Request Short-Circuiting

The main idea for reducing the response time for a request q bound for a server host b is to *short-circuit* q by multicasting its (expected) response r_q before q reaches b. As long as q has no conflicts, r_q would have been generated and sent by b later, anyway. Short-circuiting reduces the response-time for requests as well as obviating servers from encoding and sending response packets. More importantly, when combined with resilient transfer (Section 9.1) and gateway ordering (Section 9.2.4) it can be used to remove the need for two-phase commit (2PC) of multi-server requests, allowing the performance of multi-server requests [34] to approach that of single-server requests [198]. This might justify the considerable additional switch complexity introduced by short-circuiting.

A switch i can short-circuit a request bound for a set of servers $B \subseteq DATAS$ if it is a gateway switch for each server in B, that is: if all requests for servers in B pass through i. Note that in our model, the home-switch of a server b is always a gateway for b. More generally, switches in networks with a hierarchical (star) topology are gateways for the servers in their subnetwork. The core idea of short-circuiting is for

[34]Including requests redundantly sent to the replicas of a replicated logical host.

switch i to decide the total order of requests executed by the servers of B and send back their responses without waiting for the servers of B to persist their effects. In addition to forwarding a conflict-free request q towards it responders, i converts it into a response r_q and enqueues for subscribed channels, the same as if r_q had been received from the responder(s) of q. To the outside world, switch i is essentially indistinguishable from a single, high-performing server host.

While this is an attractive idea, there are several obstacles to its realization. First of all, short-circuiting more or less requires resilient transfer (Section 9.1), as switch i has effectively promised that request q will be executed and that it will generate response r_q. If one or more of the responders were to fail before completing q, the system would become inconsistent. Furthermore, short-circuting implies that requesters are trusted to issue only valid request with correctly predicted embedded responses[35]. This together with the implied hierarchical structure of the network means short-circuiting may be most applicable within the lower abstraction levels of a system, e.g. at the level of reads and writes. Still, that may also be the level where short-circuiting may be most helpful, as it is the level where many conflicts are detected and where requests and responses are comprised of a relatively large number of simple operations.

Multi-packet requests present another obstacle. First, the relative order of two multi-packet requests flowing concurrently through a switch is not entirely well-defined. We suggest using the reception of the final packet of a request as defining its order relative to other requests. Hence, once the final packet of a request is received and determined to be conflict-free, the request is added to the total order of requests and the corresponding response enqueued for transmission. While this is clearly biased against large requests, placing q into the order any sooner than this might lead to *all* other requests being delayed until the packets of q have been received and conflict-checked. Using the final packet also prevents "hung" partial requests from failed requesters from blocking progress of other requests. In the general case of a conflict being detected on two non-final packets from two concurrent requests q_1 and q_2, it is not clear which one should be dropped. While a switch could decide to arbitrarily

[35]Speculative Byzantine fault-tolerance protocols such as Zyzzyva [199] may point the way towards dealing with malicious requesters.

drop one of them, a more advanced implementation could note the conflict and let the request whose final packet is received first “win”, dropping the other request.

This highlights a second problem, which is that switch i cannot determine whether request q is conflict-free before it has received all its packets. Furthermore, it must buffer the packets up to that point, or else a conflict with one of the packets already purged from the buffer might go undetected. While this obviously increases the amount of buffer space required on switches, it might not increase it noticeably beyond what is needed for resilient transfer anyway, at least for requests of less than a few dozen packets. As mentioned in Section 9.2.1, the re-assembly of request messages in switch memory places an upper limit on request and response sizes. This might be circumvented by allowing switches to “shirk” responsibility for such requests and forward them tagged as “un-ordered”, signaling that they have not been short-circuited and may conflict with future requests. Lower-level switches or responders would re-assemble such requests and conflict-check and delay them until all responses concurrent with the shirked request are complete. These ideas await future development.

As a final remark, we note that many of the same problems and solutions for concurrency control and short-circuiting of (multi-packet) requests apply on responder hosts as well as switches. Some of the solutions might apply to multiprocessor (multi-core) CPUs, for example by using one core as an I/O processing “switch” that forwards request fragments to other cores for processing and handles conflict checking and coordination for the “sub-network” comprised of subordinate cores, e.g. But proper analysis of these issues awaits future work.

9.3.2 Request Pipelining

As we noted in Section 3.7.1, the round-trip-time between requesters and responders upper-bounds throughput in our model, as an operation on a name n cannot be successfully requested before the response to the prior conflicting operation on n has been received. This is unfortunate, especially in the case where a single requester issues requests to a set of names with complete absence of contention. As an example, the requester could be an interactive end-user application editing a document.

This restriction can be relatively easily lifted by defining a response r to be non-conflicting with a request q from the same requester, that is: if $\mathit{sender}(r) = \mathit{sender}(q)$. This allows a requester to *pipeline* a sequence of requests into the network without waiting for the responses to its earlier requests, issuing each new request based on the predicted state effects of the ones preceding it. This requires that requests from the same requester not be re-ordered in the network or alternatively that responders can re-establish the correct order.

The requester's speculation can fail if it issues a request q' assuming the effects of an earlier request q that is dropped, due to a concurrency conflict, for example. Hence, a speculative request q must have a *premise* field identifying the latest prior request whose effects were assumed as q was issued, using a sequence number, for example. Request q is then defined to conflict with the cancelation packet (see Section 9.2.1) for any request from the same requester with a lower or equal sequence number. The development and modeling of this scheme awaits future work.

9.3.3 Request Queueing for Liveness

While Atomic Transfer ensures execution correctness despite concurrency interference, it does not ensure liveness for requesters whose requests are dropped due to concurrency conflicts. In fact, as mentioned in Section 3.7.1, the network topology can induce perpetual unfairness for requesters many network hops away from a server.

One way this might be addressed is by letting the network help requesters coordinate access to contended data, by letting them "queue up" for access to the data after a conflict, rather than blindly re-issuing requests. When a request q is dropped due to a conflict with a response r, the cancelation notification would include the identity of r. Upon receiving the cancelation notification, the responder (or the switch currently buffering q, if short-circuiting is employed) would send the survey packet of q as a *queue packet* q_q, including the identity of r. Packet q_q would be forwarded normally to the subscribers of the relevant state, keeping them informed about conflicts affecting the state.

The requester host a of q knows that q has been dropped upon receiving r. However, instead of immediately re-issuing q, a would wait until receiving the queue packet q_q for q. Moreover, a would keep track of all other queue packets received between r and q_q that are tagged with r, indicating requests received ahead of q that were also dropped by r. It would delay the re-issuing of q until receiving the response corresponding to the last queue packet before q_q. In other words, a waits until the earlier conflicting requests have been re-issued and completed, before issuing its own. This way, contending requesters queue up for resubmission of their requests, giving each a fair chance to complete its request. Practically speaking, time-outs would be used to prevent unbounded waiting for failed requesters or dropped queue packets, etc.

The downside of this approach is that it places the burden of sending queue packets for dropped requests on responders, reducing the shielding from contention provided by the network. The requirement for responder involvement makes the approach less attractive. Gateway switches can shoulder that burden, when short-circuiting is employed. But even then, the forwarding of survey packets based on their prefixes may be problematic, as the set of overlapping subscribing hosts may be quite large, leading to excessive distribution of queue packets. On the flip side, the orderly queuing reduces the number of conflicting requests issued, lowering the load on switches and hosts alike. Proper analysis of these trade-offs awaits future work.

9.4 Security and Replication

This dissertation leaves security outside its scope. To be effective, though, security measures must be an integral part of a system's design. While proper consideration of security awaits future work, we briefly discuss the main security implications of systems based on Atomic Transfer.

The main observation is that Atomic Transfer needs atomic switches to see the names of requests and responses in cleartext, to be able to check them for conflicts. Furthermore, switches need to see the names to be able to forward packets to their destinations. By contrast, end-to-end concurrency control works with end-to-end encryption, allowing the data contents of a communication session to be completely hidden from eavesdroppers.

The parts of atomic packets not containing names may be encrypted end-to-end, e.g. the parts containing values. Also, the channels between switches may be encrypted, including software-defined channels such as TCP/IP connections and Virtual Private Networks (VPNs). We also note that Atomic Transfer does not adversely affect authentication, as atomic packets can include digital signatures [152, 200] and message authentication codes [201] the same as normal packets. The fact remains that trusting each switch along a network path results in lower security than trusting only the end-point hosts, as is the case with end-to-end encryption. Still, many distributed systems today run over VPNs overlaid on the Internet, sometimes placing significant trust in the gateway nodes and switches defining the VPNs. These security issue and trade-offs must be analyzed as a part of any thorough feasibility study of Atomic Transfer.

This dissertation also leaves replication outside its scope. However, some form of replication is necessary to achieve fault-tolerance in computing systems. We observe that application data caches come very close to being replicas of their data, the main difference being that caches are not required to persist their data and may unilaterally discard it at will. It would be interesting to extend our models to support flexible and dynamic replication, building on the cache synchronization protocols of Chapters 6 and 7 as well as Resilient Transfer and multi-server requests, as sketched in Sections 9.1 and 9.2 of this chapter. The main challenges, as in many replication schemes, are to detect replica host failures and/or atomically add and remove replicas from replica groups. We believe that network switches could help with these tasks, as they are in the unique situation of being able to divert network traffic from one destination to another. Furthermore, switches usually have more accurate information about the state of channels than do end-hosts. But such investigations await future work.

Adding complexity to the network is contentious [133], but we believe that switches are uniquely positioned to accelerate concurrency control, recoverability and atomic commit, in ways not open to end hosts. In summary, we believe that programmable switches can play an important role in the design of scalable infrastructures for efficient and flexible atomic applications.

References

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[2] J. Gray. Notes on Data Base Operating Systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[3] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 230–257, London, UK, 1999. Springer-Verlag.

[4] W3C. *SOAP Version Messaging Framework*, 1.2 edition, June 2003.

[5] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition, September 2001.

[6] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[7] C. T. Davies. Data processing spheres of control. *IBM Systems Journal*, 17(2):179–198, 1978.

[8] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[9] G. Vossen G. Weikum. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control.* Morgan Kaufmann, 2001.

[10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[11] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1997.

[12] A. Thomasian. Concurrency control: methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998.

[13] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *VLDB '1980: Proceedings of the sixth international conference on Very Large Data Bases*, pages 285–300. VLDB Endowment, 1980.

[14] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. *SIGMOD Rec.*, 24(2):23–34, 1995.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[16] M. J. Franklin, M. J. Carey, and M. Livny. Transactional client-server cache consistency: alternatives and performance. *ACM Trans. Database Syst.*, 22(3):315–363, 1997.

[17] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, 1974.

[18] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, 1989.

[19] M. P. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. In *PODS '88: Proceedings of the seventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 201–210, New York, NY, USA, 1988. ACM.

[20] International Business Machines Corporation. Information Managament System Virtual Storage (IMS/VS), General Information Manual, GH20-1260, 1991.

[21] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. System R: relational approach to database management. *ACM Trans. Database Syst.*, 1(2):97–137, 1976.

[22] Robert D. Sloan. A practical implementation of the data base machine - Teradata DBC/1012, 1992.

[23] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):4–24, 1990.

[24] D. J. DeWitt, R. H. Gerber, G. Graefe, M. L. Heytens, K. B. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. In *VLDB '86: Proceedings of the 12th International Conference on Very Large*

Data Bases, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[25] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[26] W. Kim. Object-oriented databases: definition and research directions. *Knowledge and Data Engineering, IEEE Transactions on*, 2(3):327–341, Sep 1990.

[27] M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The object-oriented database system manifesto. pages 1–20, 1992.

[28] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[29] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402, 1995.

[30] M. J. Carey, D. J. DeWitt, D. Frank, M. Muralikrishna, G. Graefe, J. E. Richardson, and E. J. Shekita. The architecture of the EXODUS extensible DBMS. In *OODS '86: Proceedings on the 1986 international workshop on Object-oriented database systems*, pages 52–65, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[31] M. A. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *ATEC '99: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[32] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented DBMS. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 553–564. VLDB Endowment, 2005.

[33] R. Sears and E. Brewer. Stasis: flexible transactional storage. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 29–44, Berkeley, CA, USA, 2006. USENIX Association.

[34] M. N. Nelson, B. B. Welch, and J. K. Ousterhout. Caching in the sprite network file system. *ACM Trans. Comput. Syst.*, 6(1):134–154, 1988.

[35] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 239–253, New York, NY, USA, 1991. ACM.

[36] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. *ACM Trans. Comput. Syst.*, 12(1):33–57, 1994.

[37] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 1–10, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[38] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[39] N. W. Paton and O. Diaz. Active database systems. *ACM Comput. Surv.*, 31(1):63–103, 1999.

[40] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing.* Morgan Kaufmann, 1995.

[41] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[42] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. *Computer Architecture, 1993., Proceedings of the 20th Annual International Symposium on*, pages 289–300, May 1993.

[43] N. Shavit and D. Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[44] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. P., H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

[45] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[46] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–13, New York, NY, USA, 2006. ACM.

[47] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[48] H. Garcia-Molina and K. Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM.

[49] P. K. Chrysanthis and K. Ramamritham. ACTA: a framework for specifying and reasoning about transaction structure and behavior. *SIGMOD Rec.*, 19(2):194–203, 1990.

[50] S. G. Jajodia and L. Kerschberg, editors. *Advanced Transaction Models and Architectures.* Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[51] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[52] M. Satyanarayanan. The evolution of Coda. *ACM Trans. Comput. Syst.*, 20(2):85–124, 2002.

[53] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 172–182, New York, NY, USA, 1995. ACM.

[54] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *ACM Operating Systems Review*, 36(SI):1–14, 2002.

[55] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, and B. Zhao. OceanStore: an architecture for global-scale persistent storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM.

[56] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[57] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Trans. Comput. Syst.*, 26(2):1–26, 2008.

[58] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 205–220, New York, NY, USA, 2007. ACM.

[59] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[60] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Not.*, 12(3):128–137, 1977.

[61] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, New York, NY, USA, 1975. ACM.

[62] E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. Technical report, Cambridge, MA, USA, 1981.

[63] N. A. Lynch, M. Merrit, W. E. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, 1993.

[64] B. Liskov and R. Scheifler. Guardians and actions: linguistic support for robust, distributed programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 7–19, New York, NY, 1982.

[65] B. Liskov, D. Curtis, P. Johnson, and R. Scheifer. Implementation of Argus. In *SOSP '87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 111–122, 1987.

[66] J. L. Eppinger, L. B. Mummert, and A. Z. Spector, editors. *Camelot and Avalon: a distributed transaction facility.* Morgan Kaufmann, 1991.

[67] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, 1996.

[68] E. A. Lee. The Problem With Threads. *IEEE Computer Magazine*, 39(5):33–42, May 2006.

[69] Fred B. Schneider. The State Machine Approach: A Tutorial. Technical report, Ithaca, NY, USA, 1986.

[70] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[71] H. Thorvaldsson and K. J. Goldman. Dynamic Evolution in a Survivable Application Infrastructure. In *IASTED Parallel and Distributed Computing and Systems (PDCS)*, 2006.

[72] Akamai Technoogies. `http://www.akamai.com`.

[73] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[74] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 19, Berkeley, CA, USA, 1994. USENIX Association.

[75] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical internet object cache. In *ATEC '96: Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, pages 13–13, Berkeley, CA, USA, 1996. USENIX Association.

[76] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: towards a new global caching architecture. *Comput. Netw. ISDN Syst.*, 30(22-23):2169–2177, 1998.

[77] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[78] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: distributed shared memory based on type-specific memory coherence. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–176, New York, NY, USA, 1990.

[79] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. Technical report, Pittsburgh, PA, USA, 1993.

[80] A.L. Cox, S. Dwarkadas, and P. Keleher. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Winter 94 USENIX Conference*, pages 115–131, Berkeley, CA, 1994.

[81] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992.

[82] D. J. Scales and M. S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *In First Symposium on Operating Systems Design and Implementation*, pages 101–114, 1994.

[83] L. Aguilar. Datagram routing for internet multicasting. In *SIGCOMM '84: Proceedings of the ACM SIGCOMM symposium on Communications architectures and protocols*, pages 58–63, New York, NY, USA, 1984. ACM.

[84] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[85] P. Th. Eugster and R. Guerraoui. Probabilistic multicast. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 313–324, 2002.

[86] D. Kostić, A. C. Snoeren, A. Vahdat, R. Braud, C. Killian, James W. Anderson, J. Albrecht, A. Rodriguez, and E. Vandekieft. High-bandwidth data dissemination for large-scale distributed systems. *ACM Trans. Comput. Syst.*, 26(1):1–61, 2008.

[87] D. Gelernter. Generative communication in linda. *ACM Trans. Program. Lang. Syst.*, 7(1):80–112, 1985.

[88] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[89] P. Mockapetris and K. J. Dunlap. Development of the domain name system. *SIGCOMM Comput. Commun. Rev.*, 18(4):123–133, 1988.

[90] M. Gritter and D. R. Cheriton. An architecture for content routing support in the internet. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 4–4, Berkeley, CA, USA, 2001. USENIX Association.

[91] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 186–201, New York, NY, USA, 1999. ACM.

[92] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *IEEE/ACM Trans. Netw.*, 12(2):205–218, 2004.

[93] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: routing on flat labels. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 363–374, New York, NY, USA, 2006. ACM.

[94] X. Yang, D. Clark, and A. W. Berger. NIRA: a new inter-domain routing architecture. *IEEE/ACM Trans. Netw.*, 15(4):775–788, 2007.

[95] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 654–663, New York, NY, USA, 1997. ACM.

[96] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *ACM SIGCOMM*, pages 161–172, 2001.

[97] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[98] A. Rowstron P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. 2001.

[99] B. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

[100] Van Jocobson's Google Talk. `http://video.google.com/videoplay?docid=-6972678839686672840`.

[101] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[102] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen. The Information Bus: an architecture for extensible distributed systems. In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 58–68, 1993.

[103] TIBCO. TIB/Rendezvous, whitepaper, 1999.

[104] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 262, 1999.

[105] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 219–227, 2000.

[106] G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Trans. Softw. Eng.*, 27(9):827–850, 2001.

[107] K. P. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Technical report, Ithaca, NY, USA, 1986.

[108] K. P. Birman. The process group approach to reliable distributed computing. *Commun. ACM*, 36(12):37–53, 1993.

[109] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[110] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 27(5):44–57, 1993.

[111] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 28(1):11–21, 1994.

[112] Intel Corporation. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3), 2002.

[113] Cisco Systems, Inc. `http://www.cisco.com`.

[114] J. S. Turner, P. Crowley, J. DeHart, A. Freestone, B. Heller, F. Kuhns, S. Kumar, J. Lockwood, J. Lu, M. Wilson, C. Wiseman, and D. Zar. Supercharging PlanetLab: a high performance, multi-application, overlay network platform. *SIGCOMM Comput. Commun. Rev.*, 37(4):85–96, 2007.

[115] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[116] Mark Handley, Orion Hodson, and Eddie Kohler. XORP: an open platform for network research. *SIGCOMM Comput. Commun. Rev.*, 33(1):53–57, 2003.

[117] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building PlanetLab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[118] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. *Computer*, 38(4):34–41, 2005.

[119] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. volume 26, pages 5–17, New York, NY, USA, 1996. ACM.

[120] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[121] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[122] H. D. Thorvaldsson and K. Goldman. Architecture and Execution Model for a Survivable Work Flow Architecture Infrastructure. Technical Report WUCSE-2005-61, Washington University, 2005.

[123] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[124] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *19th ACM SOSP*, pages 164–177, 2003.

[125] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[126] H. Garcia-Molina and K. Salem. Main Memory Database Systems: An Overview. volume 4, pages 509–516, Piscataway, NJ, USA, 1992. IEEE Educational Activities Department.

[127] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[128] D. Pnevmatikatos, E. P. Markatos, G. Magklis, and S. Ioannidis. On using network RAM as a non-volatile buffer. *Cluster Computing*, 2(4):295–303, 1999.

[129] L. Kleinrock. The latency/bandwidth tradeoff in gigabit networks. *Communications Magazine, IEEE*, 30(4):36–40, Apr 1992.

[130] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[131] P. F. Reynolds, C. Williams, and R. R. Wagner. Isotach Networks. *IEEE Transactions on Parallel and Distributed Systems*, 8:337–348, 1997.

[132] R. G. Bartholet. A Performance Study of Isotach Version 1.0. Technical report, Charlottesville, VA, USA, 1999.

[133] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[134] A. Thomasian. Distributed Optimistic Concurrency Control Methods for High-Performance Transaction Processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(1):173–189, 1998.

[135] W. Forrest, J. M. Kaplan, and N. Kindler. Data centers: How to cut carbon emissions and costs. *The McKinsey Quarterly*, November 2008.

[136] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, 1990.

[137] P. E. O'Neil. The Escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, 1986.

[138] T. Ballardie, P. Francis, and J. Crowcroft. Core based trees (CBT). *SIGCOMM Comput. Commun. Rev.*, 23(4):85–95, 1993.

[139] H. W. Holbrook and D. R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. *SIGCOMM Comput. Commun. Rev.*, 29(4):65–78, 1999.

[140] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. Protocol independent multicast-sparse mode (pim-sm): Protocol specification, 1998.

[141] H. T. Kung, T. Blackwell, and A.n Chapman. Credit-based flow control for atm networks: credit update protocol, adaptive credit allocation and statistical multiplexing. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 101–114, New York, NY, USA, 1994. ACM.

[142] B. Cain H. Holbrook. RFC 4607: Source-Specific Multicast for IP, 2006.

[143] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[144] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[145] W. Timothy Strayer, Bert J. Dempsey, and Alfred C. Weaver. *XTP: the Xpress Transfer Protocol.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[146] S. Paul, K. K. Sabnani, J. C. Lin, and S. Bhattacharyya. Rmtp: A reliable multicast transport protocol. In *IEEE Journal on Selected Areas in Communications*, pages 1414–1424, 1996.

[147] R. Yavatkar, J. Griffoen, and M. Sudan. A reliable dissemination protocol for interactive collaborative applications. In *MULTIMEDIA '95: Proceedings of the third ACM international conference on Multimedia*, pages 333–344, New York, NY, USA, 1995. ACM.

[148] H. W. Holbrook, S. K. Singhal, and D. R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. *SIGCOMM Comput. Commun. Rev.*, 25(4):328–341, 1995.

[149] C. Papadopoulos, G. Parulkar, and G. Varghese. An error control scheme for large-scale multicast applications. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, page 310, New York, NY, USA, 1998. ACM.

[150] J. Jing, A. Sumi Helal, and A. Elmagarmid. Client-server computing in mobile environments. *ACM Comput. Surv.*, 31(2):117–157, 1999.

[151] P. Jones D. Eastlake. RFC 3174: US Secure Hash Algorithm 1 (SHA-1), 2001.

[152] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[153] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. *SIGOPS Oper. Syst. Rev.*, 34(2):19–20, 2000.

[154] J. S. Turner. A proposed architecture for the GENI backbone platform. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 1–10, New York, NY, USA, 2006. ACM.

[155] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[156] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[157] C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *J. ACM*, 36(2):230–269, 1989.

[158] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.

[159] P. A. Franaszek, J. T. Robinson, and A. Thomasian. Concurrency control for high contention environments. *ACM Trans. Database Syst.*, 17(2):304–345, 1992.

[160] P. Franaszek and J. T. Robinson. Limitations of concurrency in transaction processing. *ACM Trans. Database Syst.*, 10(1):1–28, 1985.

[161] M. J. Carey, M. J. Franklin, and M. Zaharioudakis. Fine-grained sharing in a page server OODBMS. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 359–370, New York, NY, USA, 1994. ACM.

[162] R. E. Gruber. Optimism vs. Locking: A Study of Concurrency Control for Client-Server Object-Oriented Databases. Technical report, Cambridge, MA, USA, 1997.

[163] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design Considerations for Distributed Caching on the Internet. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, page 273, Washington, DC, USA, 1999. IEEE Computer Society.

[164] G. Barish and K. Obraczka. World Wide Web caching: Trends and techniques. *IEEE Communications Magazine*, 38:178–184, 2000.

[165] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Trans. Netw.*, 8(5):568–582, 2000.

[166] X. Tang and S. T. Chanson. Coordinated En-Route Web Caching. *IEEE Trans. Comput.*, 51(6):595–607, 2002.

[167] C. Papadopoulos, G. Parulkar, and G. Varghese. Light-weight multicast services (lms): a router-assisted scheme for reliable multicast. *IEEE/ACM Trans. Netw.*, 12(3):456–468, 2004.

[168] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software IP lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, 2004.

[169] Patrick Crowley. *Network Processor Design: Issues and Practices*. Academic Press, Inc., Orlando, FL, USA, 2002.

[170] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a Role-Based Trust Management Framework. In *IEEE Symposium on Security and Privacy*, 2002.

[171] Trevor Jim. SD3: A Trust Management System with Certified Evaluation. In *2001 IEEE Symposium on Security and Privacy*, page 106, 2001.

[172] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe. The case for separating routing from routers. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 5–12, New York, NY, USA, 2004. ACM.

[173] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Ninth Workshop on Hot Topic in Operating Systems (HotOS-IX)*, May 2003.

[174] W. Kohler, A. Shah, and A. Raab. Overview of TPC Benchmark C: The Order-Entry Benchmark. Technical report, Transaction Processing Performance Council, December 1991.

[175] R. la Briandais. File searching using variable length keys. In *Western Joint Computer Conferences*, volume 15, pages 295–298, San Fransisco, CA, USA, May 1959.

[176] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[177] V. Srinivasan and G. Varghese. Faster IP lookups using controlled prefix expansion. *SIGMETRICS Perform. Eval. Rev.*, 26(1):1–10, 1998.

[178] D.E. Taylor, J.W. Lockwood, T.S. Sproull, J.S. Turner, and D.B. Parlour. Scalable IP lookup for programmable routers. *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, 2:562–571 vol.2, 2002.

[179] D. Knuth. *The Art of Computer Programming, volume 3: Sorting and Searching, 2nd Ed.* Addison-Wesley., Boston, MA, USA, 1998.

[180] H. S. Warren. *Hacker's Delight.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[181] S. Kumar, J. Turner, P. Crowley, and M. Mitzenmacher. HEXA: Compact Data Structures for Faster Packet Processing. *Network Protocols, 2007. ICNP 2007. IEEE International Conference on*, pages 246–255, Oct. 2007.

[182] M. A. Bender, E. D. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 399, Washington, DC, USA, 2000. IEEE Computer Society.

[183] J. S. Turner. Jonathan S. Turner, personal communication.

[184] L. I. Trabb Pardo. *Set representation and set intersection.* PhD thesis, Stanford, CA, USA, 1978.

[185] S. Carlsson, C. Levcopoulos, and O. Petersson. Sublinear Merging and Natural Merge Sort. In *SIGAL '90: Proceedings of the International Symposium on Algorithms*, pages 251–260, London, UK, 1990. Springer-Verlag.

[186] M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, 1998.

[187] D. E. Taylor. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv.*, 37(3):238–275, 2005.

[188] D. R. Morrison. PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, 1968.

[189] R. Sedgewick. *Algorithms in C++*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[190] P. Bagwell. Fast And Space Efficient Trie Searches. Technical report, 2000.

[191] R. S. Bird. Two dimensional pattern matching. *Info. Process. Lett.*, 6(5):168–170, 1977.

[192] G. Jacobson. Space-efficient static trees and graphs. *Symposium on Foundations of Computer Science*, 0:549–554, 1989.

[193] H. Song, J. Turner, and J. Lockwood. Shape Shifting Tries for Faster IP Route Lookup. In *ICNP '05: Proceedings of the 13TH IEEE International Conference on Network Protocols*, pages 358–367, Washington, DC, USA, 2005. IEEE Computer Society.

[194] E. N. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[195] L. Alvisi. *Understanding the message logging paradigm for masking process crashes.* PhD thesis, Ithaca, NY, USA, 1996.

[196] E. N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback and Fast Output Commit. *IEEE Transactions on Computers*, 41:526–531, 1992.

[197] C. Dwork and D. Skeen. The inherent cost of nonblocking commitment. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 1–11, New York, NY, USA, 1983. ACM.

[198] M. Abdallah, R. Guerraoui, and P. Pucheral. Dictatorial Transaction Processing: Atomic Commitment Without Veto Right. *Distrib. Parallel Databases*, 11(3):239–268, 2002.

[199] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[200] R. L. Rivest, A. Shamir, and L. M. Adelman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Technical Report MIT/LCS/TM-82, 1977.

[201] B. Prenel and P. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Proc. 15th Conf. on Advances in Cryptology*, pages 1–14, 1995.

Vita

Haraldur Darri Thorvaldsson

Date of Birth	October 27, 1973
Place of Birth	Reykjavík, Iceland
Degrees	B.S. computer Science, June 1998 Ph.D. Computer Science, May 2009
Professional Societies	Association for Computing Machines IEEE Computer Society

Publications

Pallemulle, Sajeeva L., Thorvaldsson, Haraldur D., Goldman, Kenneth J., *Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures*, In proceedings of the 28th IEEE International Conference on Distributed Computing Systems, pages 260–268, June 2008.

Pallemulle, Sajeeva L., Wehrman, Ian, Thorvaldsson, Haraldur D., Goldman, Kenneth J., *Perpetual: Byzantine Fault Tolerance for Federated Distributed Applications*, Washington University, Department of Computer Science and Engineering, Technical Report WUCSE-2007-50, December 2007

Thorvaldsson, Haraldur D., Goldman, Kenneth J., *Dynamic Evolution in a Survivable Application Infrastructure*, In proceedings of the 18th International Conference on Parallel and Distributed Computing and Systems, November 2006.

Thorvaldsson, Haraldur D., Goldman, Kenneth J., *Architecture and Execution Model for a Survivable Workflow Transaction Infrastructure*, Washington University, Department of Computer Science and Engineering, Technical Report WUCSE-2005-61, December 2005

May 2009

Atomic Transfer, Thorvaldsson, Ph.D. 2009