Measuring Availability in Optimistic Partition-tolerant Systems with Data Constraints

Mikael Asplund, Simin Nadjm-Tehrani
Department of Computer and Information Science,
Linköping University
SE-581 83 Linköping, Sweden
{mikas,simin}@ida.liu.se

Stefan Beyer, Pablo Galdamez
Instituto Tecnolgico Informtica
Universidad Politcnica de Valencia
Camino de Vera, s/n, 46022 Valencia, Spain
{stefan, pgaldamez}@iti.upv.es

Abstract

Replicated systems that run over partitionable environments, can exhibit increased availability if isolated partitions are allowed to optimistically continue their execution independently. This availability gain is traded against consistency, since several replicas of the same objects could be updated separately. Once partitioning terminates, divergences in the replicated state needs to be reconciled. One way to reconcile the state consists of letting the application manually solve inconsistencies. However, there are several situations where automatic reconciliation of the replicated state is meaningful. We have implemented replication and automatic reconciliation protocols that can be used as building blocks in a partition-tolerant middleware. The novelty of the protocols is the continuous service of the application even during the reconciliation process. A prototype system is experimentally evaluated to illustrate the increased availability despite network partitions.

1 Introduction

Prevalence of distributed services and networked solutions has made many enterprises critically dependent on service availability. Whereas earlier centralised solutions were made resilient to service faults by deploying redundancy, the new generation of distributed services need to show resilience to overloads and network partitions. There are many applications that require automatically managed, distributed, secure, mutable object stores. A commercial instance of this problem appears in Software distribution. According to Sun Microsystems [4], network partitions are indeed interesting to study since global corporate intranetworks are typically not richly connected. Hence, service availability of distributed data storage systems is potentially affected by denial of service attacks (DoS) that render parts of the network as inaccessible [6].

This paper addresses support for maintaining distributed objects with integrity constraints in presence of network partitions. Providing fault tolerance in such distributed object systems requires relatively complex mechanisms to properly handle all the different fault scenarios. One solution is to relieve the application writers, and get them to rely on a middleware that provides fault tolerance services. This is a direction pursued in the European DeDiSys research project [9]. In particular, the algorithms for replication and reconciliation implemented in this paper will be deployed in an extension of CORBA middleware. However, they can be considered as general building blocks to be integrated in any middleware.

The basic problem of network partitions is that there is no way of knowing what is happening in the other parts of the system. A bank customer cannot make a payment if not enough money exists on his/her bank account, and you cannot book a flight that is already full. These kinds of integrity constraints exist in most applications either explicitly or implicitly in the operation semantics; but what happens if the bank account is used for two payments at the same time in two disconnected parts? One typically resorts to a "safe" solution that implies periods of unavailability.

Another way to deliver service in a partitioned system with integrity constraints is to act optimistically. This means to provisionally accept some operations, but allow them to be revoked or undone at a later stage, if necessary. To revoke previously accepted operations might be unacceptable in some cases, but there are also situations where it is better than general unavailability. Another possibility is to perform some kind of compensating action specific for that operation. Many applications have a mix of operations where some non-critical operations can be treated optimistically, while the critical operations must wait. We claim an application can improve its overall availability by provisionally accepting operations that may later be revoked. We pro-

pose protocols that allow for automatic reconciliation of the state of the network, by replaying the (logged) operations serviced during the partition and discarding some (revocable) operations that violate the integrity constraints in the reconstructed state. The novelty of our implemented algorithm is that it builds the new repaired network state and *at the same time* serves the new incoming operations. That is, operations arriving after network reunification but prior to installation of the new state are not denied service.

The contributions of this paper are twofold. First, we present the implementation of a previously unimplemented reconciliation protocol[1] embedded as a middleware service. The aim of the protocol is to give continuous service during network partitions. Second, we show the improved performance due to giving service during the degraded mode, and also during reconciliation phase after a network partition. The experiments thus constitute validation tests for performance of the Java implementation of the protocol. This provides a proof of concept for the algorithms prior to integration in CORBA.

To provide a repeatable experimental setting for measuring performance, we have implemented a synthetic application that simulates changes of numerical values for object states. This can, for example, be seen as an abstraction of distributed sensor-actor systems, and fusion of data based on reported measures. This application is introduced in Section 2 and is used for explaining the reconciliation process. The rest of the paper is organised as follows. Section 3 describes a partition-aware replication algorithm called P4, and Section 4 describes the continuous service reconciliation protocol. In Section 5 we propose a set of metrics that are suitable for evaluating availability in systems with optimistic replication. Using these metrics we evaluate the proposed algorithms in Section 6. Section 7 presents related work, and finally we conclude and give directions for future work.

2 Test application

A synthetic application has been developed to serve as a test bed for trade-off studies. We describe it here to reuse for illustration of the workings of the reconciliation process.

The application is composed of a set of real number objects. Possible operations are addition, multiplication and division. An operation is applied to the current value with a random constant. The constant is an integer uniformly distributed in the intervals $[-10, -1]$, and $[1, 10]$. This creates a total of 60 distinct operations. There are also integrity constraints in the system expressed as: $n_1 + c < n_2$ where n_1 and n_2 are object values and c is a constant. Although the application is very simple, it is complex enough to give an indication of how the algorithms perform. Moreover, the application allows key system parameters to be changed for experimentation purposes.

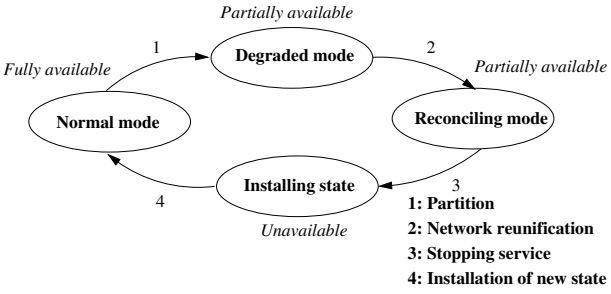

Figure 1. System modes

3 Replication

The system modes of operation can be described as the four phases depicted in Figure 1. We proceed by describing the need for a replication protocol that allows consistency to be temporarily violated but later restored.

In the passive replication model each object has a primary copy, and the distributed replicas are updated using a replication protocol. Traditional pessimistic replication techniques that attempt to provide single-copy consistency [5] are not suitable for optimistic partitionable systems in which more than one partition continues to accept updates during partitioning. Replication protocols for such systems need to temporarily accept possible inconsistencies. Hence these protocols allow the state in different partitions to diverge. If strict consistency is to be restored when the system recovers, a reconciliation protocol is required. Replication and reconciliation protocols need to match each other, as only inconsistencies that can be removed at reconciliation time can be allowed.

An optimistic protocol might allow the degree to which inconsistencies are allowed to be configured. We have designed an optimistic replication protocol, called Primary Per Partition Protocol (P4), which uses a new approach to trade consistency for system availability. The protocol bases consistency on integrity constraints. Integrity constraints can be pre-conditions, which have to met before an operation is executed, post-conditions, which have to be met after an operation is executed or invariants, which are not associated to an operation but to a set of objects and have to be met at all times. The remainder of this section provides a short summary of the protocol that is described in detail earlier [6].

The protocol assumes the presence of a group membership service that provides all the server nodes with a single view of which nodes are part of the system or the current partition. Furthermore, a group communication service provides the nodes with reliable FIFO broadcast according to

the definition by Hadzilacos and Toueg [12].

The protocol employs a relaxed passive replication model. Read-only operations are allowed on any replica, but write operations have to be directed to the primary copy of the object being accessed. If the primary copy of the object is in a different partition, a secondary copy is promoted to a temporary primary. In order to increase system availability, we allow write operations on temporary primaries in certain conditions. During partitioning, secondary copies of an object might be stale, if the primary copy resides in a different partition. During reconciliation, constraints might be violated retrospectively, when missed updates are propagated. Therefore, some operations that were performed during partitioning might have to be undone to restore consistency. This behaviour might be acceptable for the majority of the operations, but there are some operations that should never occur, if they might have to be undone later on.

These "critical operations" include operations on data that require strict consistency at all times and operations that simply cannot be undone, such as operations with irreversible side-effects. We therefore allow the labelling of integrity constraints as *critical constraints*.

A constraint labelled critical is a constraint that needs up-to-date versions of all of the participating objects. Such a constraint cannot be evaluated if a participating object is stale. Furthermore, the protocol has to take certain precautions to ensure that critical constraints are never violated "in retrospect" during reconciliation. In contrast *non-critical* constraints can be evaluated on stale objects. A non-critical invariant constraint has to be re-evaluated during the reconciliation; that is, the reconciliation protocol has to perform constraint re-evaluation.

A write operation in our replication protocol in the absence of failures can be summarised in the following steps:

1. All object write invocations have to be directed to the primary replica.
2. All the pre-condition constraints, associated with the operation are evaluated. If a constraint is not met, the invocation is aborted.
3. The operation is invoked. Nested invocations cause sub-invocations to be started.
4. Once the primary replica has updated its local state, all the post-condition and invariant constraints, associated with the operation are evaluated. If a constraint is not met, the invocation is aborted.
5. All primary replicas updated in the invocations propagate the new object states to the secondary replicas.
6. Once this update transfer has terminated, the operation result is returned to the client.

A failure might occur in the form of a node failure or a link failure. Since we cannot distinguish between a failed node and an isolated node, all failures are treated as network partitions until recovery time. A write operation in degraded mode is similar to that in normal mode with the following additions:

1. If the primary copy of an object being written to is not found, a secondary copy is chosen in some predetermined way, for example based on the replica identifier. The chosen secondary replica is promoted to a "temporary primary". This is not done, if the operation has a critical constraint as a pre- or post-condition.
2. Objects that are changed are marked as "revocable", if any of the invariant constraints associated to the operation that has been executed has been evaluated on possibly stale objects.
3. Critical constraints are not evaluated, if a participating object might be stale. If this were the case, the invocation is aborted.
4. Non-critical invariant constraints with possibly stale objects are marked for re-evaluation at reconciliation time.
5. Operations with critical constraints that include a revocable object are not permitted, so that critical constraints cannot be violated retrospectively.

Note that the above description applies both to operation in degraded mode and continuous service of incoming operations during reconciliation. In order to manage the reconciliation process (see below) the replication protocol needs to log those operations that have been serviced while in partition. These operations need to be reconsidered during the reconciliation phase.

4 Reconciliation

This section describes the implementation of a protocol that aims to continuously service client (write) requests even during the reconciliation process. A formal description of the protocol with a proof of correctness was presented earlier [1]. Here we show the architectural units that have been realised in Java and their interactions in terms of pseudo code.

Figure 2 shows the basic architectural components of our replication and reconciliation protocols. Each node contains the middleware and a number of application objects. The middleware is composed of a number of services, of which the replication support is the focus of this paper. This component is in turn composed of a replication protocol, i.e., P4, and a reconciliation protocol, i.e., the continuous service (CS) protocol. These protocols rely on additional middleware services such as Group Communication (GC) and Constraint Consistency Manager (CCM). The CCM is used to check consistency of integrity constraints. The box with

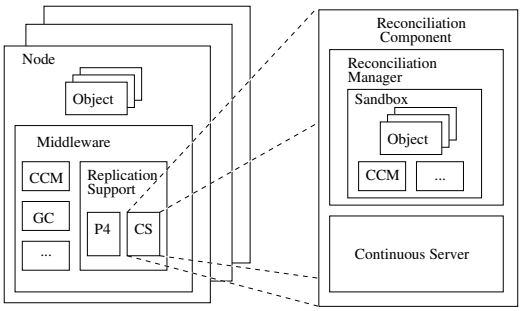

Figure 2. Architecture

"..." is an abstraction of other services in the middleware not relevant for this evaluation. Our prototype implementation that is used for evaluating the reconciliation protocol is based on this architecture.

We proceed by explaining the actions of the CS reconciliation protocol. This protocol is faced with the task of merging a number of operations that have been performed in different partitions. It must also preserve constraint consistency. Furthermore, as operations are replayed the client perceived order on operations (for operations invoked by the *same* client) is respected. In parallel with this process the protocol takes care of operations that arrive at the reunified but not fully reconciled partition.

Algorithm 1 Continuous Server

On reunify:
Send all logs to Reconciliation Manager(s)

On operation invocation:
If not stopped, apply operation
Check consistency, abort if not consistent
Send log to Reconciliation Manager
Suspend reply until later

On receive getState:
Send last stored object state (from normal mode)

On receive logAck:
Send suspended replies to client

On receive stop:
Stop accepting new operations
Send stopAck

On receive install:
Change state of local objects to received state

The reconciliation protocol is composed of two types of processes: continuous servers and reconciliation managers. Algorithms 1 and 2 show the pseudo code for the protocol running at each node. Every node will run a continuous server during the reconciliation, whereas only one elected node will run the reconciliation manager. During reconciliation, the reconciliation manager will replay previously applied operations. This replay process is performed in a sandbox environment, which contains the application objects and the basic middleware components that are required for running the application on a single node.

Algorithm 2 Reconciliation Manager

On reunify:
Elect which node acts as reconciliation manager
Determine which objects to reconcile
Send *getState* request to servers

On receive log:
Add log to $opset$
Send *logAck* to server

On receive state:
Create object in sandbox environment

If opset not empty and all states received:
Replay first operation in *opset* in sandbox environment
Check consistency, abort if not consistent

If opset empty and all states received:
Send stop message to all servers

On receive stopAck:
Wait for opset to become empty
Send out new state to all servers

The responsibility of the continuous server is to accept invocations from clients and sending logs to the elected reconciliation manager during reconciliation. At the beginning of each reconciliation phase the nodes in the repaired network elect a reconciliation manager among themselves. The reconciliation manager is responsible for merging server logs that are sent during reconciling mode. Eventually, upon reaching a stable state, the reconciliation manager sends an install message with the new state to all servers (see transition 4 in Figure 1).

During reconciliation mode, the state that is being constructed by the reconciliation manager may not yet reflect all the operations that have arrived during degraded mode. Therefore, the only state in which the incoming operations can be applied to is one of the partition states from the degraded mode. In other words, we keep virtual partitions for servicing incoming operations while the reconciliation phase lasts.

Each continuous server will immediately send a log message to the reconciliation manager if it receives an invocation during the reconciliation phase. The server will then wait until it has received an acknowledge from the manager before sending a reply to the client. When the manager has finished replaying all operations, it sends a message to all nodes to stop accepting new invocations. The manager will continue accepting log messages from servers, even after a

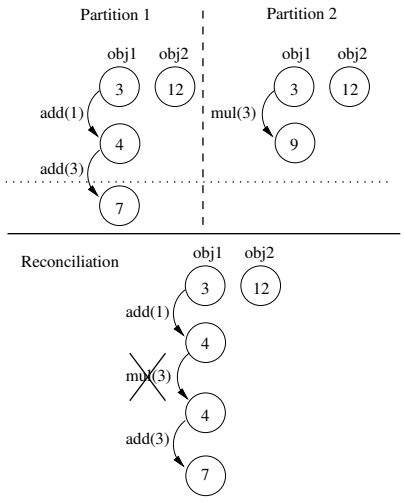

Figure 3. Reconciliation example

stop has been sent. Note also that the continuous servers keep sending the logs until they have sent their acknowledgements of the stop. However, once a stopAck message has been received from a given server, then no further log messages will arrive from it. This is to ensure that no operations are performed during the installation of a new reconciled state. This (short) period is the only interval in which the system is completely unavailable.

4.1 Example

To illustrate the potential effect of reconciliation as a result of replaying operations we show a trivial synthetic scenario with one integrity constraint in Figure 3. We use the application from Section 2, using two objects with an initial state 3 and 12 respectively. There is a constraint stating that $obj_1 + 1 < obj2$. During the degraded mode one operation is performed in each partition on the first object. At the start of the reconciliation the state is (4,12) and (9,12) respectively. Just after the reconciliation starts, yet another operation is invoked in the first partition. The reconciliation revokes operation 2 since it violates the constraint but finally accepts operations 1 and 3. So the final state that is installed is (7,12). For a deeper discussion on possible operation orderings the reader is referred to [2].

5 Performance metrics

Availability is formally expressed as the probability of a system being operational at any given point of time [20]. This is to be distinguished from reliability that measures the probability of not observing any failures before a given time point. Both reliability and availability have been extensively studied in the context of computer systems with an emphasis on hardware failures to justify a claim on a system's dependability.

In the context of this paper we are faced with a service that is to be available on a distributed (networked) platform. Measuring availability of the service is possible by performing a number of experiments on the system running over some time interval. To compute the probability of the service being available, one can measure the periods that the service is *operational* during the experiments, compute an average operational period, and then compute the probability measure by dividing the average operational period over the chosen interval. This measure is of course highly affected by the potential number of failures during the experimental period. These failures can be induced (injected) during experimentation, but their likelihood has to be supported by some empirical evidence obtained from the application domains, using the hardware and software characteristics of the real application. However, this is not the whole story. The core problem of defining metrics for consistency-dependent distributed object systems is that in presence of some degraded service, one has to identify what is exactly meant by being "operational" (which services, or which operations under which conditions).

5.1 Partially operational

Figure 4 shows the set of arriving operations during a given time interval. For a system to be considered fully operational the operations that are invoked by clients have to be performed together with checking integrity constraints. If the integrity constraints can be checked, i.e., the system is not partitioned, then the service is considered operational even though the constraint may not hold (and thus the operation not performed). If the integrity constraint cannot be checked, then we are faced with one of two situations. Either the integrity constraint is critical, in which case the operation cannot be allowed (the system is non-operational), or the constraint is non-critical. An operation with an associated non-critical constraint, which is invoked in a degraded mode, can be considered to render a system operational in the degraded mode. However, this is not the whole story either. We need to consider what happens to this operation once the degraded mode has ended. In some cases, the operation will be considered as valid after returning to the normal (fully consistent) mode of the system, and in some cases this operation has to be revoked (undone) or perhaps compensated, since the process of recovering from the earlier failure has rendered this operation as unacceptable. The new metrics thus have to consider appropriate measures that reflect these elements of partial availability and apparent availability. Another aspect in devising the set of experiments is a clear parameterisation in terms of the load; not only in terms of the volume of operations that

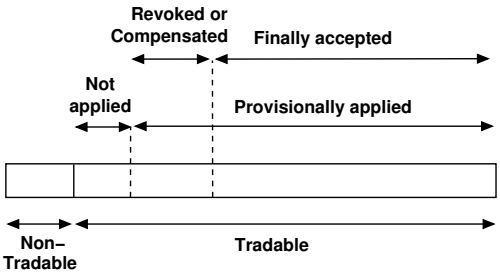

Figure 4. The set of operations during a system partition and subsets thereof

are invoked (classic throughput metrics), but also in terms of the types of operations that are invoked: those subject to critical integrity constraints and those subject to non-critical constraints.

5.2 Load profile

Operations are classified in two categories: tradable (those with associated non-critical integrity constraints), and non-tradable (those with associated critical constraints). Among those operations that are tradable we find operations that violate their integrity constraints, and would not be applied in a normal mode in similar circumstances. This is considered as a normal delivery of the service. We will denote these by "not applied operations" in Figure 4. We also find those that are provisionally applied (updating some object state). In the latter category we find operations that are later revoked (undone) when the partition failure is recovered from, and the reconciliation of states renders the application of the operation as unacceptable (due to inter-partition conflicts). Figure 4 shows these distinct load profiles.

Our evaluation metrics can be divided in two categories; time-based and operation-based metrics. The first category is typically based on the time spent in some segment of the system life time. The second category is based on the counting of the operations that pass through the system and are treated in one way or the other (subsets from Figure 4).

5.3 Time-based metrics

We consider the following metrics:

- Apparent availability: Probability of (partial) operation at a time point; that is, the average interval that the network is in partial/fully available mode divided by the length of the experiments.
- Time spent in revoking (undoing) operations within the experiment interval.

From the above, we are going to use the availability metric as a measure for improved performance. However, we need to complement this metric with other measurements in order to identify the substance of improvement (i.e., excluding the apparent availability).

The second metric is indicative of the apparent availability. That is, the higher the time spent for revocations the lower is the real availability. As far as time for revoking one operation is concerned, a real application has different values attached to different revocations (compensations). In our experimental setting, we choose to compute this time based on an estimate of an undo-time per operation (referred to as handling time in charts in Section 6, thus turning it into a parameter).

5.4 Operation-based metrics

As mentioned above measurements of apparent availability are only meaningful if they are presented together with an indication of the "loss" from revoked operations. To be specific about the level of service delivered to clients we propose two operation-based metrics:

- The number of operations finally accepted during the whole experimental interval.
- The proportion of revocations over provisionally accepted operations.

In addition, it may be interesting to study the proportion of revocations over total arrived operations in a degraded scenario for comparing different variations of reconciliation protocols.

6 Evaluation

In this section we present an experimental evaluation of the replication and continuous service reconciliation protocols. As a baseline, we consider two alternatives: first, a system that does not trade availability for consistency (using pessimistic protocols), and a second optimistic replication and reconciliation service. The second reconciliation protocol does not accept new operations during the reconciliation phase (similar to those presented in earlier work [2]).

6.1 Simulation setup

The simulations were performed with J-Sim [24] using the event-based simulation engine. A simple middleware was constructed and the test application described in Section 2 was implemented on top of it. The middleware is based on the architecture in Figure 2. However, as the main

Table 1. Simulation parameters

Number of runs	100
Number of objects	100
Number of constraints	30
Number of critical constraints	10
Simulation time	70 [s]
Number of nodes	50
Number of clients	30
Mean network delay	0.1 [s]
Normal system load	120 [ops/s]

goal of the implementation was to evaluate the reconciliation protocol some parts of the system have been simplified. The group communication component is for example simulated using a network component that also provides a group membership service. This allows fault injection and network delays to be controlled. Faults are injected by configuring the location service component to resolve object lookups in the same way as if there had been a network partition. This is done in the beginning of each simulation run.

The simulations parameters that were constant in all experiments are shown in Table 1. We base these figures on data provided by industry partners, with real applications that can benefit from partition tolerance, in the DeDiSys project.

6.2 Results

In this section we provide the results of the performed simulations.

In all of the following curves we compare three different protocol behaviours. All three measurements are performed using the same application and random seeds. Moreover, the middleware implementations only differ in the places where the replication and reconciliation differ. The first curve ("continuous") in each graph shows a middleware that acts optimistically during the partition fault, and then uses the continuous service reconciliation (CS) protocol to merge the results. The first baseline that does not accept invocations during reconciliation is denoted as "stop-the-world". Finally, the last ("pessimistic") shows the results for a pessimistic middleware which does not accept invocations during a partitioned state.

The effect of handling rate In Figure 5 the apparent availability is plotted against the handling rate of the reconciliation manager. This rate is an estimate of the average time taken to reconsider a provisionally accepted operation, replay it, and potentially undo it. The partition lasted for 10 seconds in each run. The 95% confidence intervals are within 0.35% for all measurement points. The pessimistic approach gives just over 85% independently of

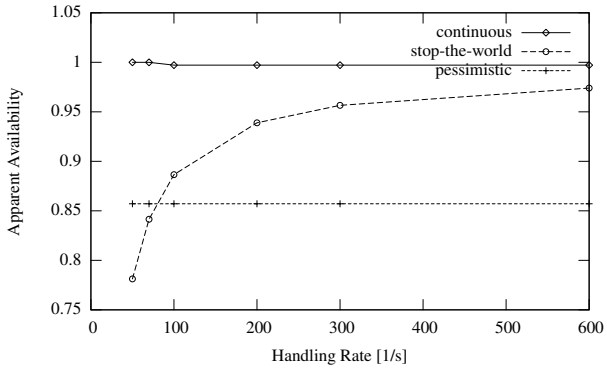

Figure 5. Apparent Availability

the nature of operations that are potentially revocable (since they are never run). This is natural since no operations are performed during partitions. The continuous service protocol manages to supply nearly full availability except for the small effect given by the time spent installing the new state. However, the availability of "stop-the-world" depends very much on the length of the reconciliation phase, which in turn is decided by how fast the handling rate of the reconciliation manager is.

There is an anomaly for the CS protocol for small handling rates. There is no period of unavailability for these rates. The reason is that the protocol will never reach the stop state during the simulation time, and thus never become unavailable. The termination proof from [1] gives that a condition for the termination is $H > \left(\frac{T_D+7d}{T_F-9d}\right) C \cdot I$ where H is the (worst case) handling rate, T_D the partition duration, d a bound on message and service time, T_F the time until next fault (in these runs the end of the simulation), C the number of clients, and I the (worst case) invocation rate for each client. If we put the (average) numbers from our simulations in this inequality we find that the handling rate must be at least 137 to *guarantee* termination. In the figure we see that termination actually occurs for rates over 100 (indicated by the fact that the CS protocol drops from full availability to just under 100%).

As the results in Figure 5 only give the apparent availability (as discussed in Section 5) we need also to compare the second availability metric, which is how many operations we have finally accepted.

In Figure 6 the relative increase of finally accepted operations compared to the pessimistic approach is plotted against the handling rate. This graph is based on the same experiment as Figure 5. The 95% confidence intervals are within 1% for all measurement points. The optimistic approaches achieve better as handling rate increases. For large enough handling rates they give significantly better results compared to the pessimistic approach. The CS reconcilia-

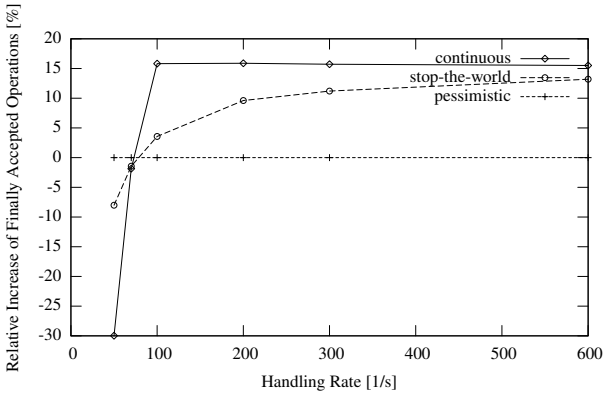

Figure 6. Increase of Finally Accepted Operations

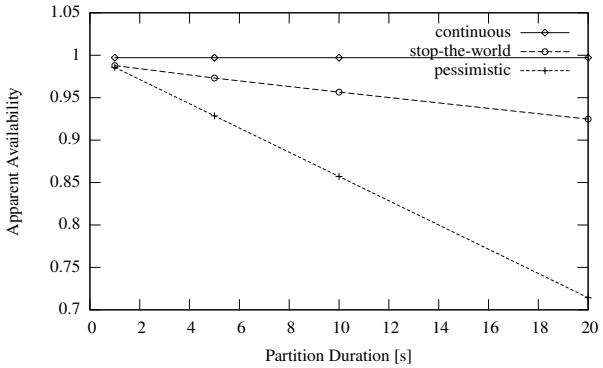

Figure 7. Apparent Availability

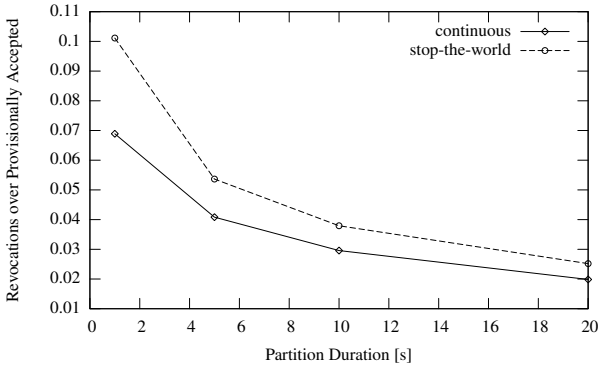

Figure 8. Revocations over Provisionally Accepted

tion protocol only gives distinctly better results than "stop the world" for handling between 100 and 300 operations per second. However, as the handling rate increases further the difference becomes marginal.

This plot indicates that an estimate of the average handling rate, based on profiling the application, is appropriate as a guideline before selecting the CS protocol in a reconfigurable middleware.

The effect of partition duration There are applications, like telecommunication, where partitions do occur but a lot of effort is spent to make them as short as possible so that acting pessimistically will not cause a big decrease in availability. In Figure 7 we see the effect that the partition duration has on the apparent availability. For long enough partitions the only approach that gives acceptable results is the continuous service reconciliation. The confidence intervals for this graph are within 0.1% for all measurement points.

Both of the optimistic reconciliation protocols considered here are operation based. That is, they use a log of operations that were performed in the degraded mode. One can also perform state-based reconciliation where only the current state of the partitions is used to construct the new state. A state based reconciliation scheme might give equally high apparent availability as the continuous service protocol but instead it might suffer in terms of finally accepted operations [2].

A very interesting metric is the number of revocations over provisionally accepted operations. This is the proportion of operations that the client thinks have been performed but which must be revoked/compensated. This is related to, but should not be confused with, the collision probability calculated by Grey et al. [11] to be proportional to the square of the number of operations. Wang et al. [23] have investigated the conflict rate for file systems. Common for these two metrics is that they consider two replicas to be in conflict if they have been updated concurrently. In our model, on the other hand, a conflict occurs only as the result of the violation of some integrity constraint. Such violations can be caused by concurrent updates, but not necessarily.

In Figure 8 we see that as the partition duration increases the ratio of revoked operations decreases. This is a bit counter-intuitive, one would expect the opposite. However, there is an explanation to this phenomenon. The cause lies in the fact that in our synthetic application two partitions perform similar kinds of client calls. This means that an operation which has been successfully applied in one partition is likely to be compatible with changes that have occurred in the other as well. The longer the partition lasts, the more operations are performed and the risk of different types of operations in the two partitions decreases. Naturally, this behaviour depends on the nature of the integrity constraints and thus on the application. The confidence intervals are within 6.9% for all measurement points.

The effect of load So far the experiments have been performed with a constant arrival rate of 120 operations per

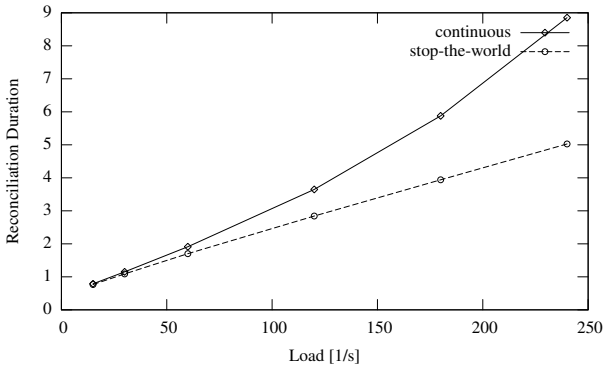

Figure 9. Reconciliation Duration

second. To see the effect of load we have plotted the reconciliation duration against load in Figure 9. Here, the 95% confidence intervalse are within 1.6% for all measurement points. The handling rate for this experiment was 300 actions per second. This figure might seem high compared to the load. However, the reconciliation process is performed at a single node which means that no network communication is needed. As can be seen in the figure the continuous service protocol suffers more than the other protocols under heavy load; especially, as it approaches the maximum load. However, this does not translate to less apparent availability as in the case of stop-the-world. The only period of unavailability for the CS protocol is during the time between the continuous servers receive a stop message from the reconciliation manager and the time to receive the installed state. The length of this period is not affected by the length of the reconciliation phase. Thus, the apparent availability of CS is not decreased (as was shown in Figure 5).

7 Related Work

In this section we will discuss how the problem of reconciliation after network partitions has been dealt with in the literature. For more references on related topics there is an excellent survey on optimistic replication by Saito and Shapiro [19]. There is also an earlier survey discussing consistency in partitioned networks by Davidson et al. [8].

The CS protocol was recently presented as a formalisation in timed I/O automata [1]. Earlier studies [2] have compared different versions of reconciliation protocols but none of them with the feature of continuously serving during reconciliation. Gray et al. [11] address the problem of update everywhere and propose a solution based on a two-tier architecture and tentative operations. However, they do not target full network partitions but individual nodes that join and leave the system (which is a special case of partition). Bayou [22] is a distributed storage system that is adapted for mobile environments. It allows updates to occur in a partitioned system. Bayou can in principle deal with integrity constraints. However, there is a limitation in the sense that a primary server must be able to commit operations locally (this prevents later revocations). This makes the use of system wide integrity constraints hard or impossible.

Some work has been done on partitionable systems where integrity constraints are not considered, which simplifies reconciliation. Babaouglu et al. [3] present a method for dealing with network partitions. They propose a solution that provides primitives for dealing with shared state. They do not elaborate on dealing with writes in all partitions except suggesting tentative writes that can be undone if conflicts occur. Moser et al. [15] have designed a fault-tolerant CORBA extension that is able to deal with node crashes as well as network partitions. There is also a reconciliation scheme described in [16]. The idea is to keep a primary for each object. The states of these primaries are transferred to the secondaries on reunification. In addition, operations that are performed on the secondaries during degraded mode are reapplied during the reconciliation phase. This approach is not directly applicable with integrity constraints.

There are some systems that use more advanced optimistic replication techniques, which allow the degree to which inconsistencies are allowed to be configured. None of these protocols are aimed at operating fully in a partitioned system. They therefore do not provide the reconciliation algorithms for such a scenario. However, it is interesting to compare the way they approach configurable consistency with our integrity constraint based approach. Yu and Vahdat [25] use consistency units (conits) to specify the bounds on allowed inconsistency. A conit is a set of three values representing "numerical error", "order error" and "staleness". Numerical error defines a weight of writes on a conit that can be applied to all replicas, before update propagation has to occur. Order error defines the number of outstanding write operations that are subject to re-ordering on a single conit. Finally, staleness defines the time update propagation can be delayed. The system does not support partitioning, although the key concept of conits could be used in a partitioned context. In CoRe [10] the principle of specifying consistency is extended to allow the programmer to define consistency using a larger set of parameters. AQua [7] approaches the solution from the other direction: configuration of the allowed consistency in order to increase availability; that is, by allowing availability requirements to be specified. In AQua "quality objects" are used to specify quality of service requirements.

Most works on reconciliation algorithms dealing with constraints after network partition focus on achieving a schedule that satisfies order constraints. Lippe et al. [14] try to order operation logs to avoid conflicts with respect to a *before* relation. However, their algorithm requires a

large set of operation sequences to be enumerated and then compared. The IceCube system [13, 18] also tries to order operations to achieve a consistent final state. However, they do not fully address the problem of integrity constraints that involve several objects. Phatak et al. [17] propose an algorithm that provides reconciliation by either using multiversioning to achieve snapshot isolation or using a reconciliation function given by the client. Snapshot isolation is more pessimistic than our approach and would require a lot of operations to be undone.

8 Conclusions and Future Work

In case of a network partition fault in a distributed system, there are two basic approaches: pessimistic and optimistic replication. We have shown that the optimistic solution does pay off in terms of availability even in systems with data constraints that have to be reconciled later on. Moreover, we have identified the need for additional availability metrics (e.g., number of accepted operations, proportion of revoked operations) to evaluate these systems. Using these metrics, we have evaluated an implementation of a reconciliation protocol [1] that aims at delivering continuous service during the reconciliation protocol.

The results show that for long partition durations this protocol gives the best performance in terms of apparent availability as well as number of applied operations. Moreover for longer partition durations, the risk of having to revoke a previously accepted operation can decrease for some applications.

Naturally, the gain comes with a cost. Apart from the fact that operations have to be revoked or possibly compensated during reconciliation, there will be an overhead associated with this solution. Currently, we are evaluating this protocol as a CORBA extension to make it partition-tolerant. This evaluation will include latency measurements to determine the overhead induced by the protocols. A natural continuation for this work is to extend it to more dynamic environments where partitions are more frequent and where the network topology is constantly changing.

The current implementation updates replicas with the installed state by sending the entire state. This is obviously not reasonable in a system with a large state. A relatively easy modification is to send increments, that represent changes to the state of various replicas.

9 Acknowledgments

This work has been supported by the FP6 IST project DeDiSys on Dependable Distributed Systems. The second author was partially supported by University of Luxembourg during preparation of this manuscript.

References

[1] M. Asplund and S. Nadjm-Tehrani. Formalising reconciliation in partitionable networks with distributed services. In M. Butler, C. Jones, A. Romanovsky, and E. Troubitsyna, editors, *Rigorous Development of Complex Fault-Tolerant Systems*, volume 4157 of *Lecture Notes in Computer Science*, pages 37–58. Springer-Verlag, 2006.

[2] M. Asplund and S. Nadjm-Tehrani. Post-partition reconciliation protocols for maintaining consistency. In *Proceedings of the 21st ACM/SIGAPP symposium on Applied computing*, April 2006.

[3] Ö. Babaoglu, A. Bartoli, and G. Dini. Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems. *IEEE Trans. Comput.*, 46(6):642–658, 1997.

[4] G. Badishi, G. Caronni, I. Keidar, R. Rom, and G. Scott. Deleting files in the celeste peer-to-peer storage system. In *SRDS'06: Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems*, October 2006.

[5] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[6] S. Beyer, M. Bañuls, P. Galdámez, J. Osrael, and F. D. Muñoz-Escoí. Increasing availability in a replicated partionable distributed object system. In *Proceedings of the Fourth International Symposium on Parallel and Distributed Processing and Applications (ISPA'2006)*. Springer–Verlag, 2006.

[7] M. Cukier, J. Ren, C. Sabnis, D. Henke, J. Pistole, W. H. Sanders, D. E. Bakken, M. E. Berman, D. A. Karr, and R. E. Schantz. Aqua: An adaptive architecture that provides dependable distributed objects. In *SRDS '98: Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, page 245, Washington, DC, USA, 1998. IEEE Computer Society.

[8] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370, 1985.

[9] DeDiSys. European IST FP6 DeDiSys Project. http://www.dedisys.org, 2006.

[10] C. Ferdean and M. Makpangou. A generic and flexible model for replica consistency management. In *ICDCIT*, pages 204–209, 2004.

[11] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM Press.

[12] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems*, chapter 5, pages 97–145. ACM Press, Addison-Wesley, 2nd edition, 1993.

[13] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The icecube approach to the reconciliation of divergent replicas. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 210–218, New York, NY, USA, 2001. ACM Press.

[14] E. Lippe and N. van Oosterom. Operation-based merging. In *SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments*, pages 78–87, New York, NY, USA, 1992. ACM Press.

[15] L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan. Consistent object replication in the eternal system. *Theor. Pract. Object Syst.*, 4(2):81–92, 1998.

[16] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Replica consistency of corba objects in partitionable distributed systems. *Distributed Systems Engineering*, 4(3):139–150, 1997.

[17] S. H. Phatak and B. Nath. Transaction-centric reconciliation in disconnected client-server databases. *Mob. Netw. Appl.*, 9(5):459–471, 2004.

[18] N. Preguica, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. *Lecture Notes in Computer Science*, 2888:38–55, October 2003.

[19] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[20] D. P. Siewiorek and R. S. Swarz. *Reliable computer systems (3rd ed.): design and evaluation*. A. K. Peters, Ltd., Natick, MA, USA, 1998.

[21] D. Szentivanyi and S. Nadjm-Tehrani. Middleware support for fault tolerance. In Q. Mahmoud, editor, *Middleware for Communications*. John Wiley & Sons, 2004.

[22] D. B. Terry, M. M. Theimer, K. 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 Press.

[23] A.-I. Wang, P. L. Reiher, R. Bagrodia, and G. H. Kuenning. Understanding the behavior of the conflict-rate metric in optimistic peer replication. In *DEXA '02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 757–764, Washington, DC, USA, 2002. IEEE Computer Society.

[24] H. ying Tyan. *Design, realization and evaluation of a component-based software architecture for network simulation*. PhD thesis, Department of Electrical Engineering, Ohio State University, 2001.

[25] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.*, 20(3):239–282, 2002.