BFT Protocols Under Fire

Atul Singh[†‡] Tathagata Das[⋆] Petros Maniatis[ϕ] Peter Druschel[†] Timothy Roscoe[χ]
[†]*MPI-SWS*, [‡]*Rice University*, [⋆]*IIT Kharagpur*, [ϕ]*Intel Research Berkeley*, [χ]*ETH Zürich*

Abstract

Much recent work on Byzantine state machine replication focuses on protocols with improved performance under benign conditions (LANs, homogeneous replicas, limited crash faults), with relatively little evaluation under typical, practical conditions (WAN delays, packet loss, transient disconnection, shared resources). This makes it difficult for system designers to choose the appropriate protocol for a real target deployment. Moreover, most protocol implementations differ in their choice of runtime environment, crypto library, and transport, hindering direct protocol comparisons even under similar conditions.

We present a simulation environment for such protocols that combines a declarative networking system with a robust network simulator. Protocols can be rapidly implemented from pseudocode in the high-level declarative language of the former, while network conditions and (measured) costs of communication packages and crypto primitives can be plugged into the latter. We show that the resulting simulator faithfully predicts the performance of native protocol implementations, both as published and as measured in our local network.

We use the simulator to compare representative protocols under identical conditions and rapidly explore the effects of changes in the costs of crypto operations, workloads, network conditions and faults. For example, we show that Zyzzyva outperforms protocols like PBFT and Q/U under most but not all conditions, indicating that one-size-fits-all protocols may be hard if not impossible to design in practice.

1 Introduction

Byzantine Fault-Tolerant (BFT) protocols for replicated systems have received considerable attention in the systems research community [3, 7, 9], for applications including replicated file systems [6], backup [4], and block stores [10]. Such systems are progressively becoming more mature, as evidenced by recent designs sufficiently fine-tuned and optimized to approach the performance of centralized [14] or crash-fault only [10] systems in some settings.

Much of the attraction of such systems stems from the combination of a simple programming interface with provable correctness properties under a strong adversarial model. All a programmer need do is write her server application as a sequential, misbehavior-oblivious state machine; available BFT protocols can replicate such application state machines across a population of replica servers, guaranteeing *safety* and *liveness* even in the face of a bounded number of arbitrarily faulty (Byzantine) replicas among them. The safety property (*linearizability*) ensures that requests are executed sequentially under a single schedule consistent with the order seen by clients. The liveness property ensures that all requests from correct clients are eventually executed.

Though these protocols carefully address such correctness properties, their authors spend less time and effort evaluating BFT protocols under severe—yet benign—failures. In fact, they often optimize under the assumption that such failures do not occur. For example, Zyzzyva [14] obtains a great performance boost under the assumption that all replica servers have good, predictable latency[1] to their clients, whereas Q/U [3] significantly improves its performance over its precursors assuming no service object is being updated by more than one client at a time.

Unfortunately, even in the absence of malice, deviations from expected behavior can wreak havoc with complex protocols. As an example from the non-Byzantine world, Junqueira et al. [11] have shown that though the "fast" version of Paxos consensus[2] operates in fewer rounds than the "classic" version of Paxos (presumably resulting in lower request latency), it is nevertheless more vulnerable to variability in replica connectivity. Because fast Paxos requires more replicas (two-thirds of

the population) to participate in a round, it is as slow as the slowest of the fastest two-thirds of the population; in contrast, classic Paxos is only as slow as the median of the replicas. As a result, under particularly skewed replica connectivity distributions, the two rounds of fast Paxos can be slower than the three rounds of classic Paxos. This is the flavor of understanding we seek in this paper for BFT protocols. We wish to shed light on the behavior of BFT replication protocols under adverse, yet benign, conditions that do not affect correctness, but may affect tangible performance metrics such as latency, throughput, and configuration stability.

As we approach this objective, we rely on simulation. We present `BFTSim`, a simulation framework that couples a high-level protocol specification language and execution system based on P2 [18] with a computation-aware network simulator built atop `ns-2` [1] (Section 3). P2's declarative networking language (OverLog) allows us to capture the salient points of each protocol without drowning in the details of particular thread packages, cryptographic primitive implementations, and messaging modules. `ns-2`'s network simulation enables us to explore a range of network conditions that typical testbeds cannot easily address. Using this platform, we implemented from scratch three protocols: the original PBFT [6], Q/U [3], and Zyzzyva [14]. We validate our simulated protocols against published results under corresponding network conditions. Though welcome, this is surprising, given that all three systems depend on different types of runtime libraries and thread packages, and leads us to suspect that a protocol's performance characteristics are primarily inherent in its high-level design, not the particulars of its implementation.

Armed with our simulator, we make an "apples to apples" comparison of several BFT protocols under identical conditions. Then, we expose the protocols to benign conditions that push them outside their comfort zone (and outside the parameter space typically exercised in the literature), but well within the realm of possibility in real-world deployment scenarios. Specifically, we explore latency and bandwidth heterogeneity between clients and replicas, and among replicas themselves, packet loss, and timeout misconfiguration (Section 4). Our primary goal is to test conventional (or published) wisdom with regards to which protocol or protocol type is better than which; it is rare that "one size fits all" in any engineering discipline, so understanding the envelope of network conditions under which a clear winner emerges can be invaluable.

While we have only begun to explore the potential of our methodology, our study has already led to some interesting discoveries. Among those, perhaps the broadest statement we can make is that though agreement protocols offer hands down the best throughput, quorum-based protocols tend to offer lower latency in wide-area settings. Zyzzyva, the current state-of-the-art agreement-based protocol provides almost universally the best throughput in our experiments, except in a few cases. First, Zyzzyva is dependent on timeout settings at its clients that are closely tied to client-replica latencies; when those latencies are not uniform, Zyzzyva tends to fall back to behavior similar to a two-phase quorum protocol like HQ [9], as long as there is no write contention. Second, with large request sizes, Zyzzyva's throughput drops and falls slightly below Q/U's and PBFT's with batching, since its primary is required to send full requests to all the backup replicas. Lastly, under high loss rates, Zyzzyva tends to compensate quickly and expensively, causing its response time to exceed that of the more mellow Q/U.

Section 2 provides some background on BFT replicated state machines. In Section 3, we explain our experimental methodology, describe our simulation environment, and validate it by comparing its predictions with published performance results on several existing BFT protocols we have implemented in `BFTSim`. Section 4 presents results of a comparative evaluation of BFT protocols under a wide range of conditions. We discuss related works in Section 5 and close with future work and conclusions in Section 6.

2 Background

In this section, we discuss the work on which this paper is based: BFT replicated state machines. Specifically, we outline the basic machinery of the protocols we study in the rest of this paper: PBFT by Castro and Liskov [6], Q/U by Abd-El Malek et al. [3], and Zyzzyva and Zyzzyva5 by Kotla et al. [14].

At a high level, all such protocols share the basic objective of assigning each client request a unique order in the global service history, and executing it in that order. Agreement-based protocols such as PBFT first have the replicas communicate with each other to agree on the sequence number of a new request and, when agreed, execute that request after they have executed all preceding requests in that order. PBFT has a three-phase agreement protocol among replicas before it executes a request. Quorum protocols, like Q/U, instead restrict their communication to be only between clients and replicas—as opposed to among replicas; each replica assigns a sequence number to a request and executes it as long as the submitting client appears to have a current picture of the whole replica population, otherwise uses conflict resolution to bring enough replicas up to speed. Q/U has a one-phase protocol in the fault-free case, but when faults occur or clients contend to write the same object the protocol has more phases. Zyzzyva is a

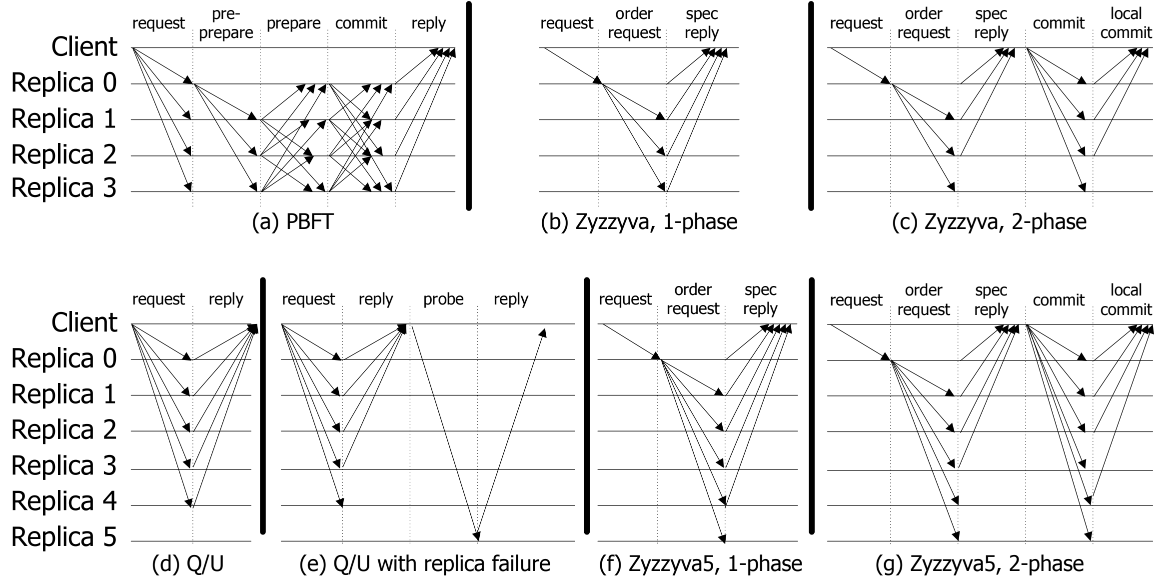

Figure 1: A high-level view of PBFT, Q/U, and Zyzzyva.

hybrid agreement/quorum protocol that shares some of PBFT's characteristics (a distinguished primary replica and an explicit "view change" to recover from a faulty primary); however, whereas PBFT performs extra work during agreement to ensure it can deal with primary failures, Zyzzyva offloads that extra work to the actual recovery from a primary failure, resulting in a leaner protocol when the fault-free case is the common one. Furthermore, Zyzzyva includes a fast path for unanimous agreement on a request's sequence number.

In more detail, PBFT requires $3f+1$ replicas to tolerate f faulty replicas. A client broadcasts its request to all replicas. The primary among them assigns the request a sequence number, and broadcasts that assignment to the replica population in a PrePrepare message. A backup that receives such an assignment acknowledges it and synchronizes with all other replicas on this assignment by broadcasting a Prepare message to the population. As soon as a replica has received a quorum of $2f+1$ Prepare messages, it promises to commit the request at that sequence number by broadcasting a Commit message. When a replica has seen a quorum of $2f+1$ such promises for the same request and sequence number, it finally accepts that assignment and executes the request in its local state after it has executed all other requests with lower sequence number, sending a Reply message to the client with the result. A client accepts the result if $f+1$ replicas send matching Reply messages, and otherwise retransmits the request. See Figure 1(a) for an illustration.

In contrast, Query/Update (Q/U) is a single-phase quorum-based protocol that tolerates up to f faulty replicas in a population of $5f+1$. Clients cache replica histories (the request sequence known by a replica), which they include in requests to replicas, and which they update using replies from replicas. These histories allow a replica that receives a client request to optimistically execute it immediately, as long as its request history is reflected in the client's view. When a client receives replies, as long as a quorum of $4f+1$ have optimistically executed its request, it completes. Normally a client only contacts its "preferred quorum" of replicas instead of the whole population; if some of the quorum replicas are slow to respond, a client might engage more replicas via a "probe" hoping to complete the quorum. If between $2f+1$ and $4f$ replies accept the request but others refuse due to a stale replica history, the client infers there exists a concurrent request from another client. Q/U provides a conflict resolution mechanism in which clients back off and later resubmit requests, after repairing the replicas to make them consistent. Figure 1(d) shows the best case for a client's request, whereas Figure 1(e) illustrates the probing mechanism.

Zyzzyva uses a primary to order requests, like PBFT, and also requires $3f+1$ replicas to tolerate f faults. Clients in Zyzzyva send the request only to the primary. Once the primary has ordered a request, it submits it in an OrderReq message to the replicas, which respond to the client immediately as in Q/U. In failure-free and synchronous executions in which all $3f+1$ replicas return the same response to the client, Zyzzyva is efficient since requests complete in 3 message delays and, unlike Q/U, write contention by multiple clients is mitigated by the primary's ordering of requests (Figure 1(b)). When some

replicas are slower or faulty and the client receives between $2f+1$ and $3f$ matching responses, it must marshal those responses and resend them to the replicas, to convince them that a quorum of $2f+1$ has chosen the same ordering for the request. If it receives $2f+1$ matching responses to this second phase, it completes the request in 5 message delays (Figure 1(c)).

To trade off fewer message delays for more replicas in high-jitter conditions, or when some replicas are faulty or slow, the authors propose Zyzzyva5, which requires $5f+1$ replicas and only $4f+1$ optimistic executions from replicas to progress in 3 message delays—up to f replicas can be slow or faulty and the single-phase protocol will still complete (Figure 1(f)). With fewer than $4f+1$ replies, Zyzzyva5 also reverts to two-phase operation (Figure 1(g)). Finally, Zyzzyva's view change protocol is more heavy-weight and complex than in PBFT, and the authors present results where one replica is faulty (mute) and observe that Zyzyvva is slower than PBFT in this situation; however a recent optimization [15] improves Zyzzyva's performance under faults. In this optimization, clients explicitly permit replicas to send a response only after having committed it (as opposed to tentatively). For such client requests, replicas agree on orderings similarly to the second phase of PBFT, and clients need not initiate a second protocol phase.

Each of these protocols achieves high performance by focussing on a specific expected common scenario (failures, latency skew, contention level, etc.). As a consequence, each is ingeniously optimized in a different way. This makes it hard for a developer with a requirement for BFT to choose a high-performance protocol for her specific application. Worse, evaluations of individual protocols in the literature tend to use different scenario parameters. Our aim in this paper is to enable, and perform, "apples to apples" comparison of such protocols as a first step in establishing their performance envelopes.

3 Methodology

We now describe in detail our approach to comparing BFT protocols experimentally. We have built `BFTSim`, which combines a declarative front end to specify BFT protocols with a back-end simulator based on the widely used `ns-2` simulator [1]. This allows us to rapidly implement protocols based on pseudocode descriptions, evaluate their performance under a variety of conditions, and isolate the effects of implementation artifacts on the core performance of each protocol.

Using simulation in this manner raises legitimate concerns about fidelity: on what basis can we claim that results from `BFTSim` are indicative of real-world performance? We describe our validation of `BFTSim` in section 3.3 below, where we reproduce published results from real implementations.

A further concern is the effort of re-implementing a published protocol inside `BFTSim`, including characterizing the costs of CPU-bound operations such as cryptographic primitives. We report on our experience doing this in section 3.4.

However, a first question is: why use simulation at all? In other words, why not simply run existing implementations of protocols in a real networking environment, or one emulated by a system like ModelNet [23]?

3.1 Why Simulation?

Subject to fidelity concerns which we address in section 3.3 below, there are compelling advantages to simulation for comparing protocols and exploring their performance envelopes: the parameter space for network conditions can be systematically and relatively easily explored under controlled conditions.

There are highly pragmatic reasons to adopt simulation. Many implementations of BFT protocols from research groups are not available at publication time, due to inevitable time pressure. Comparing the performance of protocols based on their published descriptions without requiring re-implementation in C, C++, or Java is a useful capability.

Even implementations that are available vary widely in choice of programming language, runtime, OS, cryptographic library, messaging, thread model, etc., making it hard to identify precisely the factors responsible for performance or, in some cases, to even run under emulation environments such as Emulab due to incompatibilities. For example, Q/U uses SunRPC over TCP as the communication framework while PBFT uses a custom reliability layer over UDP; Q/U is written in C++ while PBFT is written in C; Q/U uses HMAC for authenticators while Zyzzyva and PBFT use UMAC. Our results below show that performance is generally either network-bound or dominated by the CPU costs of crypto operations. We can build faithful models of the performance of such implementations based on the costs of a small number of operations, and hence directly compare algorithms in a common framework.

Furthermore, simulation makes it straightforward to vary parameters that are non-network related, and so cannot be captured with real hardware in a framework such as ModelNet. For example, since CPU time spent in cryptographic operations is at present often the dominating factor in the performance of these protocols, we can explore the future effect of faster (or parallel) cryptographic operations without requiring access to faster hardware.

Compared to a formal analytical evaluation, simulation can go where closed-form equations are difficult to derive. Existing literature presents analyses such as fault-

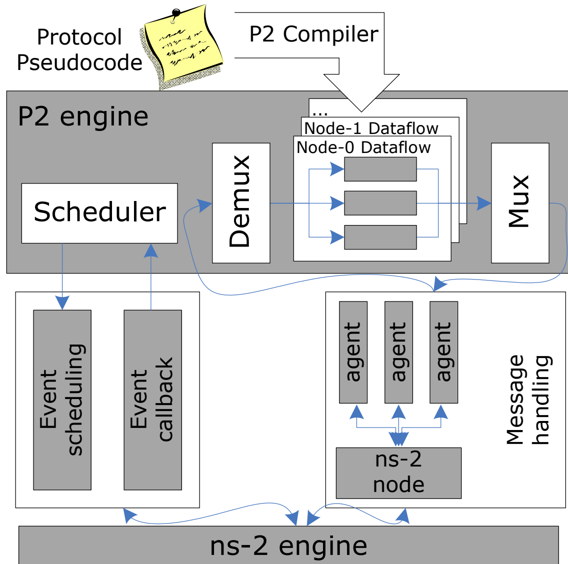

Figure 2: The BFTSim software architecture.

scalability by counting messages exchanged and cryptographic operations performed [9, 14], but it is hard to analyze the dynamic behavior of these protocols, especially for costs intended to be amortized over different sequences of requests. For example, the interaction of client retransmissions, pipeline depth in protocols with a bounded number of requests in flight, and request batching can have a complex effect on response times. Systematic evaluation using faithful simulation can answer these questions with greater ease.

A final motivation for simulation is pedagogical. It is widely believed that BFT protocols are complex to design, implement, and evaluate [8], and that it is hard to understand what aspect of a given protocol gives rise to its performance characteristics. A simple and succinct re-implementation in a declarative language which shows (under simulation) the same performance characteristics as a published C++ implementation is a powerful tool for understanding such issues. We believe that BFTSim may lead the many BFT protocols available to greater accessibility.

3.2 The Design of BFTSim

BFTSim consists of several components (Figure 2). First, protocols are implemented in, and executed by, the front end of the P2 declarative networking engine [18]. P2 allows concise specification and rapid prototyping of BFT protocols, in a manner closely following pseudocode descriptions in publications: we specify the core features of these protocols in a small number of declarative rules.

A rule is of the form "`result :- preconditions.`" and is interpreted as "the result is produced when all preconditions are met." For example, consider the following simplified rule for PBFT:

```
initPrePrepare(@A, T, OP, CID) :-
        request(@A, T, OP, CID),
        cachedReply(@A, CID, T1, REPLY), T > T1,
        isPrimary(@A, A, V).
```

It means that when a `request` tuple arrives at node `A` from the client with identifier `CID`, and if the timestamp `T` is more recent than the last reply sent to the same client (condition `T>T1`), then the primary (condition `isPrimary`) produces a `initPrePrepare` tuple to start the protocol for this request. The location specifier `@A` is used to specify on which node this particular rule is executed.

P2 compiles such descriptions to a software dataflow graph, which in the case of BFTSim includes elements to capture the timing characteristics of CPU-intensive functions without necessarily performing them. Our hypothesis (subsequently borne out by validation) was that we can accurately simulate existing BFT protocols by incorporating the cost of a small number of key primitives. We started with two: cryptographic operations and network operations (we assume for now that disk access costs are negligible, though modeling disk activity explicitly may be useful in some settings). To feed our simulator with the cost of these primitives, we micro-benchmarked the PBFT and Q/U codebases to find the cost of these primitives with varying payload sizes. Our simulator uses this information to appropriately delay message handling, but we can also vary the parameters to explore the impact of future hardware performance.

In BFTSim, a separate P2 dataflow graph is generated per simulated node, but all dataflow graphs are hosted within a single instance of the P2 engine. P2's scheduler interacts with the ns-2 engine via ns-2's event scheduling and callback interfaces to simulate the transmission of messages and to inject events such as node failures. Those interfaces in turn connect with a single ns-2 instance, initialized with the network model under simulation. Message traffic to and from a P2-simulated node is handled by a dedicated UDP agent within ns-2[3]. Note that both P2 and ns-2 are discrete-event-based systems. We use ns-2's time base to drive the system. P2's events, such as callbacks and event registrations, are wrapped in ns-2 events and scheduled via ns-2's scheduler. BFTSim is currently single-threaded and single-host.

3.3 Validation

In this section we compare the published performance of several BFT protocols with the results generated by our implementation of these protocols in BFTSim under comparable (but simulated) conditions. These results therefore yield no new insight into BFT protocols, rather they

serve to show that BFTSim succeeds in capturing the important performance characteristics of the protocols.

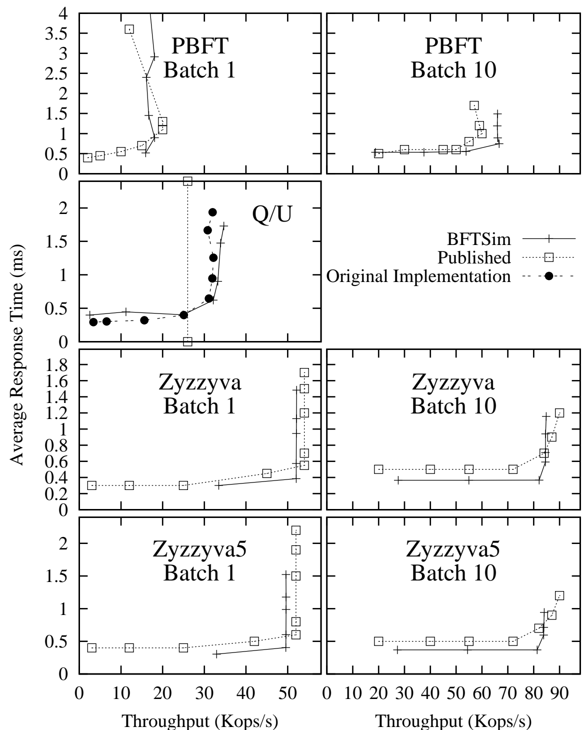

Figure 3: Baseline validation. Note that, to enhance readability, ranges differ for the y axis across protocols (i.e., rows) and for the x axis between batch sizes (i.e., columns).

We present a small selection of our validation comparisons for three protocols: PBFT [7], Q/U [3], and Zyzzyva/Zyzzyva5 [14]. PBFT was chosen because it is widely regarded as the "baseline" for practical BFT implementations. Q/U provides variety: it is representative of a class of quorum-based protocols. Finally, the recent Zyzzyva is considered state-of-the-art, and exhibits many high-level optimizations.

For all these protocols, we compare BFTSim's implementation results to either published results or the protocol authors' implementations executing in our local cluster. Table 1 lists our validation references. Our validation concentrates on latency-throughput curves for all protocols in typical "ideal" network conditions, as well as under node crashes.

3.3.1 Experimental Setup

We simulated a star network topology, where both client and replica nodes are connected to each other via a hub node. Each link is a duplex link and we set a one-way delay of 0.04ms and bandwidth of 1000Mbps on each link. This gives an RTT of 0.16ms between any pair of nodes, matching our local cluster setup. These values are also similar to those reported by the authors [13] to obtain the Zyzzyva/Zyzzyva5 and PBFT results that we use for validation. Both PBFT and Zyzzyva exploit hardware multicast to optimize the one-to-all communication pattern among replicas. BFTSim accounts for multicast by charging for a single message digest, message send, and authenticator calculation for multicast messages, but currently simulates multiple unicast messages at the ns-2 level.

Since the literature only provides peak throughput results for Q/U, we obtained the authors' implementation and ran it in our local cluster; each machine has a dual-core 2.8 GHz AMD processor with 4GB of memory, running a 2.6.20 Linux kernel.

3.3.2 Cost of Key Operations

We measured the costs of three primitive operations common to all protocols: calculating the message digest, generating a MAC and authenticator, and sending a message with varying payload sizes. We instrumented the PBFT and Q/U codebases to measure these costs.

To measure the host processing delays required to transmit and receive messages of a given size, we performed a simple ping-pong test between two machines connected by gigabit ethernet. We measured the total time taken to send and receive a response of equal size. We then simulated the same experiment in ns-2 (without BFTSim) to determine the round-trip network delays. For each message size, we then subtracted the simulated network delays from the corresponding measured round-trip times to determine the host processing delays. We used linear interpolation to obtain piece-wise linear formulae for estimating the send and receive costs as a function of the message size, with a confidence of 99%.

All protocols use MD5 for calculating message digests[4]. Based on our benchmarking results, we obtained the following linear interpolation to estimate the cost of an MD5 digest over d bytes: $digest(d) = (0.0097d + 0.74)\mu$secs, with a confidence of 99.9%. We measured the cost of authenticators by varying the number of MAC operations and used this information directly. Note that Q/U uses HMAC while PBFT and Zyzzyva use UMAC for calculating MACs; our validating simulations are parametrized accordingly.

3.3.3 Baseline Validation

Figure 3 presents latency-throughput curves for BFTSim implementations of PBFT, Q/U, Zyzzyva, and Zyzzyva5, compared to our reference sources. We present results with batch sizes of 1 and 10. We executed each protocol under increasing load by varying the number of clients between 1 and 100. Each client maintains one outstanding request at any time. Each point in the graphs represents an experiment with a given number of clients. The

Table 1: Reference sources for our validation and BFTSim implementation coverage for each protocol.

Protocol	Validated Against	Protocol features *not* present in BFTSim implementations
PBFT	[14]	State transfer, preferred quorums.
Zyzzyva/Zyzzyva5	[14]	State transfer, preferred quorums, separate agreement & execution.
Q/U	Implementation, [3]	In-line repair, multi-object updates.

knee in each curve marks the point of saturation; typically, the peak throughput is reached just beyond and the lowest response time just below the knee.

First, we note that the trends in the latency-throughput curves obtained with BFTSim closely match the reference for all the protocols we studied. The differences in the absolute values are within at most 10%. Because no latency-throughput curves for Q/U were published, we measured the original implementation in our local cluster[5]. The published peak throughput value is also shown.

The results show that our implementations in BFTSim correctly capture the common case behavior of the protocols, and that BFTSim can accurately capture the impact of the key operations on the peak performance of all protocols we implemented and evaluated.

3.3.4 Validating the Silent Replica Case

In this experiment, we make one of the replicas mute for the duration of the experiment. This experiment exercises important code paths for all protocols. In PBFT, performance may improve in this situation since replicas avoid both receiving and verifying messages from the silent replica. In contrast, Zyzzyva's performance is expected to drop in the presence of a faulty replica since it requires clients to perform the costly second phase of the protocol. In Q/U, performance also drops in the presence of a faulty replica because all requests must be processed by the remaining live quorum of 5 replicas, which increases the load on each of these replicas. In the absence of a faulty replica, each replica handles only $(4f + 1)/(5f + 1)$-th of the requests, due to Q/U's preferred quorum optimization.

We present BFTSim's results along with the published results in Figure 4. Because no published results are available for Q/U with a faulty replica, we measured the original implementation in our cluster in the presence of a silent replica for comparison. We observe that BFTSim is able to closely match the published performance of Zyzzyva, Zyzzyva5, PBFT, and Q/U in this configuration.

3.3.5 Validating Fault Scalability

In this experiment, we scale the fault tolerance of PBFT, Zyzzyva, and Q/U and compare the performance predicted by BFTSim with the published results in the Zyzzyva and Q/U papers. At higher values of f, all three protocols need to do more MAC calculations per protocol operation since there are more replicas. PBFT additionally generates more bandwidth overhead at higher values of f since each replica receives $3f$ PREPARE and COMMIT messages. We present results in Figure 5 for values of f between 1 and 3 (we do not validate Zyzzyva5 since our reference source offered no measurements for fault scalability). Again, we observe that BFTSim is able to match published results for all protocols for this experiment.

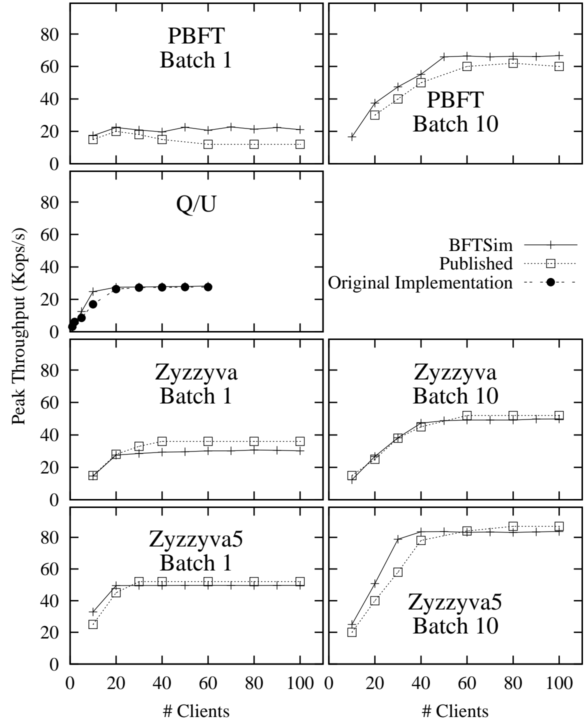

Figure 4: Validation results in the presence of a faulty replica.

3.4 Implementation Experience

One concern with our approach is the effort required to implement protocols within the framework. Here, we report on our experience implementing the protocols presented.

Our PBFT implementation consists of a total of 148 lines of OverLog (P2's specification language), of which 14 are responsible for checkpoint and garbage collection, 38 implement view changes, 9 implement the mechanism to fetch requests, and the remaining 87 provide the main part of the protocol. Our implementation of Zyzzyva is

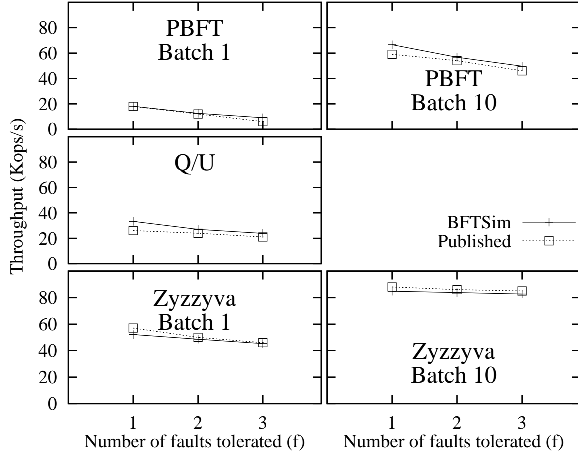

Figure 5: Validation at higher values of f.

slightly more complex: 164 rules, with 13 for checkpoint and garbage collection, 33 for view changes, 24 for handling the second phase, and the remaining 94 for the main part of the protocol. For Q/U, we wrote a total of 88 rules. The key steps in Q/U are represented by the classification (5 rules) and setup (6 rules) phases.

These protocols have served as vehicles for validating the design and fidelity of `BFTSim`. As a result, their implementation in `BFTSim` inevitably co-evolved with the simulator itself. However, the basic components (the P2 front end and `ns-2` back end) are largely unmodified and thus retain their generality, which of course extends far beyond BFT protocols. We have only modified `ns-2` to enable jumbo-frame handling, and we modified and extended P2 in four ways:

- We retargeted P2's dataflow network stack to use `ns-2` agents as opposed to P2's own congestion-controlled UDP elements.
- We added support for compound (nested) tuples. P2's original design stayed close to the relational data model, in which tuples are flat. Since support for complex objects in P2's data model was evolving separately as we were performing this work, we chose to use a simpler but coarser and less efficient approach to structuring data (via explicit nesting of serialized tuples within tuples) for building request batches and the complex view-change messages. Since P2 operates in the virtual time of `BFTSim`, this inefficiency does not affect our results, only the elapsed time required for our simulations.
- The version of P2 on which we based `BFTSim` (version 0.8) did not yet support atomic execution of individual rules. Since `BFTSim` simulates single-threaded, run-to-completion message handlers, we modified the P2 scheduler to complete all processing on outgoing tuples and send them to the `ns-2` agent before any pending incoming tuples were handled.
- We implemented complex imperative computations not involving messaging (such as the view-change logic in PBFT and Zyzzyva) as external plugins ("stages") in P2.

P2's language (OverLog) currently lacks higher-order constructs. This makes it cumbersome, for example, to make the choice of cryptographic primitive transparent in protocol specifications, leading to extra OverLog rules to explicitly specify digests and MACs as per the configuration of the particular protocol. As above, this inefficiency does not affect the (simulated) size or timing of messages transmitted, it merely means simulations take longer to complete.

Based on our experience, we feel a more specialized language might reduce protocol implementation effort, but at these code sizes the benefit is marginal. A more immediate win might be to abstract common operations from existing protocols to allow them to be reused, as well as decoupling the authentication and integrity protocol specifications from their actual implementation (i.e., the specific cryptographic tools used to effect authentication, etc.). These are a topic of our on-going research.

4 Experimental Results

`BFTSim` can be used to conduct a wide variety of experiments. One can compare the performance of different protocols under identical conditions. It is possible to explore the behavior of protocols under a wide range of network conditions (network topology, bandwidth, latency, jitter, packet loss) and under different workloads (distribution of request sizes, overheads, request burstiness). Furthermore, `BFTSim` allows us to easily answer "what-if" questions such as the impact of a protocol feature or a crypto primitive on the performance of a protocol under a given set of conditions.

Due to time and space constraints, we have taken only a first step in exploring the power of `BFTSim`, by evaluating BFT protocols under a wider range of conditions than previously considered. Specifically, we evaluate the effects of batching, workload variation (request size), network conditions (link latency, bandwidth, loss, access link variation), and client timer misconfiguration. We also perform an experiment to explore the potential of a possible protocol optimization in Q/U.

When reporting results, we either show throughput as a function of the number of clients used, or report results with sufficiently many clients to ensure that each protocol reaches its peak throughput. When reporting latency, we use a number of clients that ensures that no protocol is saturated, i.e., requests are not queued.

We use a star network topology to connect clients and replicas. Each node, either a replica or a client, is con-

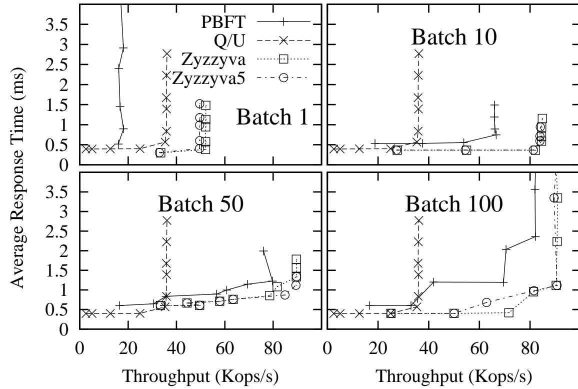

Figure 6: Baseline latency-throughput curves for all protocols, with batch sizes increasing from 1 to 100. Note that Q/U has no batching but appears in all plots for comparison.

nected to a hub of ample capacity using a bidirectional link with configurable latency, bandwidth and loss rate.

4.1 The Effects of Batching

We start with a baseline protocol comparison under typical LAN network conditions (average round-trip time of 0.16 ms, 1 Gbps bandwidth, no packet losses). The requests are no-ops (2-byte payload and no execution overhead). Request batching is used in the agreement-based protocols that support it (all but Q/U). We use the same digest and authentication mechanisms for all protocols (those of the PBFT codebase: MD5 and UMAC).

Figure 6 shows latency-throughput curves for the protocols with increasing batch sizes. As before, each point represents a single experiment with a given number of clients. In agreement-based protocols, the primary delays requests until either a batching timer expires (set to 0.5 ms) or sufficiently many requests (the batch size) have arrived; then, it bundles all new requests into a batch and initiates the protocol. Batching amortizes the messaging overheads and CPU costs of a protocol round over the requests in a batch. In particular, fewer replica-replica messages are sent and fewer digests and authenticators are computed. All three batching-enabled protocols benefit from the technique. Because PBFT has the highest overhead, it enjoys the largest relative improvement from batching: its peak throughput increases by a factor of four.

The Zyzzyva variants are more efficient under the given network conditions, requiring fewer messages and crypto operations than PBFT. As a result, though still fastest in absolute terms, they derive a smaller relative benefit from batching. Furthermore, as batch size increases, the differences between protocols shrink significantly; the Zyzzyva variants become indistinguishable

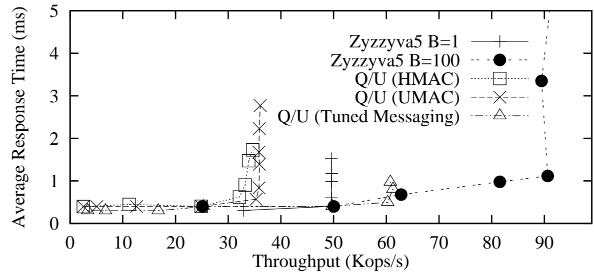

Figure 7: Hypothetical optimized Q/U. Our hypothetical Q/U competes with Zyzzyva5 without batching, but is still inferior to batched Zyzzyva5. We also show Q/U using HMAC, as in its original implementation validated in Figure 3.

with only moderate batching, whereas PBFT achieves throughput within 10% of Zyzzyva at a batch size of 100.

4.1.1 Why is Q/U Worse?

Figure 6 shows that Q/U has significantly lower peak throughput than Zyzzyva5, even with no batching and despite the absence of write contention in our workload. This is somewhat surprising, because in the absence of contention, Q/U only requires a single phase of direct client-replica communication to complete, whereas Zyzzyva5 must relay all requests from clients to replicas via the primary.

We hypothesized that one reason for Q/U's lower throughput is the size of the messages it requires. Q/U replicas include their recent object history in each response[6], and clients send with their requests a vector of such replica histories. This is an important safety requirement in Q/U; it enables non-faulty replicas to execute an operation in a consistent order without any replica-to-replica coordination. In contrast, Zyzzyva replicas include only digests of their history with response messages, and clients send no history with their requests, because the primary orders requests.

We performed an experiment to see how a hypothetical version of Q/U would perform that is optimized for message size. Our hypothetical Q/U variant is assumed to send Zyzzyva-style history digests as a vector in Q/U requests, and sends a single history digest in replica responses. The intuition is that, without faults or write contention, history digests are sufficient to convince a replica to execute an operation at a given point in the history, and replicas need exchange full histories only when they detect an inconsistency. We charge Q/U with Zyzzyva's OrderReq-sized messages with respect to digest computation and message transmission, but leave MAC costs unchanged. This experiment is an example of how `BFTSim` can be used to quickly ask "what-if" questions to explore the potential of possible protocol opti-

mizations.

Figure 7 shows the result. As expected, our hypothetical Q/U is competitive with Zyzzyva5 at a batch size of 1. However, our variant of Q/U is still no match for Zyzzyva5 at a large batch size in terms of peak throughput (though it has slightly lower latency). This seems counter-intuitive at first glance, since Q/U has no "extraneous" traffic to amortize, only client requests and replica responses. However, Q/U's lack of a primary that orders and relays requests to backups incurs extra computation on the critical path: a Q/U replica computes $4f$ MACs to send a response to a client and verifies $4f$ MACs when receiving a request from it, whereas the Zyzzyva5 primary computes and verifies one MAC for each request in a batch of b requests, plus a single authenticator for each ORDERREQ message containing $4f$ MACs for a total of $2{+}8f/b$ MACs per request in a batch. As b increases, Zyzzyva5 tends toward slightly more than 1 MAC generation and 1 MAC verification per client request compared to $8f$ MAC operations in Q/U.

4.1.2 Summary

Batching helps agreement-based protocols to achieve better performance by amortizing their protocol costs over a batch of requests. As we increase the batch size, the importance of protocol efficiency diminishes; as a result, the throughput of PBFT approaches that of Zyzzyva. The results of our experiments also predict that Q/U could benefit significantly from optimizations to reduce its message sizes.

4.2 Varying the Workload

We now turn to study the performance of BFT protocols under varying request sizes. The network conditions remain as above. Figure 8 shows latency-throughput graphs for all protocols when the size of the request (and response) ranges from 2 bytes (as in the experiments above) up to 8 kbytes. Request sizes in this range do occur in practice. Whereas many SQL workloads are reported to have request sizes of around 128 bytes, applications like block storage use larger requests (for example, Hendricks et al. [10] discuss requests containing erasure-coded disk block fragments with sizes of about 6 Kb). We obtained these graphs by choosing a number of clients such that the latency-throughput curve was beyond its characteristic "knee," i.e., the system was saturated (in this set of experiments, between 1 and 80 clients were required). It is striking how increasing payloads diminish the differences among the protocols. Whereas at 2-byte payloads, non-batched PBFT has a little more than a third the throughput of Zyzzyva and Q/U about half, at 8 kbyte payloads all protocols are very close. With batching, PBFT starts out closer to Zyzzyva but, again, the difference vanishes at higher

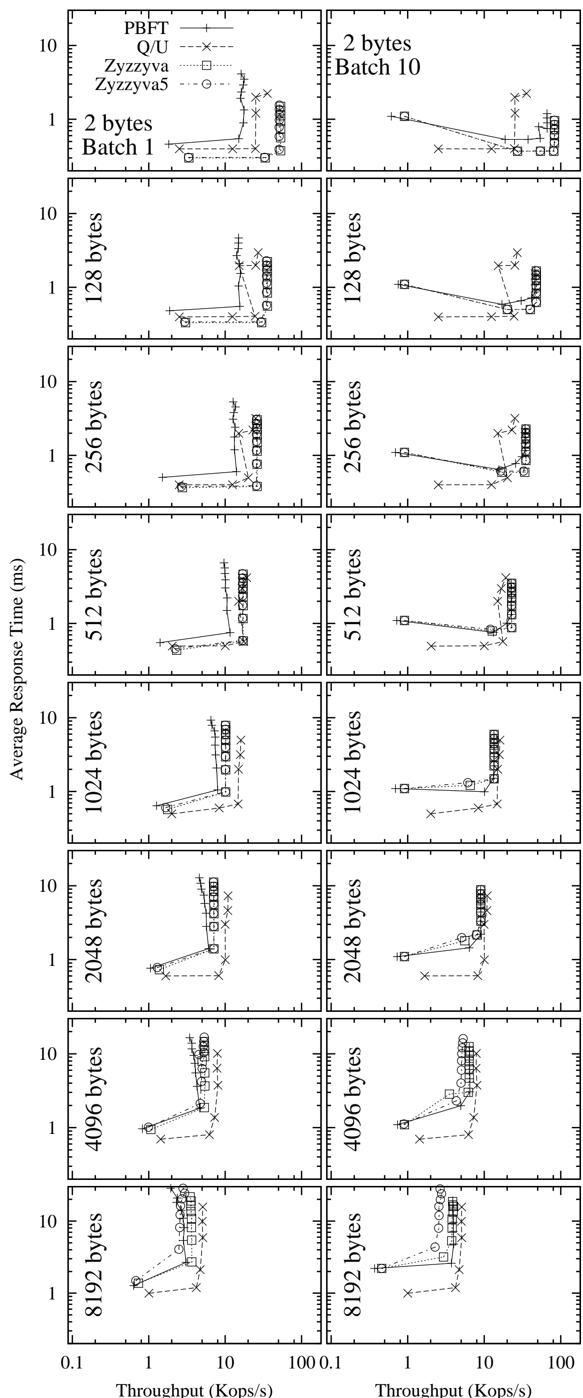

Figure 8: Latency-throughput plots for increasing request payload sizes, without batching (left) and with batch size 10 (right). Note that both axes are in logarithmic scale, to show better the relative performance differences of the protocols at different scales.

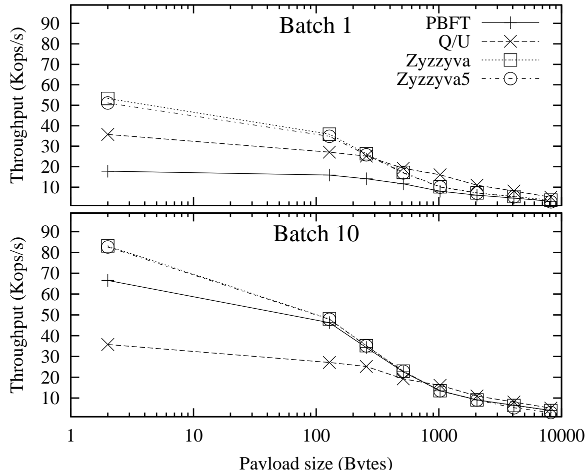

Figure 9: Peak throughput vs. request size. Top: no batching, bottom: batch size of 10. Q/U has no batching but appears in both plots for comparison. Note that the x axis is logarithmic.

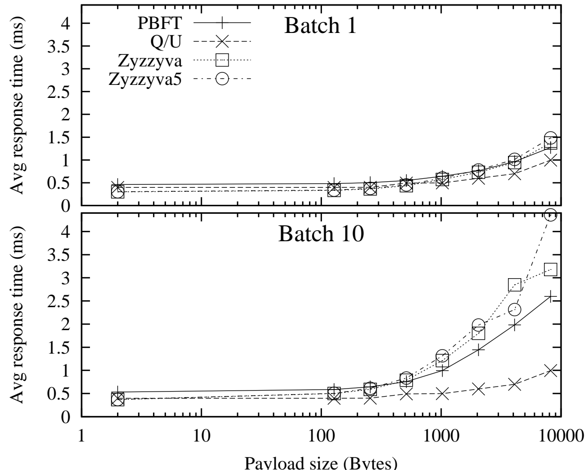

Figure 10: Minimum of per-experiment average request response time vs. request size. Top: no batching, bottom: batch size of 10. The system is *not* saturated (i.e., these are not the corresponding response times for Figure 9). Q/U has no batching but appears in both plots for comparison. Note that the x axis is logarithmic.

payload sizes as throughputs degrade dramatically. The reason is that with increasing request size, the per-byte processing costs and network transmission delays start to dominate per-request costs, which increasingly masks the differences among the protocols.

Though helpful in identifying the latency-throughput trade-offs, the plots in Figure 8 make it difficult to see the trends in peak throughput or latency below saturation. We show in Figure 9 the peak throughput as a function of the request size, and in Figure 10 the mean response time below protocol saturation. Peak throughput trends are clearer here; they are consistent with the large increase in transmission and digest computation costs resulting from larger request sizes.

Interestingly, Q/U appears more robust to increasing payload sizes than the other protocols, and exhibits the least steep decline in throughput. At high payload sizes, Q/U matches the throughput of Zyzzyva and Zyzzyva5, even when these protocols use batching. The reason is that Q/U's messages, which have a larger base size due to the history they contain, are increasingly dominated by the payload as the request size increases. Moreover, at large request sizes, the throughput is increasingly limited by per-byte processing costs like MAC computations and network bandwidth, making batching irrelevant.

Figure 10 shows the average response time for increasing request sizes below system saturation. PBFT with batching and Q/U provide the lowest response times. As payload sizes increase, response times with Zyzzyva and Zyzzyva5 suffer, because Zyzzyva and Zyzzyva5 send a full copy of all requests in a batched ORDERREQ message from the primary to the backups. In contrast, PBFT only sends message digests in its batched PREPREPARE message, leaving it to the client to transmit the request itself to the backups. The cost of transmitting and authenticating the request content is typically spread over all clients in PBFT, instead of concentrated at the primary in Zyzzyva[7].

There is no inherent reason why Zyzzyva might not benefit from an optimization similar to that of PBFT under the given network conditions; however, this optimization would increase the bandwidth consumption of the protocol (since all clients will have to send requests to all replicas, as with PBFT) and possibly reduce its ability to deal gracefully with clients who only partially transmit requests to replicas. The best approach may depend on the type of network anticipated. For instance, in networks where multicast is available (e.g., enterprise settings where local multicast deployment tends to be more common), the network component of the overhead caused by large payloads—but not the computation component—may be low.

4.3 Heterogeneous Network Conditions

Next, we place one replica behind a slow link, in order to introduce an imbalance among the replicas. PBFT, Q/U and Zyzzyva5 do not require all replicas to be in sync to deliver peak performance, therefore we expect their performance to be unaffected by heterogeneity. In contrast, we expected Zyzzyva's performance to degrade, because the protocol requires a timely response from all replicas to be able to complete a request in a single phase.

Figure 11 shows results for throughput and aver-

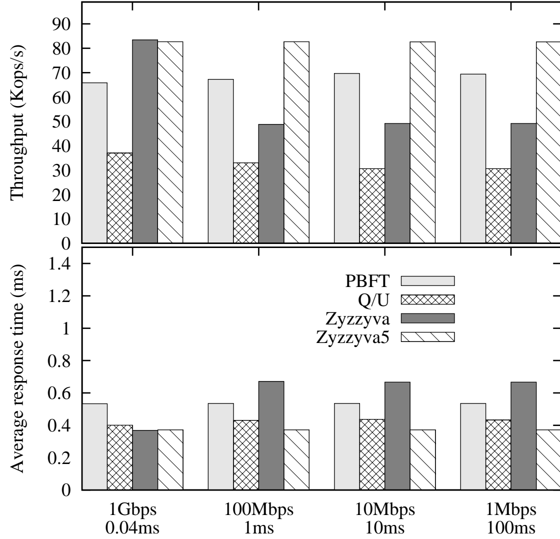

Figure 11: Impact of a replica behind a low bandwidth and high delay link. We vary the characteristics of the slow link, as shown on the x-axis. We set batchsize to 10.

age response time. All but the constrained link in this experiment have 1Gbps bandwidth and 0.04ms latency, whereas the constrained link has the characteristics shown on the x axis. The leftmost configuration is homogenous and included for comparison.

The throughput and response time of Zyzzyva is strongly affected by the presence of a replica with a constrained network link. Zyzzyva must adapt its timers[8] and eventually switch to two-phase operation as the response time of a slow replica worsens. The throughput of Zyzzyva drops as soon as the imbalance is introduced, because clients start the second phase of the protocol to complete their requests, instead of waiting for the slowest replica to complete the first phase. Consequently, the average response time for Zyzzyva also increases when there is an imbalance. While similar to the mute replica results in Section 3.3.4, our results show that Zyzzyva's throughput is sensitive even to small differences in network latency between a client and the replicas.

Q/U's throughput decreases slightly once there is a slower replica. The reason is that when clients initially include the slow replica in their preferred quorum, they may time out and subsequently choose a quorum that excludes the slow replica. As a result, the remaining replicas experience more load and throughput decreases.

The throughput of PBFT improves slightly with increased heterogeneity, because replicas receive fewer messages from the slow replica, which saves them message receive and MAC verification costs. Note that during sustained operation at peak throughput, the messages from the slow replica are queued at the routers adjacent to the slow link and eventually dropped. Of course, this loss of one replica's messages does not affect the protocol's operation and it actually increases protocol efficiency slightly.

4.4 Wide-area Network Conditions

Next, we explore the performance of BFT protocols under wide-area network (WAN) conditions, such as increased delay, lower bandwidth and packet loss. Such conditions are likely to arise in deployments where clients connect to the replicas remotely or when the replicas are geographically distributed.

Before presenting the results, we briefly review how the different protocols deal with packet loss.

4.4.1 Background

PBFT In PBFT, if no response to a request arrives, the client eventually times out and retransmits the request to the replicas. A backup replica, upon receiving such a retransmission, forwards it to the primary, assuming it must have missed it. If the primary has already sent a PREPREPARE for that retransmitted message, it does not do anything. However, this could trigger view changes if the original message was lost. To address this problem, PBFT replicas periodically (every 150 ms) multicast small status messages that summarize its state. When another replica notices that the reporting replica is missing messages that were sent in the past, it re-sends these messages [5].

Zyzzyva/Zyzzyva5 In Zyzzyva, if a backup replica receives an ORDERREQ message for seq i from the primary while it expects $k < i$ (suggesting a hole in the history), it sends a FILLHOLE message to the primary. The primary retransmits the missing messages. Loss of one such ORDERREQ message may cause a backup replica to send a FILLHOLE message to the primary for every future ORDERREQ until the hole is patched. This may cause the primary to experience additional load [14].

Q/U In Q/U, replicas inspect the histories contained in request messages to see if they have missed a prior request. If so, they request information from other replicas about the latest state of the corresponding objects. Otherwise, the protocol relies on request retransmission by clients to recover from packet losses.

4.4.2 Results

For the experiment, we configured replicas and clients with different link delay, bandwidth, and uniform random packet loss. Recall that clients and replicas are connected to a common hub via bi-directional links in a star topology. We simulate three wide area configurations as described in Table 2. 50 clients send requests containing 2-byte no-op requests.

Table 2: Wide-area experiment configurations, as shown in Figure 12.

Configuration	Client links (c)	Replica links (r)
Left	30ms, 10Mbps	1ms, 1Gbps
Middle	30ms, 10Mbps	5ms, 100Mbps
Right	30ms, 10Mbps	10ms, 100Mbps

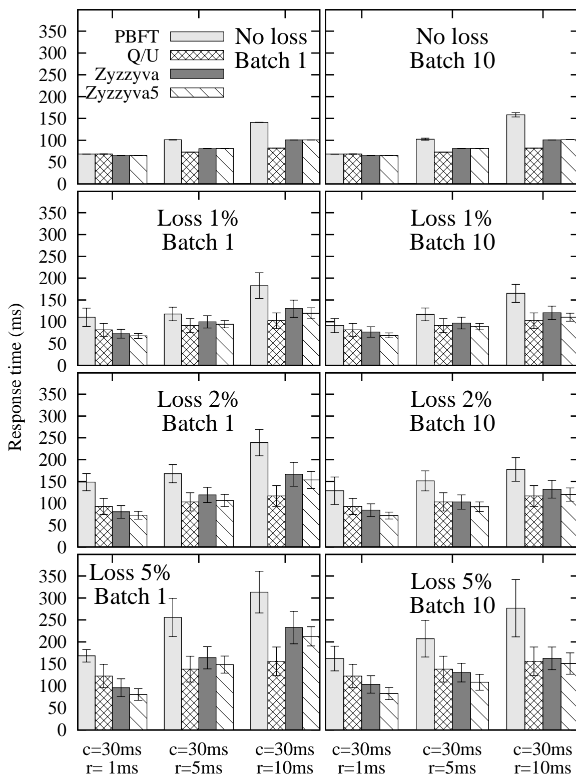

Figure 12: Three configurations on the x axis, average response latency in ms on the y axis. The error bars indicate one standard deviation around the average.

The client retransmission timeout was set based on the expected response time of each protocol. Because the request messages are very small and incur no execution overhead, we approximated the expected response time by adding the link delays on the protocol execution path. Let c be the client link latency and r the replica link latency (both one-way). In the absence of losses and faulty replicas, we expect PBFT to complete a request in time $(2c + 8r)$, Q/U in $(2c + 2r)$, and Zyzzyva/Zyzzyva5 in $(2c+4r)$. Q/U offers the lowest expected delay followed by Zyzzyva and Zyzzyva5 and then PBFT. We set the client retransmission timeout to this estimate for each protocol, plus 10ms. Figure 12 shows the average response time for all four protocols with and without batching, varying link latencies and loss rate; packet loss is uniform random and affects all links equally.

We make three observations. First, even though Q/U completes requests in a single phase with the contention-free workload used in this experiment, its response times are not better than those of the hybrid protocols when the replica-to-replica latencies are low (i.e., r=1 ms). Once again, the reason is the larger request messages required by Q/U. The extra latencies for transmitting and computing digests for the larger Q/U messages compensate for the extra latencies required for inter-replica communication in the agreement protocols. However, with increasing replica-to-replica latencies, the inter-replica communication significantly adds to the latency of the agreement protocols. Q/U's latencies, on the other hand, increase to a lesser extent because it relies on inter-replica communication only to fetch the current object state after a replica misses a request.

Second, Zyzzyva5 has slightly lower response times than Zyzzyva under message loss. The reason is that Zyzzyva5 requires only $4f + 1$ responses from its $5f + 1$ replicas, while Zyzzyva requires responses from all of its $3f + 1$ replicas to complete a request in a single phase.

Third, batching tends to improve the average latency under losses. Because batching reduces the number of messages per request, it reduces the probability that a given request is affected by a loss, thus reducing the average latency.

4.5 Clients with Misconfigured Timers

The goal of the next experiment is to understand how sensitive existing protocols are to faulty clients. Under contention, it is well known that the performance of quorum protocols like Q/U suffers. In fact, Q/U may lose liveness in the presence of faulty clients that do not back off sufficiently during the conflict resolution phase. The performance of agreement-based protocols like PBFT and Zyzzyva, is believed to be robust to faulty client behavior. The goal of our experiment is to test the validity of this hypothesis. Note that all BFT protocols preserve the safety property regardless of the behavior of clients.

We misconfigured some of the clients' retransmission timers to expire prematurely. The experiments used a LAN configuration, with round-trip delay of 0.16ms on all links (0.04ms one-way for each node-to-hub link). With 100 clients, the average response times are 3ms for Q/U, 1.5ms for PBFT with a batch size of 10, and 1.15ms for Zyzzyva and Zyzzyva5 with a batch size of 10.

We chose four settings of misconfigured timers—0.5ms, 1ms, 1.5ms, and 2ms. This choice of timers allows us to observe how the protocols behave under aggressive retransmissions by a fraction of the clients (between 0 to 25 out of a total of 100 clients.) A client with a misconfigured timer retransmits the request every time the timer expires without backing off, until a matching response arrives. Figure 13 shows the throughput of the protocols as a function of the number and settings of mis-

configured clients, with and without batching. Misconfigured timers affect the throughput of the protocols due to the extra computations of digests, MACs and transmission costs incurred whenever a protocol message must be retransmitted[9].

Our results show that all protocols are sensitive to premature retransmissions of request messages. This is expected, because premature retransmissions add to the total overhead per completed request.

PBFT and Zyzzyva replicas assume a packet loss when they receive a retransmission: PBFT backups forward the message to the primary when they receive a request for the second time, while Zyzzyva/Zyzzyva5 backups forward the request to the primary immediately because a client is expected to send a request to backups only upon a timeout. A Zyzzyva primary, upon receiving such a forwarded request message from a backup replica, responds with the OrderReq message. A PBFT primary responds with a PrePrepare to the backup only if it has never seen the request, otherwise it ignores the retransmission. By retransmitting a request to all backups, a misconfigured client causes the primary of both PBFT and Zyzzyva/Zyzzyva5 to receive additional messages from the backups.

A Q/U replica, upon receiving a retransmission, responds with the cached response if the request has executed, or else processes the request again. No additional replica-to-replica communication is necessary under a contention-free workload.

The relative impact of clients with misconfigured timers is more pronounced with batching. With batching, each individual retransmitted request can cause the retransmission of protocol messages for the entire batch that contained the original request. As a result, the aggregation of protocol messages that occurs in normal protocol operation does not occur for retransmitted requests.

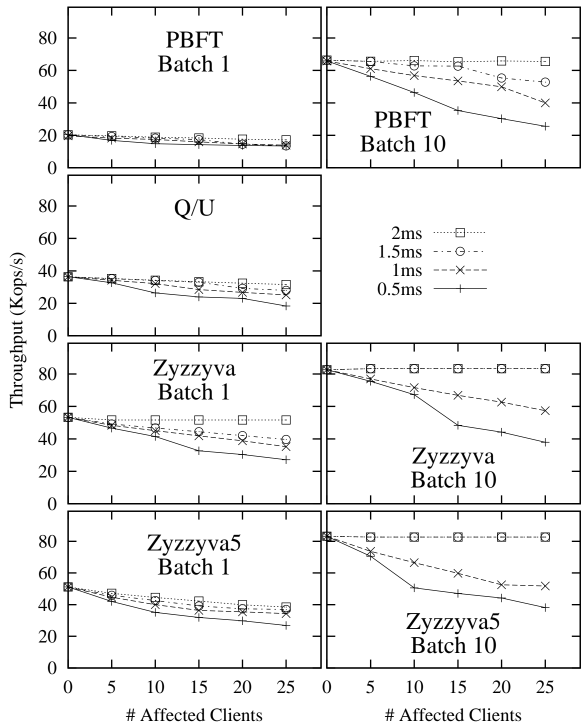

Figure 13: Clients with misconfigured retransmission timers. We vary the number of such clients on the x-axis (100 clients total).

5 Related Work

While there has been considerable work on simulators for networks and P2P systems, we are aware of relatively little work that attempts to model both CPU performance and network characteristics, though we note that a similar technique was used by Castro [5] to build an analytical model of the PBFT protocol.

Systems like WiDS [17], Macedon [21], and its successor MACE [12] allow distributed systems to be written in a state-machine language, which can then be used to generate both a native code implementation and drive a simulator, which also executes "real" code.

`BFTSim` takes a different approach, simulating both the message exchange and the CPU-intensive operations. This allows easy exploration of the effect of CPU performance for crypto operations, and P2's declarative specifications are an order of magnitude more concise than MACE (which itself is considerably more concise than a manual implementation in Java or C++). The disadvantage with `BFTSim` is that BFT protocols thus specified cannot be executed "for real" in a production system without extensions to P2, and would likely result in a less efficient implementation due to P2 currently generating software dataflow rather than native code.

We chose our BFT protocols (PBFT, Q/U, and Zyzzyva) to provide good coverage for `BFTSim` in order to evaluate its effectiveness as well as provide interesting comparisons. However, recent research has produced a slew of new protocols, which we intend to examine.

To take one example, Cowling et al.'s HQ protocol [9] ingeniously combines quorum and consensus approaches. HQ is a two-round quorum protocol in the absence of write conflicts and requires $3f + 1$ replicas. Replicas optimistically choose an ordering of requests (a *grant*) and notify the client, which collects a quorum of $2f + 1$ such grants and in a second round returns the collected grants (a *writeback*). Replicas detect contention by observing the set of grants in the writeback, and resort to Byzantine consensus for resolution rather than exponentially backing off as in a pure quorum system. As a result, HQ improves upon PBFT in low-concurrency settings, while resolving concurrency conflicts at a lower

expected latency than Q/U. Published results for HQ so far are only for fault-free settings with low-latency, high-bandwidth links.

Part of our agenda in this paper is to argue for the comparison of distributed algorithms on a level playing field in a variety of realistic, but different, scenarios. Only then can developers select appropriate algorithms for the particular tradeoffs at hand. Such a shift in thinking has been recently recognized in processor architecture, where it is termed "scenario-oriented design" [20].

6 Future Work and Conclusions

In this paper we recognize that, though bold, new moves have been made towards designing and implementing efficient, safe, and live Byzantine-fault tolerant replicated state machines, little has been done to look under the covers of those protocols and to evaluate them under realistic imperfect operating conditions. We argue that a simulation framework in which fundamentally different protocols can be distilled, implemented, and subjected to scrutiny over a variety of workloads, network conditions, and transient benign faults, can lead to the deeper understanding of those protocols, and to their broad deployment in mission-critical applications.

Our first contribution has been `BFTSim`, a simulation environment that couples a declarative networking platform for expressing complex protocols at a high level (based on P2), and a detailed network simulator (based on `ns-2`) that has been shown to capture most of the intricacies of complex network conditions. Using `BFTSim`, we have validated and reproduced published results about existing protocols, as well as the behavior of their actual implementations on real environments. We feel confident that this will encourage the systems community to look closely at published protocols and understand (and reproduce!) their inherent performance characteristics, with or without the authors' implementation, without unreasonable effort.

Second, we have taken some first steps towards this goal with three protocols, PBFT, Q/U, and Zyzzyva/Zyzzyva5. We have identified some interesting patterns in how these protocols operate:

- One-size-fits-all protocols may be tough if not impossible to build; different performance trade-offs lead to different design choices within given network conditions. For instance, PBFT offers more predictable performance and scales better with payload size compared to Zyzzyva; in contrast, Zyzzyva offers greater absolute throughput and is significantly more robust in wider-area, lossy networks.
- In the contention-free workloads we study, Q/U can demonstrate its strengths in particular as payload sizes grow and replica-replica latencies increase, compared to all competing protocols. This opens up an intriguing question: what if Q/U were less vulnerable to write contention? An overly simple assessment might argue that Zyzzyva is roughly equivalent to Q/U with an explicit preserializer of requests, which ensures that no write contention occurs in the absence of Byzantine faults [22]. It may be productive to assess to what extent this similarity is only superficial and, if not, what benefits one might gain from building a protocol from scratch, versus engineering a safe composition of existing protocols. Alternatively, a new protocol that shares Q/U's optimistic one-phase execution with HQ's efficient contention resolution may become appealing, especially for large-request workloads.
- Timeouts should be set very carefully. This should come as no surprise. However, some protocols are more vulnerable to timeout misconfigurations than others. With Zyzzyva, the ability to complete a request with a single phase offers spectacular opportunities for high throughput, but misconfiguration of the timeout, or the inevitable jitter in wide-area deployments, can rob the protocol of its benefits. In contrast, Q/U tolerates wider timer misconfigurations, but has less to lose in absolute terms.

Although we are confident that our approach is promising, the results described in this paper only scratch the surface of the problem. We have not yet assessed the particular benefits of reliable transports, especially in the presence of link losses; we have only studied a simplified notion of multicast and have yet to study data-link broadcast mechanisms used by some protocols; we have only explored relatively simplistic, geographically constrained topologies instead of more complex, wide-area topologies involving faraway transcontinental or transoceanic links; and we have limited ourselves to the relatively closed world of Byzantine-fault tolerant replicated state machines. Removing these limitations from `BFTSim` is the subject of our on-going research.

Moving forward, we are expanding the scope of our study under broader workload conditions (varying request execution costs, cryptographic costs, heterogeneous computational capacities), network conditions (skewed link distributions, jitter, asymmetric connectivity), and faults (flaky clients, denial of service attacks, timing attacks). We particularly hope to extract the salient features of different protocols (such as PBFT's and Zyzzyva's ways of dealing with client requests, or Zyzzyva's and Q/U's similarities modulo request preserialization), as well as expand to incorporate storage costs, and bandwidth measurements. Finally, we hope to stimulate further research and educational use by making `BFTSim` publicly available, along with our implementations of BFT protocols.

Acknowledgments

We are particularly indebted to Michael Abd-El-Malek, Miguel Castro, Allen Clement, and Ramakrishna Kotla for their help in sorting through protocol and implementation details needed by our work. We are grateful to Byung-Gon Chun and Rodrigo Rodrigues for their comments on earlier drafts of this paper. Finally, we heartily thank our shepherd, Miguel Castro, and the anonymous reviewers for their deep and constructive feedback.

References

[1] The NS-2 Project. `http://nsnam.isi.edu/nsnam/index.php/Main_Page`, Oct. 2007.

[2] M. Abd-El-Malek. Personal communication, 2007.

[3] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR Fault Tolerance for Cooperative Services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[5] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT Laboratory for Computer Science, Jan. 2001. Technical Report MIT/LCS/TR-817.

[6] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 1999.

[7] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4), 2002.

[8] B.-G. Chun, S. Ratnasamy, and E. Kohler. A Complexity Metric for Networked System Designs. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, 2008.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2006.

[10] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2007.

[11] F. Junqueira, Y. Mao, and K. Marzullo. Classic Paxos vs. Fast Paxos: Caveat Emptor. In *Proceedings of USENIX Hot Topics in System Dependability (HotDep)*, 2007.

[12] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: Language Support for Building Distributed Systems. In *Proceedings of ACM Programming Languages Design and Implementation (PLDI)*, 2007.

[13] R. Kotla. Personal communication, 2007.

[14] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2007.

[15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. *University of Texas at Austin, Technical Report: UTCS-TR-07-40*, 2007.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[17] S. Lin, A. Pan, Z. Zhang, R. Guo, and Z. Guo. WiDS: an Integrated Toolkit for Distributed System Development. In *Proceedings of USENIX Hot Topics in Operating Systems (HotOS)*, 2005.

[18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, 2005.

[19] B. M. Oki and B. H. Liskov. Viewstamped Replication: a General Primary Copy. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[20] S. M. Pieper, J. M. Paul, and M. J. Schulte. A New Era of Performance Evaluation. *IEEE Computer*, 40(9), 2007.

[21] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, 2004.

[22] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. Conflict-free Quorum based BFT Protocols. Technical Report 2007-2, Max Planck Institute for Software Systems, 2007.

[23] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, , and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2002.

[24] X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *Advances in Cryptology—EUROCRYPT 05*, 2005.

Notes

[1] We use the terms "latency" and "response time" interchangeably when referring to protocol performance.

[2] The Paxos protocol [16]—also concurrently discovered as Viewstamped Replication [19]—forms the basis of most fault-tolerant consensus mechanisms, in crash-fault or Byzantine-fault settings.

[3] Support for jumbo UDP frames was incomplete in the released version of `ns-2`. We had to enable jumbo-frame handling and IP fragmentation for the large-sized UDP messages of our protocols. However, we have induced no extra delays due to IP fragmentation or reordering. Furthermore, `BFTSim` does not currently simulate network congestion.

[4] There is a minor error in the Zyzzyva publication, implying that Zyzzyva uses AdHash for message digests and MD5 for MACs. We have since confirmed that, as with the PBFT codebase on which Zyzzyva is built, MD5 is used for digests (with AdHash-MD5 only for incremental state digests) and UMAC for MACs. Note that MD5, either in one-shot mode or incremental AdHash form, is no longer considered collision-resistant [24]; we use it here for validation purposes only.

[5] We were at first unable to reproduce reasonable performance results with the Q/U implementation, a similar experience to other research groups [2]. We fixed a bug with DNS resolution in the Q/U codebase (acknowledged and incorporated in release 1.2 of Q/U) that removed the problem.

[6] History is an ordered set of *candidates*, where each candidate is a pair of logical timestamps. A logical timestamp is represented as $\langle$TIME, BARRIERFLAG, CLIENTID, OPERATION, OHS$\rangle$, where OHS is the object history set.

[7] Note that PBFT offers a runtime parameter for including entire requests within batches; the default configuration of the codebase turns this option off.

[8] Note that we implement an adaptive timer mechanism for clients [13] used in, but not described in, the Zyzzyva publication. Once a client receives $2f + 1$ matching responses it starts an adaptive timer, initialized to a low value, and starts the second phase if this timer expires before receiving the full $3f + 1$ responses. If a client receives $3f + 1$ responses before completing the second phase, it increases the adaptive timer to avoid starting the second phase too early next time. If the second phase completes sooner, the timer is reset to the initial low value.

[9] The original PBFT implementation appears to implement an optimization that caches the digests of transmitted messages; therefore, digests do not have to be recomputed when retransmitting a message. We have not yet implemented this optimization in our version of PBFT.