ZZ and the Art of Practical BFT

Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet
Department of Computer Science, University of Massachusetts Amherst
Email: {twood,rahul,arun,shenoy,cecchet}@cs.umass.edu

Abstract

The high replication cost of Byzantine fault-tolerance (BFT) methods has been a major barrier to their widespread adoption in commercial distributed applications. We present ZZ, a new approach that reduces the replication cost of BFT services from $2f+1$ to practically $f+1$. The key insight in ZZ is to use $f+1$ execution replicas in the normal case and to activate additional replicas only upon failures. In shared hosting data centers where multiple applications share a physical server, ZZ reduces the aggregate number of execution replicas running in the data center, thereby improving throughput and response times. ZZ relies on virtualization—a technology already employed in modern data centers—for fast replica activation upon failures, and enables newly activated replicas to immediately begin processing requests by fetching state on-demand. A prototype implementation of ZZ using the BASE library and Xen shows that, when compared to a system with $2f+1$ replicas, our approach yields lower response times and up to 33% higher throughput in a prototype data center with four BFT web applications. We also show that ZZ can handle simultaneous failures and achieve sub-second recovery.

1 Introduction

Today's enterprises rely on data centers—server and storage farms—to run their critical business applications. As users have become increasingly dependent on online services, malfunctions have become highly problematic, resulting in financial losses, negative publicity, or frustrated users. Consequently, maintaining high availability of critical services is a pressing need as well as a challenge.

Byzantine fault tolerance (BFT) is a powerful replication approach for constructing highly-available services that can tolerate arbitrary (Byzantine) faults. This approach requires replicas to agree upon the order of incoming requests and process every request in the agreed upon order. Despite numerous efforts to improve the performance or fault scalability of BFT systems [3, 7, 14, 22, 26, 1], existing approaches remain expensive, requiring at least $2f+1$ replicas to execute each request in order to tolerate f faults [14, 28]. This high replication cost has been a significant barrier to their adoption—to the best of our knowledge, no commercial data center application uses BFT techniques today, despite the wealth of research in this area.

Many recent efforts have focused on optimizing the agreement protocol used by BFT replicas [7, 14]; consequently, today's state-of-the-art protocols can scale to a throughput of 80,000 requests/s and incur overheads of less than 10 μs per request for reaching *agreement* [14]. In contrast, request *execution* overheads for typical applications such as web servers and databases [26] can be in the order of milliseconds or tens of milliseconds—three orders of magnitude higher than the agreement cost. Since request executions dominate the total cost of processing requests in BFT services, the hardware (server) capacity needed for request executions will far exceed that for running the agreement protocol. Hence, we argue that the total cost of a BFT service can be truly reduced only when the total overhead of request executions, rather than the cost to reach agreement, is somehow reduced.

In this paper, we present ZZ, a new approach that reduces the cost of replication as well as that of request executions in BFT systems. Our approach enables general BFT services to be constructed with a replication cost close to $f+1$, halving the $2f+1$ or higher cost incurred by state-of-the-art approaches [28]. ZZ targets shared hosting data center environments where replicas from multiple applications can share a physical server. The key insight in ZZ[1] is to run only $f+1$ execution replicas per application in the graceful case where there are no faults, and to use additional sleeping replicas that get activated only upon failures. By multiplexing fewer replicas onto a given set of shared servers, our approach is able to provide more server capacity to each replica, and thereby achieve higher throughput and lower response times for request executions. In the worst case where all applications experience simultaneous faults, our approach requires an additional f replicas per application, matching the overhead of the $2f+1$ approach. However, in the common case where only a *subset* of the data center applications are experiencing faults, our ap-

[1] Denotes sleeping replicas; from the sleeping connotation of the term "zz.."

proach requires fewer replicas in total, yielding response time and throughput benefits. Like [28], our system still requires $3f + 1$ agreement replicas; however, we have argued the overhead imposed by agreement replicas is small, allowing such replicas from multiple applications to be densely packed onto physical servers.

The ability to quickly activate additional replicas upon fault detection is central to our ZZ approach. While any mechanism that enables fast replica activation can be employed in ZZ, in this paper, we rely upon virtualization—a technique already employed in modern data centers—for on-demand replica activation.

This paper makes the following contributions. First, we propose a practical solution to reduce the cost of BFT to $f+1$ execution replicas. ZZ leverages virtualization for fast replica activation and optimizes the recovery protocol to allow additional newly-activated replicas to immediately begin processing requests through an amortized state transfer strategy. Second, we implement a prototype of ZZ by enhancing the BASE library and combining it with the Xen virtual machine and the ZFS file system. We evaluate our prototype using a BFT web server and a ZZ-based NFS file server. Our experimental results demonstrate the following benefits. (1) In a prototype data center running four BFT web servers, ZZ's use of only $f+1$ execution replicas in the fault-free case yields response time and throughput improvements of up to 66% and 33%, when compared to systems using $3f + 1$ (BASE) and $2f + 1$ replicas, respectively. (2) In the presence of failures, after a short recovery period, ZZ performs as well or better than $2f + 1$ replication and still outperforms BASE's $3f+1$ replication. (3) The use of paused virtual machine replicas and on-demand state fetching allows ZZ to achieve sub-second recovery times. (4) Techniques such as batching proposed to optimize the agreement protocol have no significant benefits for BFT applications with non-trivial request execution costs, regardless of whether ZZ or another system is used.

The rest of this paper is structured as follows. Section 2 gives a background on BFT and the design of ZZ is described in Sections 3 and 4. Sections 5 and 6 describes the ZZ implementation and evaluation. Related work is presented in Section 7 and Section 8 concludes the paper.

2 State-of-the-art vs. the Art of ZZ

In this section, we give an overview of the benefits of ZZ in comparison to state-of-the-art approaches. As separation of agreement and execution [28] is critical to ZZ's design, we first explain how this idea reduces the execution replication cost of BFT from $3f + 1$ to $2f + 1$.

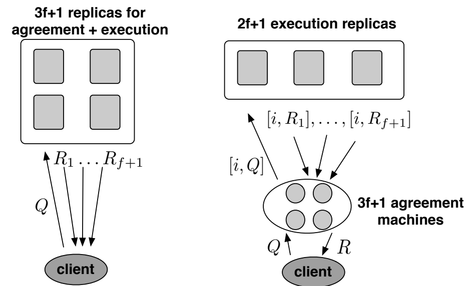


Figure 1: The PBFT approach versus the separation of agreement from execution.

2.1 From 3f+1 to 2f+1

In the traditional PBFT approach [3], during graceful execution, a client sends a request Q to the replicas. The $3f + 1$ (or more) replicas agree upon the sequence number corresponding to Q, execute it in that order, and send responses back to the client. When the client receives $f + 1$ valid and matching responses $R_1, \ldots, R_{f+1}$ from different replicas, it knows that at least one correct replica executed Q in the correct order. Figure 1 illustrates how the principle of separating agreement from execution can reduce the number of execution replicas required to tolerate up to f faults from $3f + 1$ to $2f + 1$. In this separation approach [28], the client sends Q to a primary in the agreement cluster consisting of $3f + 1$ lightweight machines that agree upon the sequence number i corresponding to Q and send $[Q, i]$ to the execution cluster consisting of $2f + 1$ replicas that store and process application state. When the agreement cluster receives $f + 1$ matching responses from the execution cluster, it forwards the response to the client knowing that at least one correct execution replica executed Q in the correct order. For simplicity of exposition, we have omitted cryptographic operations above.

2.2 Circumventing 2f+1

The 2f+1 replication cost is believed necessary [14, 7, 1] for BFT systems. For example, Zyzzyva (refer Table 1 in [14]) claims that $2f + 1$ is a lower bound on the number of replicas with application state for state machine replication (SMR). However, more than a decade ago, Castro and Liskov concluded their original paper on PBFT [3] saying "it is possible to reduce the number of copies of the state to $f + 1$ but the details remain to be worked out". In this paper, we work out those details.

Table 1 shows the replication cost and performance characteristics of existing BFT approaches, including non-SMR approaches based on quorums, in comparison to ZZ. All listed numbers are for gracious execution, i.e., when there are no faults and the network is well-behaved. Note that all approaches require at least $3f+1$ replicas in order to tolerate up to f independent Byzantine failures,

	PBFT'99 [3]	SEP'03 [28]	Q/U'05 [1]	HQ'06 [7]	Zyzzyva'07 [14]	ZZ
Agreement replicas	$3f+1$	$3f+1$	N/A	$3f+1$	$3f+1$	$3f+1$
Execution replicas	$3f+1$	$2f+1$	$5f+1$	$3f+1$	$2f+1$	$(1+r)f+1$
Total 1-way network delays	4	4	2	4	3	4
WAN round-trip delays (W)	1	1	1	2	1	1
Bottleneck agreement MACs per request (μ)	$2+\frac{8f+1}{b}$	$2+\frac{12f+3}{b}$	$2+8f$	$4+4f$	$2+\frac{3f}{b}$	$2+\frac{10f+3}{b}$
Peak throughput $\leq$ min of	$(\frac{1}{(3f+1)E},\frac{1}{2\mu})$	$(\frac{1}{(2f+1)E},\frac{1}{2\mu})$	$(\frac{1}{(3f+1)E},\frac{1}{(2+8f)\mu})$	$(\frac{1}{(2f+1)E},\frac{1}{(4+4f)\mu})$	$(\frac{1}{(2f+1)E},\frac{1}{2\mu})$	$(\frac{1}{(f+1)E},\frac{1}{2\mu})$
Client-perceived latency $\geq$	W + E	W + E	W + E	2W + E	W + E	W + E

Table 1: ZZ versus existing BFT approaches. Here, f is the number of allowed faults, $r \ll 1$ is a configurable variance parameter formally defined in §4.7.3, and b is the batch size. The quantities W, μ, and E all have the units of seconds. All numbers are for periods when there are no faults and the network is well-behaved.

consistent with classical results that place a lower bound of $3f+1$ replicas for a safe Byzantine consensus protocol that is live under weak synchrony assumptions [9].

In contrast to common practice, we do not measure replication cost in terms of the total number of physical machines as we assume a virtualized environment that is common in many data centers today. Virtualization allows resources to be allocated to a replica at a granularity finer than an entire physical machine. Virtualization itself is useful in multiplexed environments, where a data center owner hosts many services simultaneously for better management of limited available resources. Note that virtualization helps all BFT approaches, not just ZZ, in multiplexed environments. To appreciate this, consider a data center owner with S physical servers seeking to host N services. Without virtualization, a BFT approach can support at most $N = S/(3f+1)$ services, even though each replica may be using only a small portion of a physical server's resources. However, if each physical server can support K virtual service replicas, then the data center can support up to $N = KS/(3f+1)$ services. Conversely, for a given N, the provisioning cost incurred by a data center is at least $S = N(3f+1)$ without virtualization, but just $S = N(3f+1)/K$ with virtualization.

Cost: Our position is that execution, not agreement, is the dominant provisioning cost for most realistic data center services that can benefit from the high assurance provided by BFT. To put this in perspective, consider that state-of-the-art BFT approaches such as Zyzzyva show a peak throughput of over 80K requests/second for a toy application consisting of *null* requests, which is almost three orders of magnitude more than the achievable throughput for a database service on comparable hardware [26]. ZZ nearly halves the provisioning cost for the data center by nearly halving the number of replicas actively executing requests (Table 1 row 2).

Latency: Wide-area network (WAN) delays (denoted by W) and execution latency (E) dominate client-perceived response times. Although ZZ like SEP, incurs two additional one-way LAN delays (row 3) on the critical path compared to the optimal (Q/U for a workload with low contention), all approaches except HQ incur just one WAN round-trip propagation delay. Accounting for WAN transmission delays for large requests or responses further elides the difference between different approaches with respect to client-perceived response times (row 7).

Throughput: ZZ can achieve a higher peak throughput compared to state-of-the-art approaches when execution dominates request processing cost (row 6). For a fair comparison, assume that all approaches are provisioned with the same total amount of resources, say unity. Then, the peak throughput of each approach is bounded by the minimum of its best-case execution throughput and its best-case agreement throughput. Agreement throughput is primarily limited by the overhead μ of a MAC operation and can be improved significantly through batching for all approaches except Q/U and HQ. However, batching is immaterial to the overall throughput when execution is the bottleneck. All SMR-based approaches except HQ must execute requests sequentially in order to preserve the semantics of a general SMR service.

Q/U relaxes SMR semantics to that of an object read-write system and can leverage the throughput benefits of concurrent execution at the expense of more $(5f+1)$ active replicas. HQ combines the benefits of quorum-based approaches with SMR-based approaches using a preferred quorum of size $2f+1$ and a total of $3f+1$ replicas. Note that $1/((2f+1)E)$ is an upper bound on HQ's execution throughput; the remaining $f+1$ active replicas must still participate in state transfer operations for each request. Although HQ does not explicitly separate agreement and execution in the sense of SEP, HQ

reverts to a PBFT-like protocol when a client can not gather sufficient responses from the preferred quorum, so it is listed as having $3f+1$ agreement replicas.

The comparison above is for performance of different protocols during periods when there are no faults and the network is well-behaved. With faults or long unpredictable network delays, the throughput and latency of all approaches can degrade significantly and a thorough comparison is nontrivial and difficult to characterize concisely [24, 6]. For example, a very recent paper [6] shows that faulty clients can abuse MACs and cause service unavailability in PBFT and Zyzzyva by forcing repeated view changes. However, this attack technically impacts the agreement protocol, not the execution protocol. ZZ's current implementation being based on PBFT is also vulnerable to such attacks, but can in principle be addressed using similar fixes to the agreement protocol [6].

When failures occur, ZZ incurs a higher latency to execute some requests until its failure recovery protocol is complete. Our experiments suggest that this additional overhead is modest and is small compared to typical WAN delays. In a world where failures are the uncommon case, ZZ offers valuable savings in replication cost or, equivalently, improvement in throughput under limited resources.

ZZ is not a new "BFT protocol" as that term is typically used to refer to the agreement protocol; instead, ZZ is an execution approach that can be interfaced with existing BFT-SMR agreement protocols. Our prototype uses the BASE implementation of the PBFT protocol as it was the most mature and readily available BFT implementation at the time of writing. The choice was also motivated by our premise that we do not seek to optimize agreement throughput, but to demonstrate the feasibility of the ZZ's execution approach with a reasonable agreement protocol. Admittedly, it was easier to work out the details of augmenting ZZ to PBFT compared to more sophisticated agreement protocols.

3 ZZ design

In this section, we present our system and fault model, followed by an overview of the ZZ design.

3.1 System and Fault Model

We assume a Byzantine failure model where faulty replicas or clients may behave arbitrarily. There are two kinds of replicas: 1) agreement replicas that assign an order to client requests and 2) execution replicas that maintain application state and execute client requests. Replicas fail independently, and we assume an upper bound g on the number of faulty agreement replicas and a bound f on the number of faulty execution replicas in a given window of vulnerability. We initially assume an infinite window of vulnerability, and relax this assumption in Section 4.7. An adversary may coordinate the actions of faulty nodes in an arbitrary malicious manner. However, the adversary can not subvert standard cryptographic assumptions about collision-resistant hashes, encryption, and digital signatures.

Our system uses the state machine replication (SMR) model to implement a BFT service. Replicas agree on an ordering of incoming requests and each execution replica executes all requests in the same order. Like all previous SMR based BFT systems, we assume that either the service is deterministic or the non-deterministic operations in the service can be transformed to deterministic ones via the agreement protocol [3, 14, 28, 22].

Our system ensures safety in an asynchronous network that can drop, delay, corrupt, or reorder messages. Liveness is guaranteed only during periods of synchrony when there is a finite but possibly unknown bound on message delivery time. The above system model and assumptions are similar to those assumed by many existing BFT systems [3, 14, 28, 22].

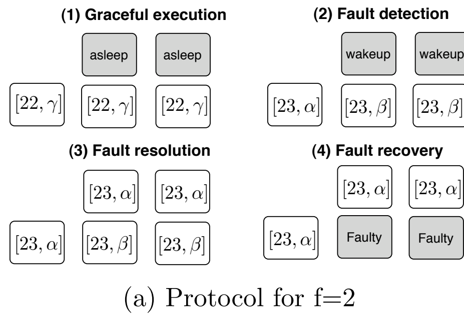
Virtualization: ZZ assumes that replicas are being run inside virtual machines. As a result, it is possible to run multiple replicas on a single physical server. To maintain the fault independence requirement, no more than one agreement replica and one execution replica of each service can be hosted on a single physical server.

ZZ assumes that the hypervisor may be Byzantine. Because of the placement assumption above, a malicious hypervisor is equivalent to a single fault in each service hosted on the physical machine. As before, we assume a bound f on the number of faulty hypervisors within a window of vulnerability. We note that even today sufficient hypervisor diversity (e.g., Xen, KVM, VMWare, Hyper-V) is available to justify this assumption.

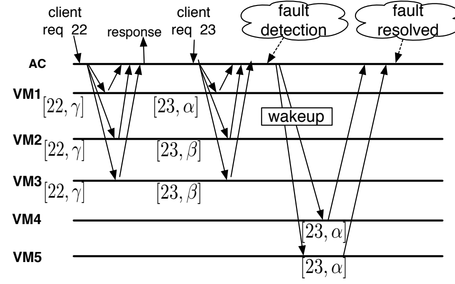
3.2 ZZ Design Overview

ZZ reduces the replication cost of BFT from $2f+1$ to nearly $f+1$ using virtualization based on two simple insights. First, if a system is designed to be correct in an asynchronous environment, it must be correct even if some or all replicas are arbitrarily slow. Second, during fault-free periods, a system designed to be correct despite f Byzantine faults must be unaffected if up to f replicas are turned off. ZZ leverages the second insight to turn off f replicas during fault-free periods requiring just $f+1$ replicas to actively execute requests. When faults occur, ZZ leverages the first insight and behaves exactly as if the f standby replicas were slow but correct replicas.

If the $f+1$ active execution replicas return matching responses for an ordered request, at least one of these responses, and by implication all of the responses, must



(a) Protocol for f=2



(b) Fault and Recovery Timeline

Figure 2: Various scenarios in the ZZ system for $f = 2$ faults. Request 22 results in matching responses γ, but the mismatch in request 23 initiates new virtual machine replicas on demand.

be correct. The problematic case is when the $f + 1$ responses do not match. In this case, ZZ starts up additional virtual machines hosting standby replicas For example, when $f = 1$, upon detecting a fault, ZZ starts up a third replica that executes the most recent request. Since at most one replica can be faulty, the third response must match one of the other two responses, and ZZ returns this matching response to the client. Figure 2 illustrates the high-level control flow for $f = 2$. Request 22 is executed successfully generating the response γ, but request 23 results in a mismatch waking up the two standby VM replicas. The fault is resolved by comparing the outputs of all $2f + 1$ replicas, revealing α as the correct response.

The above design would be impractical without a quick replica wake-up mechanism. Virtualization provides this mechanism by maintaining additional replicas in a "dormant" state. Additional replicas can be stored in memory as prespawned but paused VMs; unpausing a VM simply requires scheduling it on the CPU causing it to become active within milliseconds. Replicas can also be hibernated to disk if longer recovery times are acceptable and memory becomes a bottleneck resource across the data center. ZZ supports a mix, allowing for some (e.g., important) VMs to be paused in memory and the rest to hibernate on disk, as shown in Figure 3.

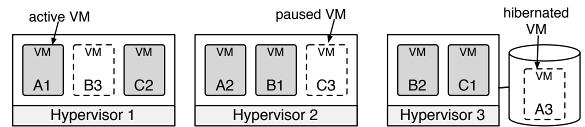


Figure 3: An example server setup with three fault tolerant applications, A, B, and C; only execution replicas are shown.

3.3 Design Challenges

The high-level approach described above raises several further challenges. First, how does a restored replica obtain the necessary application state required to execute the current request? In traditional BFT systems, each replica maintains an independent copy of the entire application state. Periodically, all replicas checkpoint their application state and discard the sequence of responses before a stable checkpoint. The checkpoint is also used to bring up to speed a slow replica that has missed several previous checkpoints by transferring the entire application state to the slow replica. However, a restored ZZ replica may not have any previous version of application state. It must be able to verify that the state is correct even though there may be only one correct execution replica (and f faulty ones), e.g., when $f = 1$, the third replica must be able to determine which of the two existing replicas possess the correct state.

Second, transferring the entire application state can take an unacceptably long time. In existing BFT systems, a recovering replica may generate incorrect messages until it obtains a stable checkpoint. This inconsistent behavior during checkpoint transfer is treated like a fault and does not impede progress of request execution if there is a quorum of $f + 1$ correct execution replicas with a current copy of the application state. However, when a ZZ replica recovers, there may exist just one correct execution replica with a current copy of the application state. The traditional state transfer approach can stall request execution in ZZ until f recovering replicas have obtained a stable checkpoint.

Third, ZZ's replication cost must be robust to faulty replica or client behavior. A faulty client must not be able to trigger recovery of standby replicas. A compromised replica must not be able to trigger additional recoveries if there are at least $f + 1$ correct and active replicas. If these conditions are not met, the replication cost savings would vanish and system performance can be worse than a traditional BFT system using $2f + 1$ replicas.

4 ZZ Protocol

This section presents the main components of the ZZ protocol for (i) graceful execution, (ii) checkpointing,

(iii) fault detection and replica recovery, and (iv) amortized state transfer, followed by a discussion of ZZ's safety and liveness properties. The notation $\langle \text{LABEL}, X \rangle_i$ denotes the message X of type LABEL digitally signed by node i. We indicate the digest of message X as $\overline{X}$.

4.1 Graceful Execution

4.1.1 Client Request

A client c sends a request $\langle \text{REQUEST}, o, t, c \rangle_c$ to the agreement cluster to submit an operation o with a timestamp t to the service. The timestamps ensure exactly-once semantics for execution of client requests as they enforce a total order across all requests issued by the client. A correct client uses monotonically increasing timestamps, e.g., the value of the client's local clock, and a faulty client's behavior does not affect other clients' requests. After issuing the request, the client waits for a response certified by at least $f + 1$ execution replicas.

4.1.2 Agreement

Upon receiving a client request $Q = \langle \text{REQUEST}, o, t, c \rangle_c$, the agreement replicas execute an agreement protocol and commit a sequence number to the request. Each agreement replica j sends a commit message $\langle \text{COMMIT}, v, n, j, Q \rangle_j$ that includes the view v and the sequence number n to all execution replicas. If the agreement replicas use the standard three-phase BFT agreement protocol [3], this commit message is the same as that sent to other agreement replicas in the third phase, so the transmission to the execution replicas is combined with the third phase of the agreement protocol.

4.1.3 Execution

An execution replica i executes a request Q when it gathers a commit certificate $\{\langle \text{COMMIT}, v, n, j, Q \rangle_j\}, j \in A|2g + 1$, i.e., a set of $2g + 1$ valid and matching commit messages from the agreement cluster, and it has executed all other requests with a lower sequence number. Let R denote the response obtained by executing Q. Replica i sends an execution report message $\langle \text{EXEC-REPORT}, n, i, \overline{R} \rangle_i$ containing the digest of the response to the agreement cluster. Replica i also sends a response message $\langle \text{REPLY}, v, t, c, i, R \rangle_i$ directly to the client.

In the normal case, the client receives a response certificate, i.e., a set of responses $\{\langle \text{REPLY}, v, t, c, i, R \rangle_i\}, i \in E|f+1$, from the execution cluster. Since at most f execution replicas can be faulty, a client receiving a response certificate knows that the response is correct.

4.1.4 Client Timeout

If a client does not receive a response certificate within a predetermined timeout, it retransmits the request message to the agreement and execution clusters until it obtains a correct response. If the client continues to retransmit the request each successive timeout, the agreement cluster will eventually commit a sequence number to the request [3]. A client can receive a correct response to a retransmitted request in two ways as follows.

If an agreement replica j receives a retransmitted request and it has received an execution certificate, i.e., a set of messages $\{\langle \text{EXEC-REPORT}, n, i, \overline{R} \rangle_i\}, i \in E|f + 1$, from the execution cluster, it sends a $\langle \text{REPLY-AFFIRM}, v, t, c, j, \overline{R} \rangle_j$ message to the client. An affirmation certificate for R is a set of messages $\{\langle \text{REPLY-AFFIRM}, v, t, c, j, \overline{R} \rangle_j\}, j \in A|g + 1$, received from the agreement cluster and at least one response $\langle \text{REPLY}, v, t, c, i, R \rangle_i, i \in E$, received from any execution replica. A client considers a response as correct if it obtains either a response certificate or an affirmation certificate for the response.

4.2 Checkpointing

Execution replicas periodically construct checkpoints of application state. The checkpoint serves two purposes. First, it enables execution replicas that were just started or that fell behind to obtain a recent consistent copy of the application state. Second, it enables agreement replicas to garbage collect their pending logs for requests that have completed execution. The checkpoints are constructed at predetermined request sequence numbers, e.g., when it is exactly divisible by 1024.

An execution replica i generates a checkpoint and sends a $\langle \text{CHECKPOINT}, n, i, \overline{C} \rangle_i$ to all other execution replicas as well as all agreement replicas. A checkpoint certificate is a set of messages $\{\langle \text{CHECKPOINT}, n, i, \overline{C} \rangle_i\}, i \in E|f + 1$. When an execution replica receives a checkpoint certificate with a sequence number n, it considers the checkpoint as stable and discards earlier checkpoints and request commit certificates with lower sequence numbers that it received from the agreement cluster. When an agreement replica receives a checkpoint certificate with sequence number n and it has received execution certificates for all requests with lower sequence numbers, it discards all messages in its log with lower sequence numbers.

4.3 Fault Detection

The agreement cluster is responsible for detecting faults in the execution cluster. In the normal case, an agreement replica j waits for an execution certificate, i.e., a set of messages $\{\langle \text{EXEC-REPORT}, n, \overline{R} \rangle_i\}, i \in E|f + 1$, for each request from the execution cluster. Replica j

inserts this certificate into a local log ordered by the sequence number of requests. When j does not obtain an execution report certificate for a request within a predetermined timeout, j sends a *recovery request* $\langle \text{RECOVER}, j, n\rangle_j$ to a subset of the hypervisors controlling the f standby execution replicas. The number of recovery requests issued by an agreement replica is at least as large as $f+1$ minus the size of the smallest set of valid and matching execution reports for the request that triggered the recovery.

When the hypervisor of an execution replica i receives a recovery certificate $\{\langle \text{RECOVER}, j, n\rangle_j\}, j \in A|g+1$ from the agreement cluster, it starts up the local execution replica.

4.4 Replica Recovery

When an execution replica k starts up, it neither has any application state nor a pending log of requests. Replica k's situation is similar to that of a long-lost replica that may have missed several previous checkpoints and wishes to catch up in existing BFT systems [3, 22, 28, 14]. Since these systems are designed for an asynchronous environment, they can correctly re-integrate up to f such recovering replicas provided there are at least $f+1$ other correct execution replicas. The recovering replica must obtain the most recent checkpoint of the entire application state from existing replicas and verify that it is correct. Unfortunately, checkpoint transfer and verification can take an unacceptably long time for applications with a large state. Worse, unlike previous BFT systems that can leverage copy-on-write techniques and incremental cryptography schemes to transfer only the objects modified since the last checkpoint, a recovering ZZ replica has no previous checkpoints.

To reduce recovery latency, ZZ uses a recovery scheme that amortizes the cost of state transfer across many requests. A recovering execution replica k first obtains an ordered log of committed requests since the most recent checkpoint from the agreement cluster. Let m denote the sequence number corresponding to the most recent checkpoint and let $n \geq m+1$ denote the sequence number of the most recent request with a valid commit certificate. Replica k begins to replay in order the requests in $[m+1, n]$.

4.5 Amortized State Transfer

How does replica k begin to execute requests without any application state? Instead of performing an expensive transfer of the entire state upfront, a recovering ZZ replica fetches and verifies the state necessary to execute each request on demand. After fetching the pending log of committed requests from the agreement cluster, k obtains a checkpoint certificate $\{\langle \text{CHECKPOINT}, n, i, \overline{C}\rangle_i\}, i \in E|g+1$ from any (execution or agreement) replica. Replica k also obtains valid digests for each object in the checkpoint from any execution replica. The checkpoint digest $\overline{C}$ is computed over the digests of individual object digests, so k can verify that it has received valid objects from a correct replica.

After obtaining the checkpoint certificate and object digests, replica k begins to execute in order the committed requests $[m+1, n]$. Let Q be the first request that reads from or writes to some object p since the most recent checkpoint. To execute Q, replica k fetches p on demand from any execution replica that can provide an object consistent with p's digest that k has already verified. Replica k continues executing requests in sequence number order fetching new objects on demand until it obtains a stable checkpoint.

Recovery is complete only when replica k has obtained a stable checkpoint. Since on-demand fetches only fetch objects touched by requests, they are not sufficient for k to obtain a stable checkpoint, so the replica also fetches the remaining state in the background. With an infinite window of vulnerability, this optimization is unnecessary.

4.6 Safety and Liveness Properties

We state the safety and liveness properties ensured by ZZ and outline the proofs. Due to space constraints, we defer formal proofs to the appendix.

ZZ ensures the safety property that if a correct client obtains either a response certificate or an affirmation certificate for a response $\langle \text{REPLY}, v, t, c, j, R\rangle_j$, then 1) the client issued a request $\langle \text{REQUEST}, o, t, c\rangle_c$ earlier; 2) all correct replicas agree on the sequence number n of that request and on the order of all requests with sequence numbers in $[1, n]$; 3) the value of the reply R is the reply that a single correct replica would have produced if it started with a default initial state S_0 and executed each operation $o_i, 1 \leq i \leq n$, in that order, where o_i denotes the operation requested by the request to which sequence number i was assigned.

The first claim follows from the fact that the agreement cluster generates valid commit certificates only for valid client requests and the second follows from the safety of the agreement protocol [4] that ensures that no two requests are assigned the same sequence number. The third claim follows from the fact that, with an infinite window of vulnerability, there is always at least one correct execution replica as at most f can be faulty.

ZZ ensures the liveness property that if a correct client sends a request with a timestamp exceeding previous requests and repeatedly retransmits the request, then it will eventually receive a response certificate or an affirmation certificate. We need eventual synchrony to be show this liveness property. If the client repeatedly retransmits the request, then the agreement cluster will eventually assign a sequence number to the request [4]

and produce a commit certificate. The existence of at least one correct execution replica ensures that the client gets at least one valid (but yet uncertified) response. The agreement cluster ensures that it either obtains an execution certificate or causes all $2f + 1$ execution replicas to be active eventually. In either case, the agreement cluster will eventually obtains an execution certificate, ensuring that the client eventually obtains an affirmation certificate.

4.7 Optimizations

4.7.1 Reducing the Window of Vulnerability

ZZ's current implementation assumes an infinite window of vulnerability, i.e., the length of time in which up to f faults can occur. However, this assumption can be relaxed using proactive recovery [4]. By periodically forcing replicas to recover to a known clean state, proactive recovery allows for a configurable finite window of vulnerability. One catch in ZZ is that because it runs only $f+1$ execution replicas during correct operation, taking one down for recovery can render the system unavailable. Fortunately, ZZ can exploit virtualization to start a new VM replica in parallel to the replica being recovered [8], and take down the latter after the new replica has obtained a stable checkpoint. This technique incurs no downtime as the replica being (proactively) recovered is not faulty. For safety to be maintained, the proactive recovery interval, and therefore the achievable window of vulnerability, must be at least several times as long as the maximum time for f replicas to obtain a stable checkpoint.

4.7.2 Separating State Updates and Execution

ZZ allows for two other straightforward optimizations during recovery. First, instead of actually replaying the requests since the most recent stable checkpoint, a recovering replica j can simply apply the writes in sequence number order while fetching the most recent checkpoint of the objects being written to on demand. For compute-intensive applications with infrequent and small writes, this optimization can significantly reduce recovery latency. However, enabling this optimization using MACs instead of digital signatures requires coordination with the agreement cluster. To this end, an execution request includes a digest of the writes if any performed by each request in EXEC-REPORT messages, which allows the agreement cluster to affirm the sequence of writes obtained by the recovering replica from (at least one) correct execution replica. Second, the recovering replica can jump directly to the request with the sequence number that caused a fault to be detected. As before, the replica fetches the objects being read or written to by each request on demand. If the object was modified by any request since the most recent stable checkpoint, the recovering replica just applies the most recent write to that object before executing the request. This optimization amortizes the cost of applying state updates further reducing the latency penalty experienced by the request that triggered a fault.

4.7.3 Fault Detection Timeouts

ZZ relies on timeouts to detect faults in execution replicas. This opens up a potential performance vulnerability. A low value of the timeout can trigger fault detection even when the delays are benign and needlessly start new replicas. On the other hand, a high value of the timeout can be exploited by faulty replicas to degrade performance as they can delay sending each response to the agreement cluster until just before the timeout. The former can take away ZZ's savings in replication cost as it can end up running more than $f + 1$ (and up to $2f + 1$) replicas even during graceful periods. The latter hurts performance under faults. Note that safety is not violated in either case.

To address this problem, we suggest the following simple heuristic procedure for estimating timeouts. Upon receiving the first response to a request committed to sequence number n, an agreement replica sets the timeout τ_n to Kt_1, where t_1 is the response time of the first response and K is a pre-configured variance bound. If the agreement replica does not receive f more matching responses within τ_n, then it triggers a fault. If the fault was triggered because of receiving fewer than f responses, it requests the initiation f new execution replicas. If the fault was triggered because of receiving $f+1$ mismatched responses, the number of new execution replicas that it requests to be initiated is $f + 1$ minus the size of the smallest matching set of responses.

As before, when the new execution replicas have generated an EXEC-REPORT for the request that triggered a fault, an agreement replica requests the shutdown of 1) all existing execution replicas that produced a wrong response, and 2) randomly chosen (using a procedure consistent across different agreement replicas) additional execution replicas so as to keep the number of active execution replicas at $f + 1$.

LEMMA 1 *If correct execution is guaranteed to return responses within times varying by at most a factor k, and a correct agreement replica triggers a fault, then at least one execution replica is faulty.*

Note that a faulty execution replica may instantaneously return a garbage response causing a timeout, but this case trivially satisfies the above lemma. If a faulty execution replica guessing the right response and instantaneously returning it is perceived as a credible

threat, the shutdown strategy above can always choose to include the execution replica returning the earliest (correct) response in the second step instead of a purely random strategy. With this we have,

LEMMA 2 *If correct execution is guaranteed to return responses within times varying by at most a factor k, and a correct agreement replica triggers a fault, at least one faulty execution replica gets shut down.*

In practice, correct execution replicas may sometimes violate the variance bound due to long but benign execution or network delays, causing a *false timeout*. However, the following lemma bounds the probability of this event for realistic delay distributions. Suppose the response times of the $f+1$ replicas are given by random variables independently and identically distributed as Ψ. Let $\Psi_{(1)}$ and $\Psi_{(f+1)}$ be the first and $(f+1)$th order statistic, or the minimum and maximum, of these random variables.

LEMMA 3 *The probability of a false timeout* Π_1 *is at most* $\int_{t=0}^{\infty} P[\Psi_{(1)} \leq t] P[\Psi_{(f+1)} > Kt] dt$.

The above lemma can be used to numerically bound the probability of a false timeout. For example, if Ψ is exponential with mean $\frac{1}{\lambda}$, then $\Psi_{(1)}$ is also an exponential with mean $\frac{1}{(f+1)\lambda}$, and $\Psi_{(f+1)}$ is given by $P[\Psi_{(f+1)} < t] = (1 - e^{-\lambda t})^{f+1}$.

A faulty replica can potentially inflate response times by up to a factor of K compared to a fault-free system. However, the expected value of such delays, referred to as *inordinate* delays, is bounded as follows.

LEMMA 4 *A faulty replica can inordinately delay the response time by a factor greater than* η *with a probability* Π_2 *that is at most* $\int_{t=0}^{\infty} P[\Psi_{(1)} < t] P[\Psi_{(f+1)} < Kt/\eta]$.

Proactive recovery can further limit performance degradation under faults by limiting the length of time for which a faulty replica operates undetected. Suppose each replica is proactively recovered once every R seconds or less. Let the corresponding mean time to failure be U and the mean time for a newly started replica to obtain a stable checkpoint be D, both exponentially distributed. Note that R must be greater than D [4].

THEOREM 1 *The expected replication cost of ZZ is less than* $(1+r)f+1$, *where* $r = D/U + \Pi_1$. *Faulty replicas can inordinately delay requests by a factor* η *with probability at most* Π_2 *for a fraction of time* fR/U .

Formal proofs for the above claims may be found in the appendix.

5 ZZ Implementation

We implemented ZZ by enhancing the BASE library so as to 1) use virtual machines to house and run replicas, 2) incorporate ZZ's checkpointing, fault detection, rapid recovery and fault-mode execution mechanisms, and 3) use file system snapshots to assist checkpointing.

5.1 Replica Control Daemon

We have implemented a ZZ replica control daemon that runs on each physical machine and is responsible for starting and stopping replicas after faults are detected. The control daemon, which runs in Xen's Domain-0, uses the certificate scheme described in Section 4.3 to ensure that it only starts or stops replicas when enough non-faulty replicas agree that it should do so.

Inactive replicas are maintained in either a paused state, where they have no CPU cost but incur a small memory overhead on the system, or hibernated to disk which utilizes no resources other than disk space. Paused replicas can be initialized very quickly after a fault. To optimize the wakeup latency of replicas hibernating on disk, ZZ uses a paged-out restore technique that exploits the fact that hibernating replicas initially have no useful application state in memory, and thus can be created with a bare minimum allocation of 128MB of RAM (which reduces their disk footprint and load times). After being restored, their memory allocation is increased to the desired level. Although the VM will immediately have access to its expanded memory allocation, there may be an application dependent period of reduced performance if data needs to be paged in.

5.2 Recovery

The implementation of ZZ's recovery protocol described in Section 4.4 proceeds in the following steps.

1. Fault Detection: When an agreement replica receives $f+1$ output messages from execution replicas that are not all identical, it sends wake-up messages to the replica control daemons on f servers not already hosting execution replicas for the application.

2. VM Wake-up: When a replica control daemon receives $f+1$ wake-up messages it attempts to "unpause" a replica if available, and if not, spawns a new VM by loading a replica hibernated on disk.

3. Checkpoint metadata transfer: A replica upon startup obtains the log of committed requests, checkpoint metadata, and any memory state corresponding the most recent stable checkpoint. The replica also obtain access to the latest disk snapshots created by all replicas.

4. Replay: The replica replays requests prior to the fault that modified any application state.

5. On-demand verification: The replica attempts to get the state required for each request from another replica's

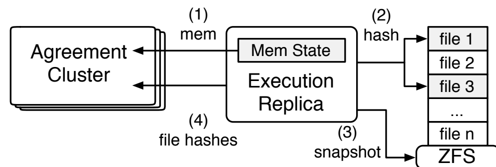


Figure 4: For each checkpoint an execution replica (1) sends any modified memory state, (2) creates hashes for any modified disk files, (3) creates a ZFS snapshot, and (4) returns the list of hashes to agreement nodes.

disk snapshot, and verifies the file contents against the disk hashes contained in the checkpoint metadata. Upon retrieving a valid file, the replica copies it to its local disk and directs all future accesses to that version of the file. *6. Eliminate faulty replicas*: With $2f + 1$ replies, the agreement cluster determines which of the original execution replicas were faulty. Agreement replicas send shutdown messages to the replica control daemons listing the replicas to be terminated.

5.3 Exploiting File System Snapshots

Checkpointing in ZZ relies on the existing mechanisms in the BASE library to save the protocol state of the agreement nodes and any memory state used by the application on the execution nodes. In addition, to efficiently checkpoint disk state of the application, we rely on the snapshot mechanism supported by modern journaled file systems [29, 20]. Creating disk snapshots is efficient because copy-on-write techniques prevent the need for duplicate disk blocks to be created, and the snapshot overhead is independent of the disk state of the application. ZZ uses ZFS for snapshot support, and works with both the native Solaris and user-space Linux ZFS implementations.

ZZ includes meta-information about the disk state in the checkpoint so that the recovery nodes can validate the disk snapshots created by other execution nodes. To do so, execution replicas create a cryptographic hash for each file in the disk snapshot and send it to the agreement cluster as part of the checkpoint certificate as shown in Figure 4. Hashes are computed only for those files that have been modified since the previous epoch; hashes from the previous epoch are reused for unmodified files to save computation overheads.

Tracking Disk State Changes: The BASE library requires all state, either objects in memory or files on disk, to be registered with the library. In ZZ we have simplified the tracking of disk state so that it can be handled transparently without modifications to the application. We define functions bft_fopen() and bft_fwrite() which replace the ordinary fopen() and fwrite() calls in an application. The bft_fwrite() function invokes the modify() call of the BASE library which must be issued

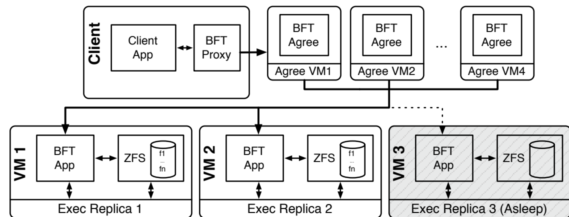


Figure 5: Experimental setup for a basic ZZ BFT service.

whenever a state object is being edited. This ensures that any files which are modified during an epoch will be rehashed during checkpoint creation.

For the initial execution replicas, the bft_fopen() call is identical to fopen(). However, for the additional replicas which are spawned after a fault, the bft_fopen call is used to retrieve files from the disk snapshots and copy it to the replica's own disk on demand. When a recovering replica first tries to open a file, it calls bft_fopen(foo), but the replica will not yet have a local copy of the file. The recovery replica fetches a copy of the file from any replica and verifies it against the hash contained in the most recent checkpoint. If the hashes do not match, the recovery replica requests the file from a different replica, until a matching copy is found and copied to its own disk.

6 Experimental Evaluation

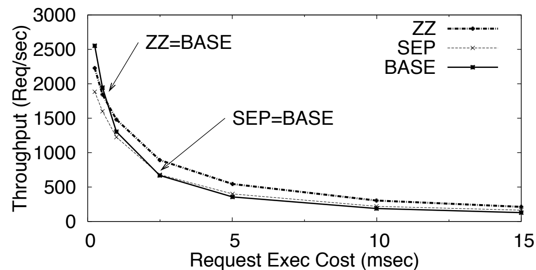
6.1 Experiment Setup

Our experimental data-center setup uses a cluster of 2.12 GHz 64-bit dual-core Dell servers, each with 4GB RAM. Each machine runs a Xen v3.1 hypervisor and Xen virtual machines. Both domain-0 (the controller domain in Xen) as well as the individual VMs run the CentOS 5.1 Linux distribution with the 2.6.18 Linux kernel and the user space ZFS filesystem. All machines are interconnected over gigabit ethernet. Figure 5 shows the setup for agreement and execution replicas of a generic BFT app for $g = f = 1$; multiple such applications are assumed to be run in a BFT data center.

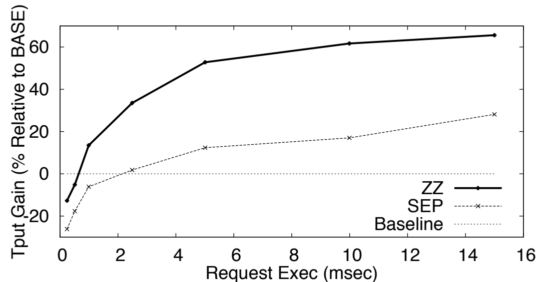
Our experiments involve three fault-tolerant server applications: a Web Server, an NFS server, and a toy client-server microbenchmark.

Fault-tolerant Web Server: We have implemented a BFT-aware HTTP 1.0 Web server that mimics a dynamic web site with server side scripting. The request execution time is configurable to simulate more complex request processing. We generate web workloads using *httperf* clients which contact a local BFT web proxy that submits the requests to the agreement nodes.

Fault-tolerant NFS: BASE provides an NFS client relay and a BFT wrapper for the standard NFS server. We have extended this to support ZZ's on demand state



(a) Application Throughput



(b) ZZ Throughput Gain

Figure 6: When resources are constrained, ZZ significantly increases system throughput by using fewer replicas.

transfer which allows a recovery replica to obtain file system state from ZFS snapshots as it processes each request.

Client-Server Microbenchmark: We utilize the simple client-server application from the BASE library to measure ZZ's performance for *null* requests and to study it's recovery costs under different application state scenarios.

Our experiments compare three systems: ZZ, BASE, and Separated (SEP). BASE is the standard BFT library described in [22]. SEP is our extension of BASE which separates the agreement and execution replicas, and requires $3f+1$ agreement and $2f+1$ execution replicas similar to [28]. ZZ also requires $3f+1$ agreement replicas, but extends SEP to use only $f+1$ active execution replicas, with an additional f sleeping replicas.

6.2 Graceful Performance

We study the graceful performance of ZZ by emulating a shared hosting environment running four independent web apps on four machines. Table 2 shows the placement of agreement and execution replicas on the four hosts. Since the agreement and execution clusters can independently handle faults, each host can have at most one replica of each type per application.

6.2.1 Throughput

We first analyze the impact of request execution cost under ZZ, SEP, and BASE, which require $f+1$, $2f+1$,

	Graceful performance				After failure		
	h_1	h_2	h_3	h_4	h_1	h_2	h_3
BASE	1234	1234	1234	1234	1234	1234	1234
SEP_{Agree}	1234	1234	1234	1234	1234	1234	1234
SEP_{Exec}	134	124	123	234	134	124	123
ZZ_{Agree}	1234	1234	1234	1234	1234	1234	1234
ZZ_{Exec}	12	12	34	34	12**3**	12**4**	34
ZZ_{Sleep}	3	4	1	2			1

Table 2: Placement of the 4 web servers' virtual machines (denoted 1 to 4) on the 4 data center hosts (h_1 to h_4) under graceful performance and on the 3 remaining hosts after h_4 failure.

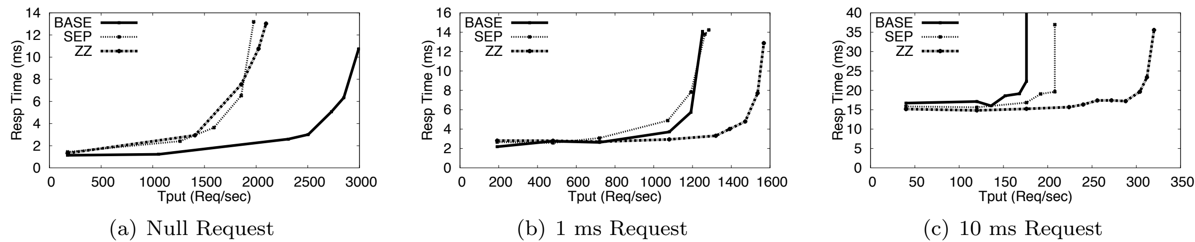
and $3f+1$ execution replicas per web server respectively. Figure 6(a) compares the throughput of each system as the execution cost per web request is adjusted. When execution cost averages 100 μs, BASE performs the best since the agreement overhead dominates the cost of processing each request and our implementation of separation incurs additional cost for the agreement replicas. However, for execution costs exceeding 0.75 ms, the execution replicas become the system bottleneck. As shown in Figure 6(b), ZZ begins to outperform BASE at this point, and performs increasingly better compared to both BASE and SEP as execution cost rises. SEP surpasses BASE for request costs over 2ms, but cannot obtain the throughput of ZZ since it requires $2f+1$ replicas instead of only $f+1$. ZZ provides as much as a 66% increase in application throughput relative to BASE for requests with large execution costs.

6.2.2 Latency

This experiment further characterizes the performance of ZZ in graceful operation by examining the relation between throughput and response time for different request types. Figure 7 shows the relation between throughput and response time for increasingly CPU intensive request types. For null requests or at very low loads, Figure 7(a), BASE beats SEP and ZZ because it has less agreement overhead. At 1ms, ZZ's use of fewer execution replicas enables it to increase the maximum throughput by 25% over both SEP and BASE. When the execution cost reaches 10ms, SEP outperforms BASE since it uses $2f+1$ instead of $3f+1$ replicas. ZZ provides a 33% improvement over SEP, showing the benefit of further reducing to $f+1$.

6.3 Simultaneous Failures

When several applications are multiplexed on a single physical host, a faulty node can impact all its running applications. In this experiment, we simulate a malicious hypervisor on one of the four hosts that causes multiple applications to experience faults simultaneously. Host



(a) Null Request (b) 1 ms Request (c) 10 ms Request

Figure 7: For high execution costs, ZZ achieves both higher throughput and lower response times.

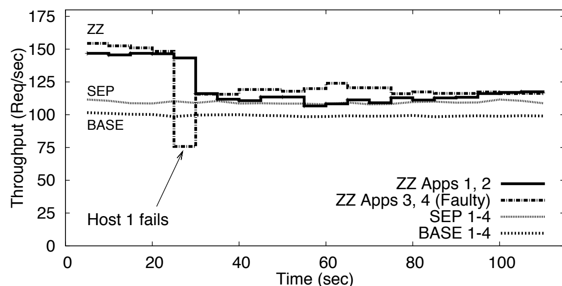
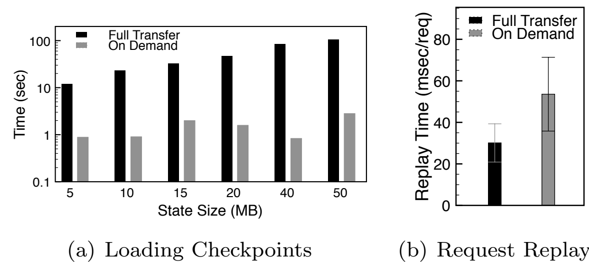


Figure 8: Under simultaneous failures of several applications, ZZ quickly recovers and still maintains good throughput.



(a) Loading Checkpoints (b) Request Replay

Figure 9: (a) The cost of full state transfer increases with state size. (b) On Demand incurs overhead when replaying requests since state objects must be verified.

h_4 in Table 2 is set as a faulty machine and is configured to cause faults on all of its replicas 20 seconds into the experiment as shown in Figure 8. For ZZ, the failure of h_4 directly impacts web servers 3 and 4 which have active execution replicas there. The replica for server 2 is a sleeping replica, so its corruption has no effect on the system. The failure also brings down one agreement replica for each of the web servers, however they are able to mask these failures since $2f + 1$ correct agreement replicas remain on other nodes.

ZZ recognizes the corrupt execution replicas when it detects disagreement on the request output of each service. It responds by waking up the sleeping replicas on hosts h_1 and h_2. After a short recovery period (further analyzed in the next section), ZZ's performance is similar to that of SEP with three active execution replicas competing for resources on h_1 and h_2. Even though h_3 only has two active VMs and uses less resources with ZZ, applications 3 and 4 have to wait for responses from h_1 and h_2 to make progress. Both ZZ and SEP maintain a higher throughput than BASE that runs all applications on all hosts.

6.4 Recovery Cost

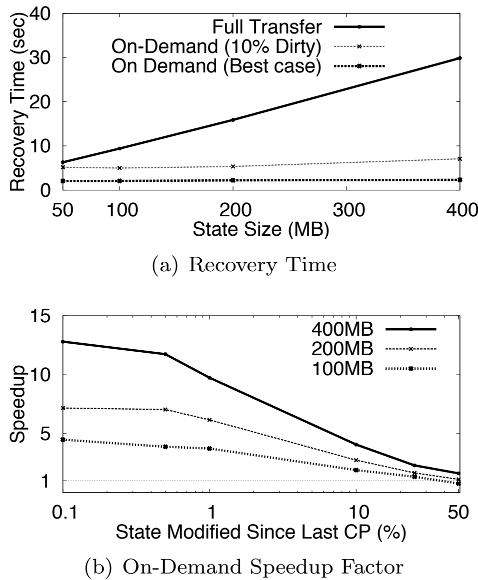
The following experiments study the cost of recovering replicas in more detail using both microbenchmarks and our fault tolerant NFS server. We study the recovery cost, which we define as the delay from when the agreement cluster detects a fault until the client receives the correct response.

6.4.1 NFS Recovery Costs

We investigate the NFS server recovery cost for a workload that creates 200 files of equal size before encountering a fault while reading back a file that has been corrupted by one of the replicas. We vary the size of the files to adjust the total state maintained by the application, which also impacts the number of requests which need to be replayed after the fault.

ZZ uses an on-demand transfer scheme for delivering application state to newly recovered replicas. Figure 9(a) shows the time for processing the checkpoints when using full transfer or ZZ's on-demand approach (note the log scale). The full state transfer approach performs very poorly since the BFT NFS wrapper must both retrieve the full contents of each file and perform RPC calls to write out all of the files to the actual NFS server. When transferring the full checkpoint, recovery time increases exponentially and state sizes greater than a mere 20 megabytes can take longer than 60 seconds, after which point NFS requests typically will time out. In contrast, the on-demand approach has a constant overhead with an average of 1.4 seconds. This emphasizes the importance of using the on-demand transfer for realistic applications where it is necessary to make some progress in order to prevent application timeouts.

We report the average time per request replayed and the standard deviation for each scheme in Figure 9(b). ZZ's on demand system experiences a higher replay cost due to the added overhead of fetching and verifying state on-demand; it also has a higher variance since the first access to a file incurs more overhead than subsequent



(a) Recovery Time

(b) On-Demand Speedup Factor

Figure 10: The worst case recovery time depends on the amount of state updated between the last checkpoint and the fault.

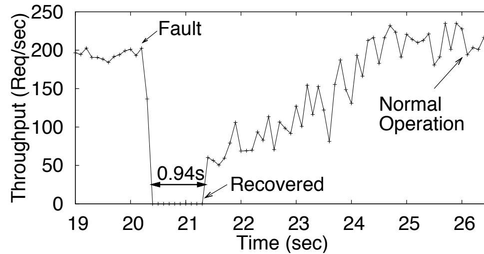


Figure 11: The recovery period lasts for less than a second. At first, requests see higher latency since state must be fetched on-demand.

calls. While ZZ's replay time is larger, the total recovery time is much smaller when using on-demand transfer.

6.4.2 Obtaining State On-Demand

This experiment uses a BFT client-server microbenchmark which processes requests with negligible execution cost to study the recovery cost after faults are caused in applications with different state sizes.

In the best case, when a fault occurs immediately after a checkpoint, a new replica only needs to load and resume from a checkpoint. We observe a constant time of about 2s to recover from state sizes ranging from 50 to 400MB as illustrated by the On-Demand (Best Case) line in Figure 10(a). If the fault occurs immediately *before* a checkpoint, old requests need to be replayed before recovery replicas can respond to the client. In this case, the cost of on-demand recovery varies depending on the amount of application state that was modified since the last checkpoint. The "10% Dirty" line shows the recovery cost when 10% of the application's state needs to be fetched during replay. In that case, ZZ's recovery time varies from 5.2s to 7.1s for states of 50 and 400MB, respectively. This remains much faster than the Full Transfer technique which requires over 30s to transfer and verify 400MB of state.

The tradeoff between amount of dirty state and recovery speed is further studied in Figure 10(b). The benefit of getting state on-demand is very significant with less than 1% dirty state and it decreases the recovery time by at least five times when less than 10% of a 400MB state is modified between the checkpoint and a fault. When a 100MB state is more than 50% dirty, it becomes more expensive to replay than to perform a full transfer. Fortunately, we have measured the additional cost of ZFS checkpoints at 0.03s, making it practical to checkpoint every few seconds, during which time most applications will only modify a small fraction of their total application state.

While obtaining state on-demand significantly reduces the initial recovery time, it may increase the latency for subsequent requests since they must fetch and verify state. In this experiment we examine the throughput and latency of requests after a fault has occurred. The client sends a series of requests involving random accesses to 100KB state objects.

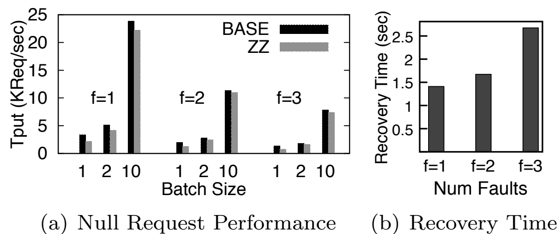
As shown in Figure 11, we inject a fault after 20.2 seconds. The faulty request experiences a sub-second recovery period, after which the application can handle new requests. The mean request latency prior to the fault is 5 milliseconds with very little variation. The latency of requests after the fault has a bimodal distribution depending on whether the request accesses a file that has already been fetched or one which needs to be fetched and verified. The long requests, which include state verification and transfer, take an average of 20 milliseconds. As the recovery replica rebuilds its local state, the throughput rises since the proportion of slow requests decreases. After 26 seconds, the full application state has been loaded by the recovery replica, and the throughput prior to the fault is once again maintained.

6.5 Trade-offs and Discussion

6.5.1 Impact of Multiple Faults

We examine how ZZ's graceful performance and recovery time changes as we adjust f, the number of faults supported by the system when *null* requests are used requiring no execution cost. Figure 12(a) shows that ZZ's graceful mode performance scales similarly to BASE as the number of faults increases. This is expected because the number of cryptographic and network operations rises similarly in each system.

We next examine the recovery latency of the client-server microbenchmark for up to three faults. In each



(a) Null Request Performance (b) Recovery Time

Figure 12: Recovery time increases for larger f from message overhead and increased ZFS operations.

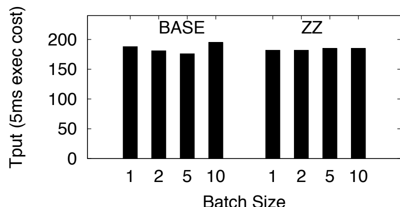


Figure 13: Throughput of ZZ and BASE with different batch sizes for a 5ms request execution time.

case, we maintain f paused recovery replicas. We inject a fault to f of the active execution replicas and measure the recovery time to handle the faulty request. Figure 12(b) shows how the recovery time for $f = 1$ to $f = 3$ increases slightly due to increased message passing and because each ZFS system needs to export snapshots to a larger number of recovering replicas. We believe the overhead can be attributed to our use of the user-space ZFS code that is less optimized than the Solaris kernel module implementation, and messaging overhead which could be decreased with hardware multicast.

6.5.2 Agreement Protocol Performance

The separated architecture employed by ZZ increases agreement protocol overhead compared to BASE. Various agreement protocol optimizations exist such as request batching, but these may have less effect when request execution costs are non-trivial. While Figure 12(a) shows a large benefit of batching for null requests, Figure 13 depicts a similar experiment with a request execution time of 5ms. We observe that batching improvements become insignificant with non-trivial execution costs. This demonstrates the importance of reducing execution costs, not just agreement overhead, for real applications.

6.5.3 Maintaining Spare VMs

In our previous experiments, the recovery VMs were kept in a paused state which provides a very fast recovery but consumes memory. Larger non-interactive applications that have less stringent latency requirements can keep their recovery VMs hibernated on disk, removing the memory pressure on the system.

With a naive approach, maintaining replicas hibernated to disk can increase recovery latency by a factor proportional to the amount of memory allocated to each VM. This is because restoring a hibernated VM involves loading the VM's full memory contents from disk. The table below shows how our paged-out restore technique can reduce the startup time for a VM with a 2GB memory allocation from over 40 seconds to less than 6 seconds.

Operation	Time (sec)
Xen Restore (2GB image)	44.0
Paged-out Restore (128MB→2GB)	5.88
Unpause VM	0.29
ZFS Clone	0.60

ZZ utilizes ZFS to simplify checkpoint creation at low cost. The ZFS clone operation is used during recovery to make snapshots from the previous checkpoint available to the recovery VMs. This can be done in parallel with initializing the recovery VMs, and incurs only minimal latency.

7 Related Work

This section discusses work not covered elsewhere. *A Brief History of BFT SMR:* Lamport, Shostak, and Pease [16] introduced the problem of Byzantine agreement. Lamport also introduced the state machine replication approach [17] (with a popular tutorial by Schneider [23]) that relies on consensus to establish an order on requests. Consensus in the presence of asynchrony and faults has seen almost three decades of research. Dwork et al. [9] established a lower bound of $3f + 1$ replicas for Byzantine agreement given partial synchrony, i.e., an unknown but fixed upper bound on message delivery time. The classic FLP [10] result showed that no agreement protocol is guaranteed to terminate with even one (benignly) faulty node in an asynchronous environment. Viewstamped replication [19] and Paxos [15] describe an asynchronous state machine replication approach that is safe despite crash failures.

Early BFT systems [21, 13] incurred a prohibitively high overhead and relied on failure detectors to exclude faulty replicas. However, accurate failure detectors are not achievable under asynchrony, thus these systems effectively relied on synchrony for safety. Castro and Liskov's PBFT [3] introduced a BFT SMR-based system that relied on synchrony only for liveness. The view change protocol at the core of PBFT shares similarities with viewstamped replication [19] or Paxos [15] but incurs a replication cost of at least $3f + 1$ for safety. PBFT showed that the latency and throughput overhead of BFT can be low enough to be practical. The FARSITE system [2] reduces the replication cost of a BFT file-system to $f + 1$; in comparison, ZZ has simi-

lar goals, but is able to provide the same cost reduction for any application which can be represented by a more general SMR system. ZZ draws inspiration from Cheap Paxos [18], which advocated the use of cheaper auxiliary nodes used only to handle crash failures of main nodes. Our contribution lies in extending the idea to Byzantine faults and demonstrating its practicality through a system design and implementation.

BFT and Virtualization: Virtualization has been used in several BFT systems recently since it provides a clean way to isolate services. The VM-FIT systems exploits virtualization for isolation and to allow for more efficient proactive recovery [8]. Like ZZ, VM-FIT employs an amortized state transfer mechanism to efficiently update replicas, but it is designed for a system running 2f+1 execution nodes. The idea of "reactive recovery", where faulty replicas are replaced after fault detection, was used in [25], which also employed virtualization to provide isolation between different types of replicas. In ZZ, reactive recovery is not an optional optimization, but a requirement since in order to make progress it must immediately instantiate new replicas after faults are detected.

Rapid activation of virtual machine replicas for dynamic capacity provisioning has been studied in [12]. In contrast, ZZ uses VM replicas for high availability rather than scale. Live migration of virtual machines was proposed in [5]. Such techniques can be employed by ZZ to intelligently distribute agreement and execution replicas in the data center, although we leave an implementation to future work. Virtualization has also been employed for security. Potemkin uses dynamically invocation of virtual machines to serve as a honeypot for security attacks [27]. Terra is a virtual machine platform for trusted computing that employs a trusted hypervisor [11]; ZZ allows hypervisors to be Byzantine faulty.

8 Conclusions

In this paper, we presented ZZ a new execution approach that can be interfaced with existing BFT-SMR agreement protocols to reduce the replication cost from $2f+1$ to practically $f+1$. Our key insight was to use $f+1$ execution replicas in the normal case and to activate additional VM replicas only upon failures. We implemented ZZ using the BASE library and the Xen virtual machine and evaluated it on a prototype data center that emulates a shared hosting environment. The key results from our evaluation are as follows. (1) In a prototype data center with four BFT web servers, ZZ lowers response times and improves throughput by up to 66% and 33% in the fault-free case, when compared to systems using $3f+1$ (BASE) and $2f+1$ replicas, respectively. (2) In the presence of multiple application failures, after a short recovery period, ZZ performs as well or better than $2f+1$ replication and still outperforms BASE's $3f+1$ replication. (3) The use of paused virtual machine replicas and on-demand state fetching allows ZZ to achieve sub-second recovery times. (4) We find that batching in the agreement nodes, which significantly improves the performance of null execution requests, yields no perceptible improvements for realistic applications with non-trivial request execution costs. Overall, our results demonstrate that, in shared data centers that host multiple applications with substantial request execution costs, ZZ can be a practical and cost-effective approach for providing BFT. Our future work will focus on using ZZ to provide BFT in multi-tier web services, by deploying our techniques at each tier and measuring the end-to-end impact.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, February 1999.

[4] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 2002.

[5] C. Clark, K. Fraser, S. Hand, J. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfiel. Live migration of virtual machines. In *Proceedings of Usenix Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[6] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making byzantine fault tolerant systems tolerate byzantine faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, April 2009.

[7] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington, November 2006.

[8] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. Efficient state transfer for hypervisor-based proactive recovery. In *WRAITS '08: Proceedings of the 2nd workshop on Recent advances on intrusiton-tolerant systems*, pages 1–6, New York, NY, USA, 2008. ACM.

[9] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[11] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: a virtual machine-based platform for trusted computing. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 193–206, New York, NY, USA, 2003. ACM Press.

[12] Laura Grit, David Irwin, , Aydan Yumerefendi, and Jeff Chase. Virtual machine hosting for networked clusters: Building the foundations for autonomic orchestration. In *In the First International Workshop on Virtualization Technology in Distributed Computing (VTDC)*, November 2006.

[13] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The securering protocols for securing group communication. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 3*, page 317, Washington, DC, USA, 1998. IEEE Computer Society.

[14] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, New York, NY, USA, 2007. ACM.

[15] L. Lamport. Part time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.

[16] L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[17] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[18] Leslie Lamport and Mike Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Depen dable Systems and Networks*, page 307, Washington, DC, USA, 2004. IEEE Computer Society.

[19] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: a general primary copy. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, New York, NY, USA, 1988. ACM.

[20] Z. Peterson and R. Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, 2005.

[21] Michael K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 99–110, London, UK, 1995. Springer-Verlag.

[22] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2001. ACM Press.

[23] F. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A tutorial. *Computing Surveys*, 22(3):299–319, September 1990.

[24] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI '08: Proceedings of the Usenix Symposium on Networked System Design and Implementation*, 2008.

[25] Paulo Sousa, Alysson N. Bessani, Miguel Correia, Nuno F. Neves, and Paulo Verissimo. Resilient intrusion tolerance through proactive and reactive recovery. In *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing*, Washington, DC, USA, 2007.

[26] Ben Vandiver, Hari Balakrishnan, Barbara Liskov, and Sam Madden. Tolerating byzantine faults in database systems using commit barrier scheduling. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP)*, Stevenson, Washington, USA, October 2007.

[27] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of SOSP*, 2005.

[28] J. Yin, J.P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[29] Zfs: the last word in file systems. http://www.sun.com/2004-0914/feature/.

9 Appendix

This appendix contains formal proofs of the lemmas and theorems in the body of the paper. This section also rectifies two errors (described in the two footnotes here) present in the body of the paper.

9.1 Fault Detection Timeouts

9.1.1 Agreeing on the Blame

LEMMA 1. *If correct execution is guaranteed to return responses within times varying by at most a factor K, and a correct agreement replica triggers a fault, then at least one execution replica is faulty.*

Proof: A correct agreement replica i triggers a fault in two cases.

Case 1: Replica i received fewer than f responses in the time interval $[t_1, Kt_1]$ since dispatching the request. By assumption, correct execution is guaranteed to return $f+1$ responses within times varying by at most a factor K. So, at least one execution replica must be faulty.

Case 2: Replica i received $f+1$ responses by time Kt_1 that were not all matching. Since all correct replicas produce matching responses, at least one execution replica must be faulty. ∎

In order to prove Lemma 2, we first prove a simpler Lemma 2A below. We also introduce the notion of *convictable* mismatches that allows us to treat timeouts and mismatches elegantly in a uniform manner.

LEMMA 2A. *If no timeouts occur and the agreement cluster wakes up new execution replicas, at least one faulty execution replica gets shut down.*

Proof: When no timeouts happen, faults and wakeups are only due to mismatched responses. A new execution replica is woken up only if at least $g+1$ replicas report a *convictable* mismatch. We explain what a convictable mismatch means with an example first before defining it formally. Let $f = 1$ and $g = 1$; let E_1, E_2 denote the two execution replicas, and A_1, A_2, A_3, A_4 denote the four agreement replicas. A new execution replica is woken up only if at least two agreement replicas, say A_1 and A_2, report a mismatch as follows: A_1 and A_2 report E_1 as having returned a common response P and report E_2 as having returned a different common response Q. Thus, only the last of the four scenarios below is considered a convictable mismatch. Here, the four values after the colon represent the responses reported by the four agreement replicas respectively.

Unconvictable mismatch
$E_1 : P\ Q\ P\ P$
$E_2 : R\ P\ P\ P$

Unconvictable mismatch
$E_1 : Q\ P\ P\ P$
$E_2 : Q\ P\ P\ P$

Unconvictable mismatch
$E_1 : Q\ P\ P\ P$
$E_2 : P\ P\ P\ P$

Convictable mismatch
$E_1 : Q\ Q\ P\ P$
$E_2 : P\ P\ P\ P$

Definition 1 *To formally define a convictable mismatch, consider a matrix with $f + 1$ rows and $3g + 1$ columns. Each entry (m,n) represents the response reported by agreement replica n as being returned by execution replica m. An execution replica j is convictable iff despite removing up to g entries in row j, the row does not have identical entries. A mismatch is convictable if at least one execution replica is convictable.*

A hypervisor wakes up a new execution replica only when it gathers evidence of a convictable mismatch (that by definition implies that at least $g + 1$ agreement replicas reported a mismatch). A convictable mismatch will lead to at least one faulty replica being shut down. In the above example, a new execution replica will be woken up only in the fourth case, and the corresponding faulty execution replica (E_1) will be shutdown. In all other cases, no wakeups or shutdowns will occur. By using digitally signed wakeup messages, it is easy to ensure that wakeups are necessarily followed by shutdowns. This last detail is necessary to prevent (up to g) faulty agreement replicas from participating in a wakeup but not cooperating (by not reproducing evidence of the convictable mismatch) with the corresponding shutdown.

The formal proof follows from the definition of a convictable mismatch. A convictable mismatch always identifies a convictable execution replica by definition. Wakeups happen only upon a convictable mismatch, and at least one convictable (faulty) execution replica is shut down as a result. ∎

Lemma 2. *If correct execution is guaranteed to return responses within times varying by at most a factor K, and the agreement cluster wakes up new execution replicas, at least one faulty execution replica gets shut down.*[2]

Proof: Agreement replicas can also report a fault due to a timeout in addition to mismatches. As before, at least $g + 1$ agreement replicas must report a timeout in order to trigger a wakeup, and as a result f new execution replicas will be woken up. Several different combinations of timeouts and mismatches are possible as exemplified below.

Convictable timeout:
$E_1 : P\ P\ P\ P$
$E_2 : \hat{P}\ \hat{P}\ P\ P$

Convictable timeout, convictable mismatch:
$E_1 : Q\ Q\ P\ P$
$E_2 : \hat{P}\ \hat{P}\ P\ P$

Convictable timeout, unconvictable mismatch:
$E_1 : P\ Q\ P\ P$
$E_2 : \hat{P}\ \hat{P}\ P\ P$

Convictable timeout, unconvictable mismatch:
$E_1 : P\ P\ P\ P$
$E_2 : \hat{P}\ \hat{Q}\ P\ P$

Unconvictable timeout:
$E_1 : P\ P\ P\ P$
$E_2 : P\ \hat{P}\ P\ P$

Unconvictable timeout, unconvictable mismatch:
$E_1 : P\ Q\ P\ P$
$E_2 : P\ \hat{P}\ P\ P$

Unconvictable timeout, convictable mismatch:
$E_1 : P\ P\ P\ P$
$E_2 : \hat{Q}\ Q\ P\ P$

It is straightforward to determine the corresponding shutdowns with the following assumption: *If a replica returns a correct response after time t_1, then all correct replicas will return a correct response by time Kt_1, i.e., a faulty replica can not return a correct response instantaneously and cause a timeout.*[3]

Now, timeouts can be processed similar to mismatches, i.e., only if they are convictable. We use the notation $\hat{P}$ for a response P received after a timeout and consider it distinct from P. With this notation, we need no change to our wakeup rule, i.e., a hypervisor wakes up a new execution replica only when it gathers evidence of a convictable mismatch. The replicas to be shutdown can be determined from the convictable mismatch certificate as before. Thus, in the above seven examples, the following shutdowns will happen: 1) E_2, 2) E_1, 3) E_2, 4) E_2, 5) no wakeups or shutdowns, 6) no wakeups or shutdowns, 7) E_2. Note that mismatches are considered more egregious than late but correct responses, e.g., in the second example E_2 is not considered faulty as it is clear that E_1 sent an early bad response.

The formal proof follows from the observation that only convictable faults result in wakeups. By definition,

[2]The statement of this lemma rectifies an error in Lemma 2 stated in the body of the paper.

[3]The body of the paper claims that this assumption is not necessary, which we now think is an error.

a convictable fault always identifies at least one faulty execution replica that is shut down after the new execution replicas have produced a (correct) response to the request that triggered the wakeup. ∎

9.1.2 Limiting Frivolous Timeouts

Lemma 2 above showed that frivolous wakeups will not happen if timing assumptions are satisfied. In practice, correct execution replicas may sometimes violate the variance bound due to long but benign execution or network delays, causing a *false timeout*. However, the following lemma bounds the probability of this event for realistic delay distributions. Suppose the response times of the $f+1$ replicas are given by random variables independently and identically distributed as Ψ. Let $\Psi_{(1)}$ and $\Psi_{(f+1)}$ be the first and $(f+1)$th order statistic, or the minimum and maximum, of these random variables.

Lemma 3. The probability of a false timeout Π_1 is at most $\int_{t=0}^{\infty} \mathrm{p}[\Psi_{(1)} \leq t]\mathrm{P}[\Psi_{(f+1)} > Kt]dt$.

Proof: A false timeout occurs when the timing assumptions are not satisfied, i.e., correct execution replicas return responses within times that vary by more than a factor K. The probability of a false timeout when the first response arrives within time t is given by $\mathrm{P}[\Psi_{(1)} \leq t] \cdot \mathrm{P}[\Psi_{(f+1)} > Kt \mid \Psi_{(1)} \leq t]$. The first term is the probability that the first response arrives within time t and the second term is the probability that the $(f+1)$th response arrives within time Kt conditional on the first response arriving within time t. Note that $\Psi_{(1)}$ and $\Psi_{(f+1)}$ are not independent, hence the conditional in the second term. The probability of a false timeout Π_1 is

$$\Pi_1 = \int_{t=0}^{\infty} p[\Psi_{(1)} \leq t] \cdot P[\Psi_{(f+1)} > Kt \mid \Psi_{(1)} = t]dt$$

where the integral is over all possible values of t and the notation $p[.]$ using a small 'p' denotes the pdf[4], i.e., $p[\Psi_{(1)} \leq t] = \frac{d}{dt}(P[\Psi_{(1)} \leq t])$.In order to prove the lemma, we note that removing the conditional in the second term inside the integral strictly increases the second term for all possible values of t. In other words, the probability that the $(f+1)$th response time is greater than Kt is greater than the probability that the $(f+1)$th response time is greater than Kt conditional on the first response time being equal to t. Thus,

$$\Pi_1 \leq \int_{t=0}^{\infty} p[\Psi_{(1)} \leq t] \cdot P[\Psi_{(f+1)} > Kt]dt$$

proving the lemma. ∎

Note that the above lemma is only the probability of a false timeout at a single correct agreement replica. The probability of a new execution replica being woken up is even smaller as $g+1$ agreement replicas must be able to produce a convictable timeout.

Corollary 1 *The probability of a new execution replica being unnecessarily woken up despite all execution replicas being correct is at most* Π_1.

The above lemma can be used to numerically compute the bound on the probability of a false timeout or wakeup. For example, if Ψ is exponential with mean $\frac{1}{\lambda}$, then $\Psi_{(1)}$ is also an exponential with mean $\frac{1}{(f+1)\lambda}$ with a pdf given by $\mathrm{p}[\Psi_{(1)} \leq t] = \lambda(f+1)e^{-\lambda(f+1)t}$; and $\Psi_{(f+1)}$ is given by $\mathrm{P}[\Psi_{(f+1)} > t] = 1-(1-e^{-\lambda t})^{f+1}$.

9.1.3 Limiting Response Time Inflation

Lemma 3 above helps limit the replication cost by limiting the probability of frivolous timeouts. However, a faulty replica can potentially inflate response times by up to a factor of K compared to a fault-free system. Fortunately, the expected value of such delays, referred to as *inordinate* delays, is bounded as follows.

Lemma 4. A faulty replica can inordinately delay the response time by a factor $\eta \in [1, K)$ with a probability Π_2 that is at most $\int_{t=0}^{\infty} p[\Psi_{(1)} \leq t] \cdot P[\Psi_{(f+1)} \leq Kt/\eta]\mathrm{dt}$.

Proof: The reason why we are also able to limit the response time inflation due to faulty replicas is that there is inherent variation even among response times of correct execution replicas. To see this, suppose that correct execution replicas have a mean response time of D=2ms. The "normal" execution response time, i.e., the response time when all execution replicas are correct, in expectation will *not* be 2ms. It will be the maximum of $f+1$ distributions as above each with a mean of 2ms. For example, if response times are exponentially distributed and $f=3$, then the time to receive all $f+1$ responses, or the $(f+1)$th order statistic, is given by $P[\Psi_{(f+1)} < t] = (1-e^{-t/3})^4$. Thus, its expected value is $\int_{t=0}^{\infty} 2(1-e^{-t/2})^3 e^{-t/2} t dt$. Solving this integral numerically yields an expected response time of about 4.2ms.

Suppose exactly one execution replica is correct and the other f are faulty. To cause the most amount of inordinate delay when the correct execution replica produces a response within time t, the faulty execution replicas must return a response after time Kt. On the other hand $f+1$ correct execution replicas would have returned a response time given by $\Psi_{(f+1)}$. Thus, the probability of an inordinate delay by a factor η when a correct replica returns a response at time t is given by $P[\Psi_{(f+1)} < Kt/\eta]$. The probability Π_2 of an inordinate delay by a factor η is thus

[4] The notation $p_{\Psi_{(1)}}(t)$ would be more commonly used, but the subscript overkill seemed confusing.

$$\Pi_2 = \int_{t=0}^{\infty} p[\Psi \leq t] \cdot P[\Psi_{(f+1)} \leq Kt/\eta \mid \Theta] dt$$

where Ψ is the response time distribution of any correct execution replica. The second term includes a conditional on Θ that is the event that at least one of the $f+1$ response times was equal to t. As $K/\eta \geq 1$, as in Lemma 3, we can drop the conditional Θ, which yields

$$\Pi_2 \leq \int_{t=0}^{\infty} p[\Psi \leq t] \cdot P[\Psi_{(f+1)} \leq Kt/\eta] dt$$

The analysis above assumed exactly one correct execution replica, which causes the most inordinate delays. With f correct replicas and one faulty replica, we obtain the (tighter) bound on Π_2 as stated in the lemma. ∎

For the numerical example above if $K = 6, f = 3, D =$ 2ms, the probability that 3 faulty replicas can actually cause an inordinate delay by a factor $\eta = 6$ is less than 0.2. The expected value of η is less than 1.5, i.e., the response time could inordinately increase to an expected value of less than 3ms. Note also that these delay factors are only for the execution times and do not affect the WAN propagation and transmission delays.

9.1.4 Further Limiting Response Time Inflation

Proactive recovery can further limit performance degradation under faults by limiting the length of time for which a faulty replica operates undetected. Suppose each replica is proactively recovered once every R seconds or less. Let the corresponding mean time to failure be U and the mean time for a newly started replica to obtain a stable checkpoint be D, both exponentially distributed. Note that R must be greater than D [4].

Theorem 2 *The expected replication cost of ZZ is less than $(1+r)f+1$, where $r = D/U + \Pi_1$. Faulty replicas can inordinately delay requests by a factor η with probability at most Π_2 for a fraction of time fR/U .*

Proof: The theorem follows from Lemmas 3 and 4 above. The expected replication cost depends on the probability D/U of a replica being faulty and the probability Π_1 of a false timeout. The combined probability is less than $D/U + \Pi_1$. Note that a replica can be faulty for at most D/U time as it will be convicted and shut down, by Lemma 2. The replication cost is less than $(1+r)f+1$ even if we assume that f new replicas are woken up upon each convictable fault. With proactive recovery, a faulty replica remains faulty for at most R time and a fault occurs once every U/f time . Thus, inordinate delays can occur as per Π_2 for at most a fraction fR/U time. ∎

9.2 Safety and Liveness Properties

For a formal proof of the safety and liveness of the agreement protocol, refer [4]. The liveness property follows from the fact that the client eventually either obtains a response certificate or an affirmation certificate, or the agreement cluster activates up to $2f+1$ replicas. The liveness property depends on eventual synchrony and fair message delivery [28]. The safety and liveness properties of the proactive recovery protocol follow from [4] as long as the recovery period and the window of vulnerability are at least several times the time required for a new replica to obtain a stable checkpoint.