

ZZ and the Art of Practical BFT Execution

Timothy Wood, Rahul Singh, Arun Venkataramani,
Prashant Shenoy, And Emmanuel Cecchet
Department of Computer Science, University of Massachusetts Amherst

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 data centers where multiple applications share a physical server, ZZ reduces the aggregate number of execution replicas running in the data center, 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 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 in modern data centers.

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 them in the agreed upon order. Despite numerous efforts to improve the performance or fault scalability of BFT systems [3, 6, 15, 25, 1, 13], existing approaches remain expensive, requiring at least $2f + 1$ replicas to execute each request in order to tolerate f faults [15, 27]. 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 [6, 15]; consequently, today’s state-of-the-art protocols can scale to a throughput of 80,000 requests/s and incur overheads of less than $10 \mu\text{s}$ per request for reaching *agreement* [15]. In contrast, request *execution* overheads for typical applications such as web servers and databases [25] 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 [27]. ZZ targets shared hosting data center environments where replicas from multiple applications can share a physical server. The key insight in ZZ¹ is to run only $f + 1$ execution replicas per application in the graceful case where there are no faults, and to activate additional sleeping replicas 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 approach requires fewer replicas in total, yielding response time and throughput benefits. Like [27], our system still requires $3f + 1$ agreement replicas; however, we argue that 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 em-

¹Denotes sleeping replicas; from the sleeping connotation of the term “zz.”

	PBFT'99	SEP'03	Zyzyva'07	ZZ
Agreement replicas	$3f + 1$	$3f + 1$	$3f + 1$	$3f + 1$
Execution replicas	$3f + 1$	$2f + 1$	$2f + 1$	$(1+r)f+1$
Agreement MACs/req per replica	$2 + \frac{8f+1}{b}$	$2 + \frac{12f+3}{b}$	$2 + \frac{3f}{b}$	$2 + \frac{10f+3}{b}$
Minimum work/req (for large b)	$(3f+1) \cdot (E + 2\mu)$	$(2f+1)E + (3f+1)2\mu$	$(2f+1)E + (3f+1)2\mu$	$(f+1)E + (3f+1)2\mu$
Maximum throughput (if $E \gg \mu$)	$\frac{1}{(3f+1)E}$	$\frac{1}{(2f+1)E}$	$\frac{1}{(2f+1)E}$	$\frac{1}{(f+1)E}$

Table 1. ZZ versus existing BFT approaches. Here, f is the number of allowed faults, b is the batch size, E is execution cost, μ is the cost of a MAC operation, and $r \ll 1$ is a variable formally defined in §4.3.3. All numbers are for periods when there are no faults and the network is well-behaved.

ployed in ZZ, in this paper, we rely upon virtualization—a technique already employed in modern data centers—for on-demand replica activation.

The following are our contributions. We propose a practical solution to reduce the cost of BFT to nearly $f + 1$ execution replicas and define formal bounds on ZZ’s replication cost. As reducing the execution cost in ZZ comes at the expense of potentially allowing faulty nodes to increase response times, we analyze and bound this response time inflation and show that in realistic scenarios malicious applications cannot significantly reduce performance. We also implement a prototype of ZZ by enhancing the BASE library and combining it with the Xen virtual machine and the ZFS file system. ZZ leverages virtualization for fast replica activation and optimizes the recovery protocol to allow newly-activated replicas to immediately begin processing requests through an amortized state transfer strategy. We evaluate our prototype using a BFT web server and ZZ-based NFS file server. Experimental results demonstrate that 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 still enables rapid recovery after simultaneous failures occur. Overall, our evaluation emphasizes the importance of minimizing the execution cost of real BFT services and demonstrates how ZZ provides strong fault tolerance guarantees at significantly lower cost compared to existing systems.

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

In this section, we compare ZZ to state-of-the-art approaches and describe how we reduce the execution cost to $f + 1$.

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 correspond-

ing to Q , execute it in that order, and send responses back to the client. When the client receives $f + 1$ valid and matching responses from different replicas, it knows that at least one correct replica executed Q in the correct order. Figure 1(a) 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 [27], 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 [15, 6, 1] for BFT systems. 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 compares the replication cost and performance characteristics of several BFT State Machine Replication (BFT-SMR) approaches to ZZ. Quorum based approaches [6, 1] lead to a similar comparison. 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, 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 [10].

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.

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 Zyzyva 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 [25]. Thus in realistic systems, the primary

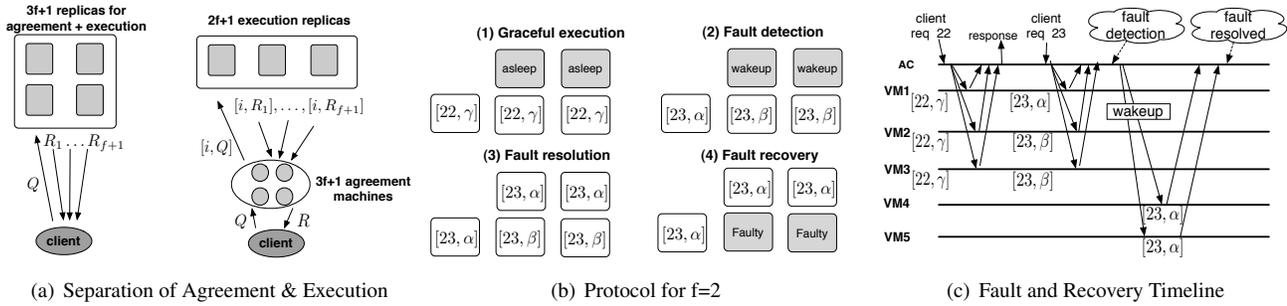


Figure 1. (a) The PBFT approach versus the separation of agreement from execution. (b-c) Various scenarios in the ZZ system for $f = 2$ faults. Request 22 results in matching responses γ , but the mismatch in request 23 initiates new virtual machine replicas on demand.

cost is that of hardware performing application execution, not agreement. ZZ nearly halves the data center provisioning cost by reducing the number of replicas actively executing requests (Table 1 row 2); however, this benefit can only be realized when BFT is used in a data center running multiple applications so that sleeping replicas can be distributed across a pool of servers.

Throughput: ZZ can achieve a higher peak throughput compared to state-of-the-art approaches when execution dominates request processing cost and resources are constrained. For a fair comparison, assume that all approaches are provisioned with the same total amount of resources. Then, the peak throughput of each approach is bounded by the minimum of its best-case execution throughput and its best-case agreement throughput (row 4). Agreement throughput is primarily limited by the overhead μ of a MAC operation and can be improved significantly through batching. However, batching is immaterial to the overall throughput when execution is the bottleneck (row 5).

The comparison above is for performance during periods when there are no faults and the network is well-behaved. In adverse conditions, the throughput and latency of all approaches can degrade significantly and a thorough comparison is nontrivial and difficult to characterize concisely [23, 5].

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 ZZ’s execution approach with a reasonable agreement protocol. Admittedly, it was easier to work out the details of augmenting PBFT with ZZ compared

to more sophisticated agreement protocols.

3 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 4.3.4. An adversary may coordinate the actions of faulty nodes in an arbitrary manner. However, the adversary can not subvert standard cryptographic assumptions about collision-resistant hashes, encryption, and digital signatures.

ZZ uses the state machine replication 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].

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, 15, 27, 22].

Virtualization: ZZ assumes that replicas are 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

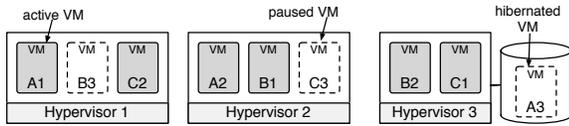


Figure 2. An example server setup with three $f = 1$ fault tolerant applications, A, B, and C; only execution replicas are shown.

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 $f + 1$ 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 replicas are out of date. 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 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 1(b-c) 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. Figure 2 illustrates how ZZ can store additional replicas both in memory as prespawmed but paused VMs and hibernated to disk. Paused VMs resume within milliseconds but consume memory resources. Hibernated replicas require only storage resources, but can incur greater recovery times.

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 create application checkpoints that can be used to bring up to speed any replicas which become out of date. However, a restored ZZ replica may not have any previous version of application state. It must be able to verify that the state it obtains 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 possesses 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 could be worse than a traditional BFT system using $2f + 1$ replicas.

4 ZZ Protocol

In this section we briefly describe the separated protocol from [27], and present ZZ’s modifications to support switching from $f + 1$ to $2f + 1$ execution replicas after faults are detected.

4.1 Graceful Execution

Client Request & Agreement: In Figure 3 step 1, a client c sends a request Q to the agreement cluster to submit an operation o with a timestamp t . The timestamps ensure exactly-once semantics for execution of client requests, and a faulty client’s behavior does not affect other clients’ requests.

Upon receiving a client request Q , the agreement replicas will execute the standard three phase BFT agreement protocol [3] in order to assign a sequence number n to the request. When an agreement replica j learns of the sequence number n committed to Q , it sends a commit message C to all execution replicas (Fig. 3 step 2).

Execution: An execution replica i executes a request Q when it gathers a commit certificate $\{C_i\}$, $i \in A \mid 2g + 1$, 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. Each execution node produces a

reply R which it sends to the client and an execution report message ER sent to all agreement nodes (Fig. 3 steps 3-4).

In the normal case, the client receives a response certificate $\{R_i\}, i \in E|f+1$ —matching reply messages from $f+1$ execution nodes. Since at most f execution replicas can be faulty, a client receiving a response certificate knows that the response is correct. If a client does not obtain matching replies, it resends its request to the agreement cluster. If an agreement node receives a retransmitted request for which it has received $f+1$ matching execution report messages, then it can send a reply affirmation, RA to the client (Fig. 3 step 5). If a client receives $g+1$ such messages containing a response digest, \bar{R} , matching one of the replies already received, then the client can accept that reply as valid. This “backup” solution is used by ZZ to prevent unnecessary wakeups where a partially faulty execution node may reply to the agreement cluster, but not to the client. If the agreement cluster cannot produce an affirmation for the client, then additional nodes must be started as described in subsequent sections.

4.2 Dealing with Faults

4.2.1 Checkpointing

Checkpoints are used so that newly started execution replicas can obtain a recent copy of the application state and so that replicas can periodically garbage collect their logs. The checkpoints are constructed at predetermined request sequence numbers, e.g., when it is exactly divisible by 1024.

With at least $2f+1$ execution replicas in other BFT-SMR systems, a recovering replica is guaranteed to get at least $f+1$ valid and matching checkpoint messages from other execution replicas, allowing checkpoint creation and validation to be done exclusively within the execution cluster [27]. However, ZZ runs only $f+1$ execution replicas in the normal case, and thus a new replica may not be able to tell which of the checkpoints it is provided are correct.

To address this problem, ZZ 's execution cluster must coordinate with the agreement cluster during checkpoint creation. ZZ execution nodes create checkpoints of application state and their reply log, then assemble a proof of their checkpoint, CP , and send it to all of the execution *and* agreement nodes (Fig. 3 step 6). Informing the agreement nodes of the checkpoint digest allows them to assist recovering execution replicas in verifying the checkpoint data they obtain from potentially faulty nodes.

As in most BFT-SMR systems, ZZ uses a copy-on-write technique to preserve the state of each object when checkpoints must be made. All correct execution replicas will save a copy of each object as well as a cryptographic digest of the object's contents. Digests only need to be computed for objects modified since the last checkpoint; this can be done at the time of checkpoint creation, or proactively after an object is modified in order to decrease the checkpointing latency.

A checkpoint certificate $\{CP_i\}, i \in E|f+1$ is a set of $f+1$ CP messages with matching digests. When an execution

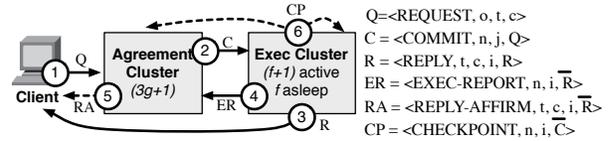


Figure 3. The normal agreement and execution protocol in ZZ proceeds through steps 1-4. Step 5 is needed only after a fault. Checkpoints (step 6) are created on a periodic basis. The notation $\langle \text{LABEL}, X \rangle$ denotes the message of type LABEL with parameters X . We indicate the digest of parameter Y as \bar{Y} .

replica receives a checkpoint certificate with a sequence number n , it considers the checkpoint stable and discards earlier checkpoints and request commit certificates with lower sequence numbers that it received from the agreement cluster. Likewise, the agreement nodes use these checkpoint certificates to determine when they can garbage collect messages in their communication logs with the execution cluster.

Newly awoken replicas in ZZ may not yet have the full application state at checkpoint time, but this does not prevent them from continuing to process requests, since out-of-date replicas are permitted to skip checkpoints. ZZ 's window of vulnerability, defined in Section 4.3.4, assumes that additional faults will not occur until all execution replicas have obtained the full state and are able to create a complete, stable checkpoint.

4.2.2 Fault Detection

The agreement cluster is responsible for detecting faults in the execution cluster. Agreement nodes in ZZ are capable of detecting invalid execution or checkpoint messages; the fault detection and recovery steps for each of these are identical, so for brevity we focus on invalid or missing execution responses. In the normal case, an agreement replica j waits for an execution certificate, $\{ER_i\}, i \in E|f+1$, from the execution cluster. Replica j inserts this certificate into a local log ordered by the sequence number of requests. When j receives ER messages which do not match, or waits for longer than a predetermined timeout, j sends a *recovery request*, $W = \langle \text{RECOVER}, j, n \rangle_j$, to the f hypervisors controlling the standby execution replicas. When the hypervisor of a sleeping execution replica receives a recovery certificate, $\{W_i\}, i \in A|g+1$, it wakes up the local execution replica.

4.2.3 Replica Recovery with Amortized State Transfer

When an execution replica k starts up, it 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. Worse, unlike previous BFT systems that can leverage incremental cryptography schemes to transfer only the objects modified since the last checkpoint, a recovering ZZ replica has no previous checkpoints.

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. Replica k first fetches a log of committed requests since the last checkpoint from the agreement cluster and a checkpoint certificate $\{CP_i\}, i \in A|g+1$ from $g+1$ agreement replicas. This checkpoint certificate includes digests for each state object, allowing the replica to verify that any state object it obtains has come from a correct replica.

After obtaining the checkpoint certificate with object digests, replica k begins to execute in order the recently committed requests. 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 learned from the certificate. Replica k continues executing requests in sequence number order fetching new objects on demand until it obtains a stable checkpoint.

The recovery time can be optimized by only replaying requests which cause writes to state. Note that since on-demand transfers only fetch objects touched by requests, they are not sufficient for k to obtain a stable checkpoint, so the replica must also fetch the remaining state in the background. Recovery is complete only when replica k has obtained a stable checkpoint, although it will be able to correctly respond to replicas as soon as it obtains the subset of the application state needed to process the request that triggered the fault.

4.3 System Properties

We formally define the performance, replication cost, safety and liveness properties of ZZ. Due to space constraints we defer complete proofs to the appendix.

4.3.1 Response Time Inflation

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 procedure for setting timeouts to limit response time inflation. 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 and wakes up f additional replicas.

This procedure trivially bounds the response time inflation of requests to a factor of K , but we can further constrain the

performance impact by considering the response time distribution as follows. Given p , the probability of a replica being faulty,

THEOREM 1 *Faulty replicas can inflate average response time by a factor of: $\max\left(1, \sum_{0 \leq m \leq f} P(m)I(m)\right)$ where:*

$$\begin{aligned} P(m) &= \binom{f}{m} p^m (1-p)^{f-m} \\ I(0) &= 1, \text{ else:} \\ I(m) &= \max\left(1, \frac{K \cdot E[\text{MIN}_{f+1-m}]}{E[\text{MAX}_{f+1}]}\right) \end{aligned}$$

$P(m)$ represents the probability of m simultaneous failures and $I(m)$ is the response time inflation that m faulty nodes can inflict. To get the total impact of response time inflation, we must sum this product for all possible values of m . $E[\text{MIN}_{f+1-m}]$ is the expected minimum response time for a set of $f+1-m$ replicas and $E[\text{MAX}_{f+1}]$ is the expected maximum response time of all $f+1$ replicas, assuming all response times are identically distributed as some distribution Ψ . The top term in $I(m)$ follows from the rule defined above: a faulty node can increase response time by at most K compared to the fastest correct replica (i.e. the replica with the minimum response time out of $f+1-m$ nodes). The bottom term is the non-faulty case where response time is limited by the slowest of the $f+1$ replicas.

As an example, suppose $K=4$, $f=3$, and response times are exponentially distributed with $E[\Psi]=2ms$. Then $E[\text{MIN}_{f+1-m}] = \frac{2}{3+1-m}ms$ and $E[\text{MAX}_{f+1}] = 4.2ms$. If $p=0.1$, then $I=1.0009$, i.e., average response time rises by only 0.09%. Only for $p > 0.48$ is the inflation greater than 10%. Note that proactive recovery can be used to ensure p remains small [4] and that to achieve this worst case bound faulty nodes must be able to predict the earliest response time of correct replicas. In practice, correct execution replicas may sometimes violate the variance bound due to benign execution or network delays, causing a *false timeout*. These false timeouts can impact overall replication cost as described in section 4.3.3.

4.3.2 Waking Up and Shutting Down

Since waking up nodes to respond to faults is an expensive procedure, ZZ distinguishes between “blocking” and “non-blocking” faults, and only triggers a wakeup event for blocking faults—those which cannot be resolved without a wakeup. Fortunately, blocking faults by definition are more widespread in their impact, and thus can always be traced back to a faulty execution node which can then be shutdown.

THEOREM 2 *If a wakeup occurs, ZZ will be able to terminate at least one faulty replica.*

This theorem is proved in the appendix, and is based on the following wake up rule.

Wakeup Rule: A wakeup happens if and only if a mismatch report is “blocking”.

To understand the difference between blocking and non-blocking faults, consider the response matrix where position (i, j) indicates E_i 's response as reported by agreement node A_j . Consider two examples where the client receives conflicting responses P and Q , and $f = g = 1$,

Non-blocking fault	Blocking fault
$A_1 A_2 A_3 A_4$	$A_1 A_2 A_3 A_4$
$E_1 : Q P P P$	$E_1 : Q P P P$
$E_2 : Q P P P$	$E_2 : Q Q Q P$

In the first scenario, it is impossible to distinguish whether only A_1 is faulty or if an execution replica and A_1 is faulty; however, $g + 1$ agreement nodes can provide a reply affirmation that P is the correct response. In the second case, there is no way to tell whether Q or P is the correct response, so a wakeup is required. Once this replica is started, ZZ will be able to determine which replicas were faulty so it can terminate them and reduce the number of active replicas back to only $f + 1$. To do this, ZZ employs the following rules:

Shutdown Rule: If any replicas can be convicted as faulty, shut down f replicas starting with all convictably faulty replicas, and followed by additional replicas that were just woken up if needed. If no replicas can be convicted, delay the shutdown procedure until all replicas produce a stable checkpoint. Then shut down any f of the original replicas.

Note that in most cases a blocking fault will allow ZZ to convict at least one faulty replica causing an immediate shutdown; however, in certain cases where multiple faulty execution and agreement nodes collude, it may not be possible to determine which nodes are faulty. ZZ prefers to shut down replicas immediately after a wakeup because this prevents malicious nodes from reducing ZZ's performance benefits that rely on running only $f + 1$ active replicas in the normal case. We define how ZZ is able to convict faulty replicas in the appendix, and describe how these rules ensure ZZ's safety and liveness properties in Section 4.3.4.

4.3.3 Overall Replication Cost

The expected replication cost of ZZ varies from $f + 1$ to $2f + 1$ depending on the probability of replicas being faulty p , and the likelihood of false timeouts, Π_1 .

THEOREM 3 *The expected replication cost of ZZ is less than $(1 + r)f + 1$, where $r = 1 - (1 - p)^{f+1} + (1 - p)^{f+1}\Pi_1$.*

These two factors influence the replication cost because additional nodes are started only if 1) a replica is truly faulty (which happens with probability $1 - (1 - p)^{f+1}$), or 2) there are no faults, but a correct slow replica causes a false timeout (which happens with probability $(1 - p)^{f+1}\Pi_1$). In either case, the replication cost is increased by f , resulting in the theorem. The value of p can be reduced by proactive recovery, and Π_1 is dependent on the value of K . Adjusting K results in a tradeoff between the replication cost and the re-

sponse time inflation bound. Note that in practice the replication cost may be even lower than this because ZZ will quickly shutdown nodes after the fault has been resolved.

4.3.4 Safety and Liveness Properties

We state the safety and liveness properties ensured by ZZ and outline the proofs.

Safety: ZZ ensures the safety property that if a correct client obtains either a response certificate or an affirmation certificate for a response $\langle \text{REPLY}, t, c, j, R \rangle_j$, 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.

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 that ensures that no two requests are assigned the same sequence number [4]. To show the third claim, we must prove that if a client receives a reply in ZZ, that the reply must be the one produced by a correct execution replica. We consider two cases, depending on whether a wakeup is required to process request n . If a wakeup is required, we must show that after the fault is resolved, the system produces equivalent behavior as if no wakeup had been needed.

In the first scenario a client receives either $f + 1$ matching execution reports or a single execution report that matches $g + 1$ reply affirmations from the agreement cluster without requiring wakeups. The client is assured that reply R is the correct response because matching responses must have been produced by $f + 1$ different execution nodes; since at least one of those nodes is correct, the reply must be correct.

In the second case a client does not immediately receive matching execution reports or reply affirmations. We show that the client will retransmit its requests, and correct agreement replicas will wakeup an additional f execution replicas using the following Lemma in the appendix:

Lemma 1: A client will receive an affirmation certificate unless a mismatch is blocking.

This lemma allows us to guarantee that if clients are not able to directly obtain an affirmation certificate, then a blocking mismatch must have occurred, which in turn will cause an additional f replicas to be started.

The new replicas will be able to obtain a correct snapshot since that only requires a single correct execution replica with application state. The new replicas will now be equivalent to any non-faulty replicas at the start of the last checkpoint epoch, and will correctly replay any requests up to n . Of the $2f + 1$ replicas now active, at most f can be faulty, leaving $f + 1$ responses made by correct replicas. These responses must be identical and will be used to produce the execution report needed by the client for a valid response.

This guarantees that after a *wakeup* the system behavior

is equivalent to that of a correct replica. However, we still must show that once f replicas are *shutdown* that the system will continue to function properly even though newly awoken replicas may not yet have the full application state. In order to maintain the safety property, at least one of the “old” replicas with full state must remain active and not be shutdown. We ensure this with the following lemma proved in the appendix:

Lemma 2: If a mismatch is blocking, then (a) at least one faulty replica can be shutdown and (b) the system will be able to make a stable checkpoint.

If the blocking mismatch results in at least one convictably faulty replica, then ZZ will shut down that node and up to f new replicas. This directly satisfies (a), and part (b) is guaranteed because this procedure will leave at least one correct original node active that will be able to provide the full application state and create a stable checkpoint.

If no nodes can be convicted as faulty, then ZZ will not shut down any nodes until a new stable checkpoint is created, fulfilling requirement (b). Once the checkpoint is made, ZZ will shut down f of the original replicas; since unconvictable faults can only occur if more than one node is faulty, this will eliminate at least one faulty node.

At this point, ZZ has ensured that a correct response has been returned to the client and that at least one active execution replica contains the full application state. This is equivalent to a correctly operating system, ensuring ZZ’s safety properties.

Liveness: ZZ ensures the liveness property that if a correct client sends a request R with a timestamp exceeding previous requests and repeatedly retransmits the request, then it will eventually receive a response certificate or an affirmation certificate for R . We need eventual synchrony to show this liveness property. If the client repeatedly retransmits R , then the agreement cluster will eventually commit a sequence number to R . A correct execution replica will receive a commit certificate, i.e., messages from at least $g + 1$ agreement replicas assigning a common sequence number n to R , by the following property of the agreement protocol: Any request that commits locally at a correct agreement replica will eventually commit locally at, at least, $g + 1$ correct replicas. These properties are guaranteed by the agreement protocol used by ZZ [4].

The existence of at least one correct execution replica ensures that the client gets at least one valid (but yet uncertified) response for R . The agreement cluster in ZZ ensures that it either obtains an execution certificate for R or wakes up the full $2f + 1$ execution replicas. In either case, the agreement cluster will eventually obtain an execution certificate, ensuring that the client eventually obtains an affirmation certificate for R .

Window of Vulnerability: ZZ’s current implementation assumes a window of vulnerability equal to the time needed to detect a fault, wakeup additional replicas, and finish the state transfer so that at least one new replica can create a stable checkpoint. Since at least one faulty replica is detected and eliminated after each failure the system periodically “re-

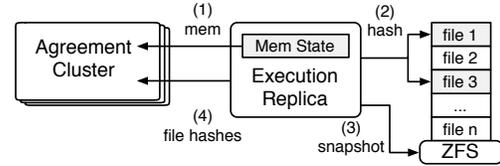


Figure 4. For each checkpoint an execution replica (1) sends any modified memory state, (2) creates hashes for any modified disk files, (3) creates a ZFS snapshot, and (4) returns the list of hashes to agreement nodes.

freshes” itself. However, since there might only be one correct replica with the full application state, this replica cannot become faulty until it finishes transferring the full state to another replica. This is a similar requirement to other BFT-SMR systems which generally rely on proactive recovery (where a replica must be restarted and recover the full application state) in order to reduce the window of vulnerability from infinity.

5 ZZ Implementation

We implemented ZZ by enhancing the 2007 version of BASE [22] so as to 1) use virtual machines to 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 managing replicas after faults occur. The control daemon, which runs in Xen’s Domain-0, uses the certificate scheme described in Section 4.2.2 to ensure that it only starts or stops replicas when enough non-faulty replicas agree that it should do so.

Inactive replicas are maintained either in 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. 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 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

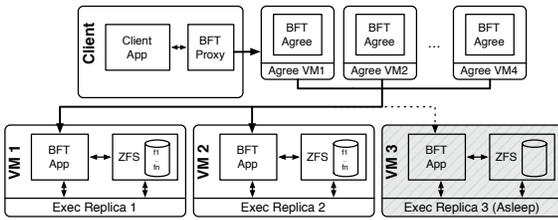


Figure 5: Experimental setup for a basic ZZ BFT service.

execution nodes. In addition, ZZ supports using the snapshot mechanism provided by modern journaled file systems [28] to simplify checkpointing disk state. 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 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 a file 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.

	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	123	124	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.

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.

6.1.1 Throughput

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 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 its 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 [27]. ZZ also requires $3f + 1$ agreement replicas, but extends SEP to use only $f + 1$ active execution replicas, with an additional f sleeping replicas. While more recent agreement protocols provide higher performance than BASE, our evaluation fo-

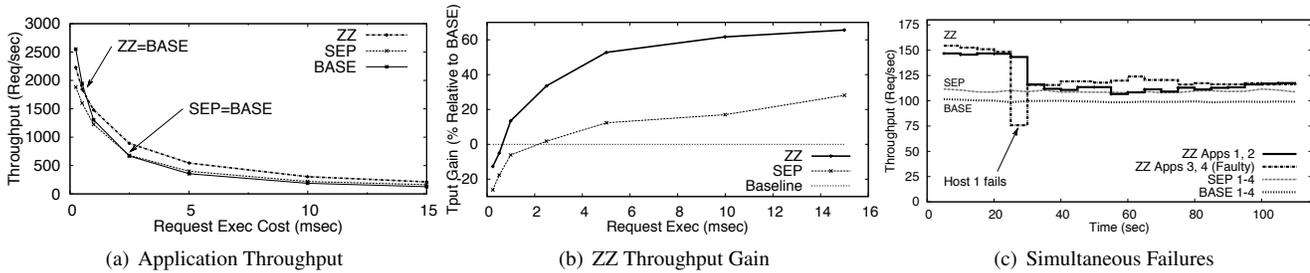


Figure 6. (a-b) When resources are constrained, ZZ significantly increases system throughput by using fewer replicas. (c) Under simultaneous failures of several applications, ZZ quickly recovers and still maintains good throughput.

causes on cases where execution is at least an order of magnitude more expensive than agreement; we believe our conclusions are consistent with what would be found with more optimized agreement protocols.

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. As the agreement and execution clusters can independently handle faults, each host can have at most one replica of each type per application.

We first analyze the impact of request execution cost under ZZ, SEP, and BASE, which require $f + 1$, $2f + 1$, 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 $100 \mu\text{s}$, 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 2 ms , 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.

6.2.1 Latency

We further characterize 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 requests. For null requests or very low loads, Figure 7(a), BASE beats SEP and ZZ because it has less agreement overhead. At 1 ms , 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 10 ms , SEP outperforms BASE since it uses $2f + 1$

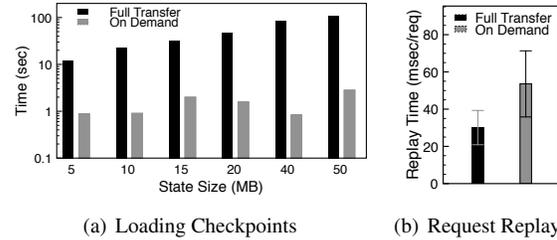


Figure 8. (a) The cost of full state transfer increases with state size. (b) On Demand incurs overhead when replaying requests since state objects must be verified.

instead of $3f + 1$ replicas. ZZ provides a 50% 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 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 6(c). 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.

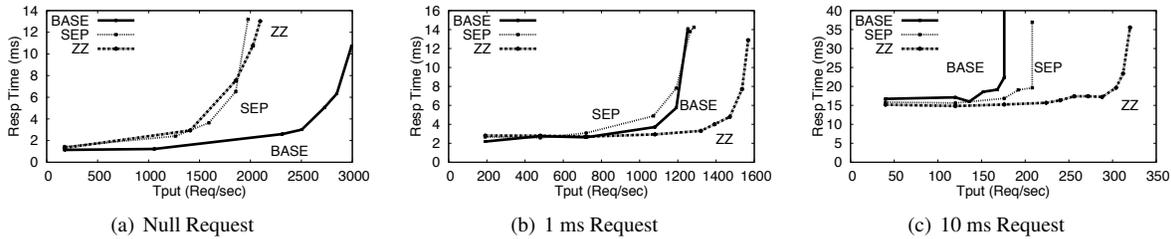


Figure 7: For high execution costs, ZZ achieves both higher throughput and lower response times.

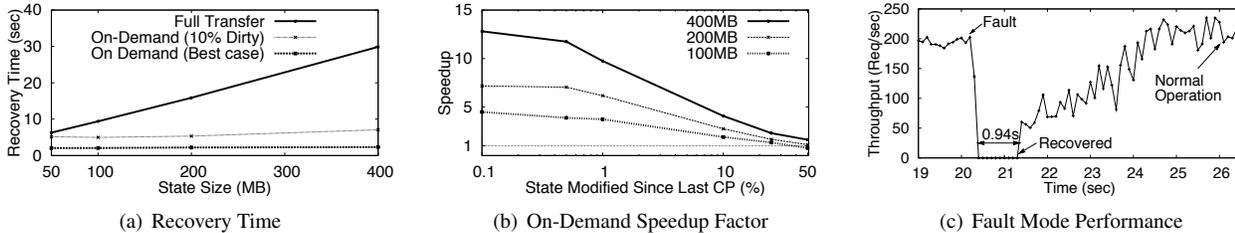


Figure 9. (a-b) The worst case recovery time depends on the amount of state updated between the last checkpoint and the fault. (c) The recovery period lasts for less than a second. At first, requests see higher latency since state must be fetched on-demand.

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. 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 8(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 8(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 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, a fault occurs immediately after a checkpoint and new replicas only need to load and resume from the last save, taking a constant time of about 2s regardless of state size (Figure 9(a)). Otherwise, 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 9(b). Even when 10% of application state is modified between each checkpoint, on-demand transfers speed up recovery by at least five times. Only when more than 50% of state is dirty does it become 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.

Next we examine the impact of on-demand recovery on throughput and latency. The client sends a series of requests involving random accesses to 100kB state objects and a fault is injected after 20.2s (Figure 9(c)). The faulty request experiences a sub-second recovery period, after which the applica-

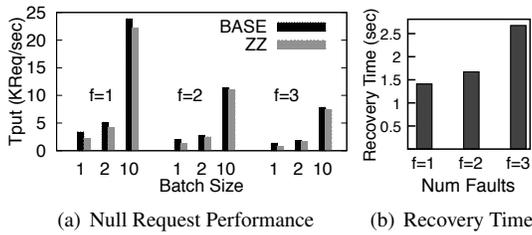


Figure 10. Recovery time increases for larger f from message overhead and increased ZFS operations.

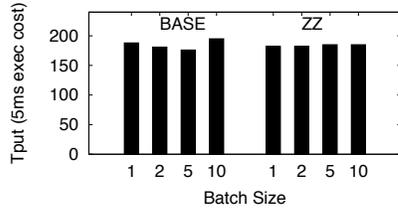


Figure 11. Throughput of ZZ and BASE with different batch sizes for a 5ms request execution time.

tion can handle new requests. The mean request latency prior to the fault is 5ms 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 20ms. As the recovery replica rebuilds its local state, the throughput rises since the proportion of slow requests decreases. After 26s, the full application state has been loaded by the recovery replica, and the throughput prior to the fault is once again maintained.

6.4.3 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 10(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.

We next examine the recovery latency of the client-server microbenchmark for up to three faults. We inject a fault to f of the active execution replicas and measure the recovery time for f new replicas to handle the faulty request. Figure 10(b) shows how the recovery time 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 Trade-offs and Discussion

6.5.1 Agreement Protocol Performance

Various agreement protocol optimizations exist such as request batching, but these may have less effect when request execution costs are non-trivial. While Figure 10(a) shows a large benefit of batching for null requests, Figure 11 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.

6.5.2 Maintaining Spare VMs

In our previous experiments recovery VMs were kept in a paused state which provides a very fast recovery but consumes memory. 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 VMs hibernated to disk can increase recovery latency by a factor proportional to their amount of RAM. 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 reduces 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.

6.5.3 Limitations & Potential for Optimization

ZZ obtains lower hardware costs during graceful performance at the expense of increased delay when a fault occurs. We believe this is a valuable trade-off, and that obtaining state on demand will mitigate the recovery cost in many scenarios. However, there are some applications which contain too large a state or modify it too often for this to be reasonable. Recovering replicas in ZZ must obtain all the state objects relevant for replaying every request that occurred since the last checkpoint. Thus the recovery cost directly depends on the amount of state actively used during each checkpoint period. If the request replay procedure involves reads to a large number of state objects, then those objects will need to be transferred and verified before recovery can finish. One approach to reduce this cost is for ZZ’s sleeping replicas to periodically wakeup and obtain the most recent application state. This approach is used in SPARE to reduce the amount of checkpoint

data that must be obtained when recovering an out of date replica [9]. This form of proactive recovery can be easily employed by ZZ to prefetch state objects that only change across long time scales.

In addition to obtaining the relevant state, the recovering ZZ replicas must perform the computation required to replay every request which involved a write since the last checkpoint. This cost again will depend on the rate at which checkpoints can be made and the execution cost of the write requests. For example, processing the writes in the BFT NFS application studied in Section 6.4.1 are relatively expensive, taking about 30ms per request replayed. If more than a few hundred such requests need to be processed, the recovery time could grow into the tens of seconds. We believe this cost could be reduced by having recovering replicas only replay a subset of the write requests which occurred since the last checkpoint. For example, there may be multiple writes to the same state object; only the most recent such request needs to be replayed. This would reduce both the number of requests that need to be replayed, and possibly the amount of state that needs to be immediately obtained from the last checkpoint. In fact, the replay process could be further optimized by only replaying the missed requests that are “dependencies” for processing the request which produced a fault (or any subsequent requests)—e.g. only those requests that modify a state object that is read by a later request.

7 Related Work

[19] introduced the problem of Byzantine agreement. Lamport also introduced the state machine replication approach [16] 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. [10] 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 [11] result showed that no agreement protocol is guaranteed to terminate with even one (benignly) faulty node in an asynchronous environment. Viewstamped replication [20] and Paxos [17] describe an asynchronous state machine replication approach that is safe despite crash failures.

Early BFT systems [21, 14] 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 [20] or Paxos [17] 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 similar 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 is extending the idea to Byzantine faults and demonstrating its practicality through system design and implementation.

Virtualization has been used in several BFT systems recently since it provides a clean way to isolate services. The SPARE system also uses virtualized replicas to reduce the cost of BFT execution to $f + 1$ replicas [9]. SPARE exploits virtualization to reduce the cost of proactive recovery, allowing it to periodically wakeup passive replicas so they can efficiently catch up to the latest application state. ZZ amortizes recovery cost by obtaining only the necessary state on demand, but its sleeping replicas do not proactively obtain state like those in SPARE. Unlike ZZ, SPARE relies on a trusted component to order requests and does not fully explore the potential for response time inflation from malicious replicas. The cost of BFT execution has also been attacked in [8] where only $f + 1$ replicas process each request when there are no faults. The idea of “reactive recovery”, where faulty replicas are replaced after fault detection, was used in [24], which 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 instantiate new replicas after faults are detected.

The Remus system uses virtualization to provide black-box crash fault tolerance using a standby VM replica [7]. ZZ seeks to provide stronger Byzantine fault tolerance guarantees at a similar replication cost, although ZZ, like all BFT systems, requires application support for the BFT protocol. Terra is a virtual machine platform for trusted computing that employs a trusted hypervisor [12]; 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 $3f + 1$ 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 hosting multiple applications with substantial request execution costs, ZZ can be a practical and cost-effective approach for providing BFT.

Acknowledgements: This work was supported in part by NSF grants CNS-0916972, CNS-0910671, CNS-0834243, and CNS-0720616. We also thank our shepherd, Jacob Lorch, for his many suggestions on improving the paper.

References

- [1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.
- [2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. 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*, Feb. 1999.
- [4] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4), Nov. 2002.
- [5] 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)*, Apr. 2009.
- [6] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. 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, Nov. 2006.
- [7] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *NSDI*, 2008.
- [8] T. Distler and R. Kapitza. Increasing Performance in Byzantine Fault-Tolerant Systems with On-Demand Replica Consistency. In E. C. of ACM SIGOPS, editor, *Proceedings of the EuroSys 2011 Conference (EuroSys '11)*, 2011.
- [9] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on Hold. In I. S. (ISOC), editor, *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS '11)*, 2011.
- [10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), 1988.
- [11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.
- [12] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. 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.
- [13] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 363–376, New York, NY, USA, 2010. ACM.
- [14] K. P. 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*, Washington, DC, USA, 1998.
- [15] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, New York, NY, USA, 2007. ACM.
- [16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
- [17] L. Lamport. Part time parliament. *ACM Transactions on Computer Systems*, 16(2), May 1998.
- [18] L. Lamport and M. Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 307, Washington, DC, USA, 2004. IEEE Computer Society.
- [19] L. Lamport, R. Shostack, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- [20] B. M. Oki and B. H. Liskov. Viewstamped replication: a general primary copy. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles*

of distributed computing, New York, NY, USA, 1988. ACM.

- [21] M. K. Reiter. The rampart toolkit for building high-integrity services. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, London, UK, 1995. Springer-Verlag.
- [22] R. Rodrigues, M. Castro, and B. Liskov. Base: using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001.
- [23] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *NSDI '08: Proceedings of the Usenix Symposium on Networked System Design and Implementation*, 2008.
- [24] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. 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.
- [25] B. Vandiver, H. Balakrishnan, B. Liskov, and S. 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, Oct. 2007.
- [26] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. ZZ and the art of practical BFT. Technical report, University of Massachusetts Amherst, Feb. 2011.
- [27] J. Yin, J. 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*, Oct. 2003.
- [28] The last word in file systems. <http://www.sun.com/2004-0914/feature/>, 2004.

9 Appendix

9.1 Limiting Response Time Inflation

ZZ triggers a fault if replicas respond more than K times slower than the fastest replica. This procedure trivially bounds the response time inflation of requests to a factor of K , but we can further constrain the performance impact by considering the response time distribution as follows. Given p , the probability of a replica being faulty,

THEOREM 1 *Faulty replicas can inflate average response time by a factor of: $\max\left(1, \sum_{0 \leq m \leq f} P(m)I(m)\right)$ where:*

$$P(m) = \binom{f}{m} p^m (1-p)^{f-m}$$

$$I(0) = 1, \text{ else:}$$

$$I(m) = \max\left(1, \frac{K \cdot E[\text{MIN}_{f+1-m}]}{E[\text{MAX}_{f+1}]}\right)$$

$P(m)$ represents the probability of m simultaneous failures and $I(m)$ is the response time inflation that m faulty nodes can inflict. To get the total impact of response time inflation, we must sum this product for all possible values of m . $E[\text{MIN}_{f+1-m}]$ is the expected minimum response time for a set of $f+1-m$ replicas and $E[\text{MAX}_{f+1}]$ is the expected maximum response time of all $f+1$ replicas, assuming all response times are identically distributed as some distribution Ψ . The top term in $I(m)$ follows from the rule defined above: a faulty node can increase response time by at most K compared to the fastest correct replica (i.e. the replica with the minimum response time out of $f+1-m$ nodes). The bottom term is the non-faulty case where response time is limited by the slowest of the $f+1$ replicas.

Proof: During fault-free execution, the response time is given by the random variable MAX_{f+1} , i.e., the time to receive the latest of the $f+1$ responses. Now, if up to $m \leq f$ replicas may be faulty, the response time of the earliest correct replica is given by the random variable MIN_{f+1-m} . The faulty replica(s) can delay the response time up to $K \cdot \text{MIN}_{f+1-m}$ without triggering a timeout. We must sum this fraction over all possible cases, $m = 1, 2, \dots, f$ faults, normalized by the probability of m faults, $P(m)$. ■

In the worst case, faulty replicas can inflate the average response time by up to a factor K , e.g., when $E[\text{MAX}_{f+1}]$ is not much greater than $E[\Psi]$ (because there is little variance in response times from different replicas) and there is only one correct replica, i.e., $m = f$. However, several factors mitigate this inflation in practice. First, if there is little variance in response times, then a small value of K suffices to keep the probability of false timeouts negligibly small. Second, in order to inflate response times without triggering a timeout, faulty replicas need to know the response time of the earliest response from a correct replica, which requires collusion with the agreement cluster. Third, the inflation is only for execution times and does not affect WAN propagation and

transmission delays that typically account for the bulk of the client-perceived or end-to-end delays. Finally, as response times can be inflated only by faulty replicas, proactive recovery can further limit performance degradation under faults by limiting the length of time for which a faulty replica operates undetected [4].

9.2 Wakeup and Shutdown Rules

In this section we prove the following theorem about ZZ's wakeup and shutdown rules.

THEOREM 2 *If a wakeup occurs, ZZ will be able to terminate at least one faulty replica.*

To ensure this theorem, ZZ uses the following rule:

Wakeup Rule: A wakeup happens if and only if a mismatch report is “blocking”.

A mismatch occurs when an agreement replica receives execution replies which are not identical. Suppose that for a particular request there are $g + c$ agreement replicas which experience a mismatch. Consider the *mismatch matrix* of size $(f + 1) * (g + c)$ where entry i, j corresponds to the reply by execution replica E_i as reported by agreement node A_j . Let the execution mismatch of a row be defined as the smallest number of entries that need to be changed in order to make all $g + c$ entries in that row identical. Let the smallest such execution mismatch across all $f + 1$ rows be m . The mismatch is considered blocking if $m < c$.

To understand the difference, consider two examples where the client receives conflicting responses P and Q , and $f = g = 1$,

Full Matrix	Mismatch Matrix	
$A_1 A_2 A_3 A_4$	$A_3 A_4$	
$E_1 : Q Q P Q$	$E_1 : P Q$	$c = 1$
$E_2 : Q Q Q P$	$E_2 : Q P$	$m = 1$

In this scenario, the mismatch matrix has size $g + c = 2$. Since $g = 1$, $c = 1$. Both rows require one entry to be changed in order to create a match, so the minimum mismatch is $m = 1$. Since $m = c$, this is not a blocking fault. The client will be able to receive a reply affirmation that Q is correct from A_1 and A_2 . Note that if an additional node were woken up, it would not be possible to tell which execution node was faulty since it is impossible to tell if A_3 or A_4 is also faulty.

Full Matrix	Mismatch Matrix	
$A_1 A_2 A_3 A_4$	$A_2 A_3$	
$E_1 : Q Q Q P$	$E_1 : Q Q$	$c = 1$
$E_2 : Q P P P$	$E_2 : P P$	$m = 0$

The second scenario illustrates a blocking mismatch. In this case, the *rows* in the mismatch matrix require no changes, thus $m = 0$. Since $m < c$ we have a blocking fault. This makes sense because there is no way to tell whether Q or P is the correct response, so a wakeup is required. To ensure Theorem 2, we also must guarantee:

Lemma 1: A client will receive an affirmation certificate unless a mismatch is blocking.

Since $g + c$ agreement replicas report a mismatch, the number of replicas that have matching replies is $2g + 1 - c$. For a client to get an affirmation certificate, we must show that there are at least $g + 1$ *correct* agreement replicas with matching replies.

As all rows have an execution mismatch of at least m , at least m of the agreement replicas out of the $g + c$ reporting a mismatch must be faulty. To see why this is true, consider that some row in the matrix must correspond to a correct execution replica. If there are no faulty agreement replicas, then all entries in that row should be identical. However, recall that m is the minimum number of entries in any row which would need to be changed to make the whole row identical. This means that even the row for the correct execution replica requires at least m entries to be changed to match. Thus at least m agreement replicas in the mismatch matrix must be lying about the correct replica's response.

Since m agreement replicas that are part of the mismatch matrix are faulty, that means that at most $g - m$ of the remaining $2g + 1 - c$ replicas can be faulty. Therefore, $2g + 1 - c - (g - m) = g + 1 - c + m$ are correct. If the fault is categorized as non-blocking, then $m \geq c$, giving us at least $g + 1$ correct agreement replicas with matching replies. These nodes will be able to provide an execution affirmation to the client without requiring a wakeup, proving the lemma.

Colluding Replicas: In some cases when $f > g$, agreement and execution replicas can collude to cause a blocking fault but hide which execution replicas were incorrect. For example, if $f = 2$ and $g = 1$, this case may arise:

Full Matrix	Mismatch Matrix	
$A_1 A_2 A_3 A_4$	$A_1 A_2 A_3$	
$E_1 : P Q Q Q$	$E_1 : P Q Q$	$f = 2, g = 1$
$E_2 : Q P Q Q$	$E_2 : Q P Q$	$c = 2$
$E_3 : Q Q P Q$	$E_3 : Q Q P$	$m = 1$

Here two execution replicas are faulty, but each one only replies incorrectly to a single agreement node. In addition, one agreement node is faulty and lies about the reply received from the correct execution replica. This is a blocking fault since $m < c$, but when the additional two execution replicas come online it will not be possible to convict any of the older execution replicas as being faulty since they are all indistinguishable. As a result, ZZ will have to delay the shutdown procedure until the new nodes have obtained a stable checkpoint, at which point any of the original three nodes can be terminated. Note that this case can only arise when there can be more faults in the execution cluster than the agreement cluster (i.e., $f > g$). By properly setting f and g , this can be avoided and ZZ will always be able to convict at least one faulty replica immediately after a blocking mismatch.

Lemma 2: If a mismatch is blocking, then (a) at least one faulty replica can be shutdown and (b) the system will still be able to make a stable checkpoint.

The lemma is based on the shutdown rule:

Shutdown Rule: If any replicas can be convicted as faulty, shut down f replicas starting with all convictably faulty replicas, and followed by additional replicas that were just woken

up if needed. If no replicas can be convicted, delay the shutdown procedure until all replicas produce a stable checkpoint. Then shut down any f of the original replicas.

To convict a faulty execution replica we need at least $g + 1$ agreement replicas to concur that the execution node produced an incorrect answer; such an execution replica is *convictably faulty*. Consider the complete $(2f + 1) * (3g + 1)$ response matrix obtained by the replica control daemon after all recovery replicas have produced a reply to the faulty request. Since at most g agreement replicas (columns) and f execution replicas (rows) can be faulty, there must be an $(f + 1) * (2g + 1)$ sub-matrix which contains identical, correct responses. The other entries in the matrix may contain some wrong response. The replica control daemon determines if an execution replica is faulty by looking at its row and checking if there are at least $g + 1$ entries with an incorrect response. Such an execution replica must be faulty because $g + 1$ agreement nodes report it gave an invalid reply, and at least one of those nodes must be correct.

If $f \leq g$, then at least one replica can be convicted as faulty and the shutdown rule trivially proves part (a) of the lemma. Part (b) must also hold because only new replicas or convictably faulty nodes will be terminated, leaving at least one old replica able to create a stable checkpoint.

If no replicas can be convicted due to collusion between the agreement and execution clusters (only possible for $f > g$), then ZZ will not shut down any nodes until the $f + 1$ correct execution replicas create a stable checkpoint, fulfilling part (b). After the checkpoint is made, f of the original replicas will be shutdown; this must include at least one faulty replica since multiple faulty replicas most collude to prevent conviction, satisfying part (a). This guarantees the lemma and proves Theorem 2. ■

9.3 Replication Cost

ZZ provides the following guarantee about its expected replication cost:

THEOREM 3 *The expected replication cost of ZZ is less than $(1 + r)f + 1$, where $r = 1 - (1 - p)^{f+1} + (1 - p)^{f+1}\Pi_1$.*

where p is the probability of a node being faulty and Π_1 is the probability of a correct node causing a false timeout.

Lemma 3: The probability of a false timeout, Π_1 is bounded by:

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

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 $P[\Psi_{(1)} \leq t] \cdot P[\Psi_{(f+1)} > Kt \mid \Psi_{(1)} \leq t]$. 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 $p[\cdot]$ using a small ‘ p ’ denotes the pdf², 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 $p[\Psi_{(1)} \leq t] = \lambda(f + 1)e^{-\lambda(f+1)t}$; and $\Psi_{(f+1)}$ is given by $P[\Psi_{(f+1)} > t] = 1 - (1 - e^{-\lambda t})^{f+1}$.

Given this bound on false timeouts and the probability of a node being faulty p , we can prove Theorem 3 as follows.

Proof: The replication cost is more than $f + 1$ only when a fault is detected either because some replica is faulty or because a false timeout occurs. The probability that at least one replica is faulty is $1 - (1 - p)^{f+1}$. When all replicas are correct, Lemma 1 implies that a false timeout occurs with probability at most Π_1 . Thus, a fault is detected with probability $r \leq 1 - (1 - p)^{f+1} + (1 - p)^{f+1}\Pi_1$. Up to f replicas may be woken up when a fault is detected. Hence the theorem.

²The notation $p_{\Psi_{(1)}}(t)$ is more common, but results in a confusing subscript overkill.