ZZ: Cheap Practical BFT using Virtualization

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

Abstract

Despite numerous efforts to improve their performance and scalability, Byzantine fault-tolerance (BFT) techniques remain expensive, and few commercial systems use BFT today. We present ZZ, a novel approach to construct general BFT services with a replication cost of practically $f + 1$, halving the $2f + 1$ or higher cost incurred by state-of-the-art approaches. The key insight in ZZ is to use $f + 1$ execution replicas in the normal case and to activate additional replicas only upon failures. ZZ uses virtual machines for fast replica activation and several novel mechanisms for rapid recovery of these replicas such as using filesystem snapshots to reduce checkpointing overhead, replaying state updates instead of full requests, and an amortized state transfer mechanism that fetches state on-demand. We have implemented ZZ using the BASE library, Xen virtual machines and the ZFS file system. Our experimental evaluation shows that the recovery time of ZZ replicas is independent of the application disk state, taking less than 4s for 400MB of disk state, at the expense of a small increase in request latency in the fault-mode.

1 Introduction

Today's enterprises rely on data centers—server and storage farms—to run their critical business applications. As users have become increasingly dependent on online services, any malfunctions are highly problematic, resulting in financial losses, negative publicity, or frustrated users. Service malfunctions can be caused by hardware failures, software bugs, or malicious break-in. Consequently, maintaining high availability of critical services has become a pressing need as well as a challenge.

Byzantine fault tolerance (BFT) is a powerful but expensive technique to maintain high availability while tolerating arbitrary (Byzantine) faults. BFT uses replication to mask faults, and BFT state machine replication is a well-studied approach to construct general BFT services. This approach requires replicas to agree upon the order of incoming requests and process every request in the agreed upon order. Despite numerous efforts to improve the performance or fault scalability of BFT systems [2, 5, 12, 21, 26, 1], they expensive today. Existing approaches for BFT require at least $2f + 1$ replicas to execute each request in order to tolerate f faults [12, 30].

Our position is that this high cost has discouraged widespread adoption of BFT techniques for commercial services. On the other hand, data centers today do employ fail-stop fault-tolerance techniques or simple mirroring of data that require only $f + 1$ replicas to tolerate f faults. This suggests that reducing the replication cost of BFT may remove a significant barrier to its adoption.

In this paper, we present a novel approach to construct general BFT services with a replication cost close to $f + 1$, halving the $2f + 1$ or higher cost incurred by state-of-the-art approaches, effectively providing BFT at the cost of fail-stop fault tolerance. Like [30], our system still requires $3f + 1$ agreement replicas, but it only requires $f + 1$ execution replicas in the common case when there are no faults. The key insight in ZZ is to use additional replicas that get activated only upon failures. Furthermore, since not all applications hosted by a data center are likely to experience faults simultaneously, ZZ can share a small pool of free servers across a large number of applications and multiplex these servers on-demand to host newly activated replicas, thereby reducing the total hardware provisioning cost for the data center to practically $f + 1$.

We can circumvent "lower bounds" on replication costs stated in previous work [12], because our approach employs virtualization to enable additional replicas to be quickly activated on demand upon fault detection, and to allow a small number of physical servers to be multiplexed as hosts for the recovery replicas. ZZ employs the following mechanisms to reduce the cost of replication and checkpointing and to enable rapid recovery.

First, ZZ employs virtualization to enable fast replica startup as well as to dynamically multiplex a small number of "auxiliary" servers across a larger number of applications. This minimizes the cost in physical servers when running multiple Byzantine fault tolerant applications by

acknowledging that applications are unlikely to all fail simultaneously. Second, ZZ optimizes the recovery protocol to allow additional newly-activated replicas to immediately begin processing requests through an amortized state transfer mechanism that fetches state on-demand. Third, ZZ exploits the snapshot feature of modern journaled file systems to obtain the benefits of incremental checkpointing without requiring modifications to application source code. This further lowers the barrier of deploying BFT services by reducing the number of modifications required to existing applications.

We have implemented a prototype of ZZ by enhancing the BASE library and combining it with the Xen virtual machine and the ZFS file system. We have evaluated our prototype using a canonical client-server application and a ZZ-based NFS file server. Our experimental results demonstrate that, by verifying state on demand, ZZ can obtain a recovery time that is independent of application state size—a newly activated ZZ replica with 400MB of disk state can recover in less than 4s in contrast to over 60s for existing approaches. ZZ incurs minimal overhead in the fault-free case when utilizing batching, and the recovery speed comes at the expense of only a small increase in request latency in the fault-mode due to the on-demand state verification.

2 Background

2.1 From $3f + 1$ to $2f + 1$

In the traditional PBFT approach [2], during graceful execution a client sends a request Q to the replicas. The $3f + 1$ (or more) replicas agree upon the sequence number corresponding to Q, execute it in that order, and send responses back to the client. When the client receives $f + 1$ valid and matching responses $R_1, \ldots, R_{f+1}$ from different replicas, it knows that at least one correct replica executed Q in the correct order. Figure 1 illustrates how the principle of separating agreement from execution can reduce the number of execution replicas required to tolerate up to f faults from $3f + 1$ to $2f + 1$. In this separation approach [30], 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 State-of-the-art

The 2f+1 replication cost is believed necessary [12, 5, 1] for BFT systems. For example, Kotla et al. in their paper describing Zyzzyva (refer Table 1 in [12]) claim that $2f+1$ is a lower bound on the number of replicas with application state for state machine replication (SMR). However, more than a decade ago, Castro and Liskov concluded their original paper on PBFT [2] 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.

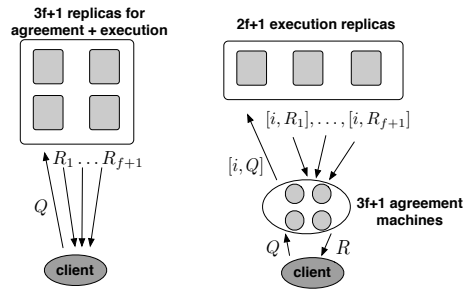


Figure 1: The PBFT approach versus the separation of agreement from execution.

Table 1 shows the cost of existing BFT approaches including non-SMR approaches based on quorums in comparison to ZZ. All approaches require a total of at least $3f + 1$ machines 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 [7].

The quorum-based approach, Q/U, requires an even higher number of replicas but provides better fault scalability compared to PBFT, i.e., better throughput as f increases provided there is low contention across requests. The hybrid quorum (HQ) approach attempts to achieve the best of PBFT and Q/U, i.e., lower replication cost as well as fault scalability, but is unable to leverage the throughput benefits of batching. Zyzzyva, based on the more traditional state machine approach, goes a step further to incorporate speculative execution, resulting in fewer cryptographic operations per request while retaining the throughput benefits of batching. Our implementation of ZZ incurs $2f + 2$ additional cryptographic operations per batch compared to PBFT, but this can be reduced through more efficient piggybacking of protocol messages.

Q/U incurs just two one-way network delays on the critical path when contention is low and is optimal in this respect. All approaches require four or more delays when contention is high or when faults occur. However, the dominant latency in practice is likely to be the wide-area latency from the client to the replicas and back as the replicas in a data center are likely to be located on the same local area network. All approaches except HQ incur just one wide-area network round-trip latency.

The comparison above is for performance of different protocols during fault-free and timeout-free execu-

	PBFT'99 [2]	Q/U'05 [1]	HQ'06 [5]	Zyzzyva'07 [12]	ZZ
Total machines	$3f+1$	$5f+1$	$3f+1$	$3f+1$	$3f+1$
Application state and execution replicas (with separation'03 [30])	$2f+1$	$5f+1$	$3f+1$	$2f+1$	$\approx f+1$
Execution replicas for N apps	$N(2f+1)$	$N(5f+1)$	$N(3f+1)$	$N(2f+1)$	$N(f+1)+N*r*f$
Batching $b>1$ requests	Y	N	N	Y	Y
Critical Path NW 1-way latencies	4	2	4	3	4
Crypto operations per request	$2+(8f+1)/b$	$2+8f$	$4+4f$	$2+3f/b$	$2+(10f+3)/b$
Fault-mode latency ratio	$\approx$1	$\approx$1	≈ 1	≈ 1	>1

Table 1: Properties of various approaches. f is the number of supported faults, r is the ratio of recovery time to mean time to failure (typically $r << 1$), and b is the batch size.

tion. When failures occur, the throughput and latency of all protocols can degrade significantly and a thorough comparison is nontrivial and difficult to characterize concisely [23]. When failures occur, a recovering ZZ[1] replica incurs a higher latency to execute some requests until it has fully recovered. Our experiments suggest that this additional overhead is tolerable. Also, the client can tentatively accept a replica's response while waiting to collect a certificate for the response, and issue other requests that do not depend on this response. In a world where failures are the uncommon case, ZZ offers valuable savings in replication cost while minimally impacting performance.

ZZ is not a new BFT protocol; instead, it is an extension that can be applied to existing BFT-SMR protocols in order to lower the effective cost of running such systems from $2f+1$ to about $f+1$. Our deployment utilizes the BASE implementation of the PBFT protocol since it was the most mature and readily available BFT implementation at the time of writing. In theory, ZZ's execution replicas can be interfaced with other agreement protocols, including Zyzzyva to match its low cryptographic overhead as well as fault scalability.

Unlike previous approaches, ZZ can exploit server multiplexing in order to reduce the total cost when running many applications in a BFT data center. This reduces the expected machine cost for running N applications to $N(f+1)+N*r*f$, a significant reduction compared to other approaches which require $N(2f+1)$ or more replicas, as described in detail in Section 3.4.

3 ZZ design

3.1 System model

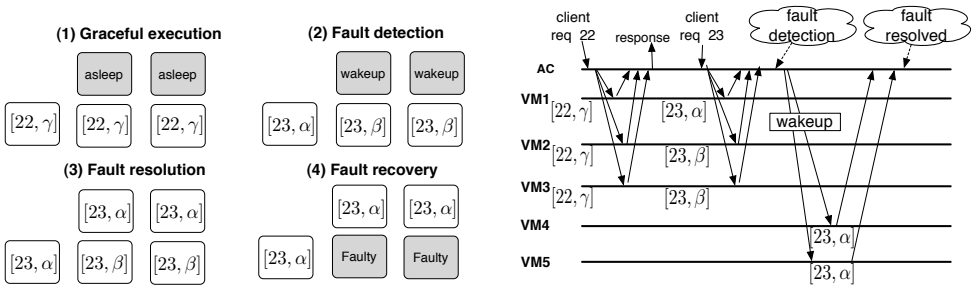
We assume a Byzantine failure model where faulty replicas or clients may behave arbitrarily. There are two kinds of replicas: 1) agreement replicas that assign an order to client requests and 2) execution replicas that maintain application state and execute client requests. Replicas fail independently, and we assume an upper bound g on the number of faulty agreement replicas and an upper 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 4.8. An adversary may coordinate the actions of faulty nodes in an arbitrary malicious manner. However, the adversary can not subvert standard cryptographic assumptions about collision-resistant hashes, encryption, and digital signatures. The notation $\langle \text{LABEL}, X \rangle_i$ denotes the message X of type LABEL digitally signed by node i. We indicate the digest of message X as $\overline{X}$.

Our system uses the state machine replication (SMR) model to implement a BFT service. Replicas agree on an ordering of incoming requests and each execution replica executes all requests in the same order. Like all previous SMR based BFT systems, we assume that either the service is deterministic or the non-deterministic operations in the service can be transformed to deterministic ones via the agreement protocol [2, 12, 30, 21].

Our system ensures safety in an asynchronous network that can drop, delay, corrupt, or reorder messages. Liveness is guaranteed only under eventual synchrony, i.e., there is an unknown point in the execution after which either all messages delivered within some constant time Δ, or all non-faulty clients have received replies to their requests. Our system model and assumptions are similar to those assumed by many existing BFT systems [2, 12, 30, 21].

ZZ assumes replicas are being run within a virtualized data center. As a result, it is possible to run multiple replicas on a single host, a situation not encountered in other BFT systems. To tolerate faults caused by physical server crashes which would bring down all hosted replicas, ZZ requires that no more than one replica per application is hosted on a single physical server. With this requirement, ZZ does not require a trusted hypervisor; if a malicious hypervisor crashes a system or maliciously manipulates application responses, it will only count as a single fault occurring in each of the hosted applications. We utilize a hypervisor controlled service for starting and stopping virtual machines; in the case of a malicious hypervisor, we assume an out of band mechanism exists to power down the physical server.

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



(a) Protocol for f=2

(b) Fault and Recovery Timeline

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

3.2 Design overview

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

Given an ordered request, if the $f+1$ active replicas return matching responses, 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 2 illustrates the high-level flow of control for the case when $f = 2$. Request 22 is executed successfully generating the response γ, but request 23 results in a mismatch waking up the two standby VM replicas.

3.3 Design Challenges

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

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 may stall request execution until f recovering replicas have obtained a stable checkpoint.

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

3.4 Using Virtual Machine Replicas

ZZ assumes an environment such as a data center that runs N independent applications; replicas belonging to each application are run inside separate virtual machines, which are managed by a hypervisor. In the normal case, at least $f+1$ execution replicas are assumed to be running on a mutually exclusive set of servers. A single physical server can house replicas from independent applications; for instance, a server may host multiple agreement or execution replicas, each from a different application. The primary gain in multiplexing comes from maintaining a pool of free (or under-loaded) servers that are used on-demand to host newly activated replicas. Assuming that not all of the N executing applications will see a fault simultaneously, this free server pool can be multiplexed across the N applications, allowing the pool to be much

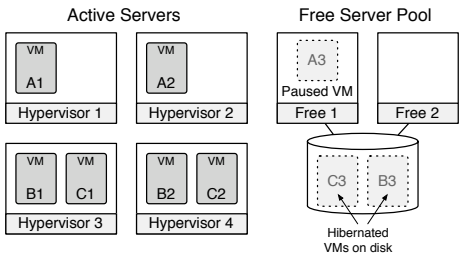


Figure 3: An example server setup with three fault tolerant applications, A, B, and C. The recovery pool stores a mix of paused and hibernated VMs.

smaller than the worst-case needs of $N(f+1)$ additional execution servers.

The number of free servers needed as a "backup" for fault-mode execution depends on r, the ratio of the time for ZZ to recover and replace a faulty replica to the mean time to failure for an application. This ratio is typically much less than one, since the recovery time is on the order of seconds, while mean time to failure can be days or more. In this case, the expected machine cost for running N BFT applications in a ZZ data center is $N(f+1) + N * r * f$, a significant reduction compared to other approaches which require $N(2f+1)$ or more execution replicas. Even for a single application without multiplexing, ZZ provides valuable savings in power and operational costs as each request is executed only $f+1$ times. Finally, if a trusted hypervisor is used, ZZ's virtual replicas can reduce the number of physical servers needed to $f+1$ even for a single application.

Virtualization also enables fast replica startup by maintaining extra replicas in a "dormant" state. Replicas can be stored hibernated to disk, or for faster startup prespawned VMs can be kept in pause mode in memory, where it consumes no CPU and uses limited memory; unpausing a VM simply requires scheduling it on the CPU, causing it to become active within milliseconds. The ZZ implementation uses the Xen platform and supports a mix, allowing for some (e.g., important) VMs to be paused and the rest to hibernate (see Figure 3).

4 ZZ Protocol

This section presents the ZZ protocol for (i) handling incoming client requests, (ii) graceful execution, (iii) checkpointing, (iv) fault detection, and (v) replica recovery. We also present our amortized state transfer mechanism and discuss safety and liveness properties of ZZ.

4.1 Client Request

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

4.2 Graceful Execution

The agreement cluster 1) assigns an order to incoming client requests, 2) sends committed requests to the execution cluster, 3) receives execution reports back from the execution cluster, and 4) relays response certificates to the client when needed.

Upon receiving a client request $Q = \langle \text{REQUEST}, o, t, c \rangle_c$, the agreement replicas execute an agreement protocol and commit a sequence number to the request. Each agreement replica j sends an order message $\langle \text{ORDER}, v, n, Q \rangle_j$ that includes the view v and the sequence number n to all execution replicas.

An execution replica i executes a request Q when it receives a request certificate $\langle \text{ORDER}, v, n, Q \rangle_{A|2g+1}$ signed by at least $2g+1$ agreement replicas and it has executed all other requests with a lower sequence number. Let R denote the response obtained by executing Q. Replica i sends a message containing the digests of the response and the writes $\langle \text{EXEC-REPORT}, n, \overline{R} \rangle_i$ to the agreement cluster. Replica i also sends a response message $\langle \text{REPLY}, v, t, c, i, R \rangle_i$ directly to the client.

In the normal case, the client receives $f+1$ valid and matching response messages from execution replicas. Two response messages match if they have the same values for v, t, c, and R. The client determines a message from replica i to be valid if it has a valid signature. This collection of $f+1$ valid and matching responses is called a response certificate. Since at most f replicas can be faulty, a client receiving a response certificate knows that the response is correct.

Request Timeout at Client: If a client does not receive a response certificate within a predetermined timeout, it retransmits the request message to all agreement replicas. If an agreement replica j receives such a retransmitted request and it has received $f+1$ matching EXEC-REPORT messages, it sends the response message $\langle \text{REPLY-A}, v, t, c, j, \overline{R} \rangle_j$. A client considers a response R as correct when it receives $g+1$ valid and matching REPLY-A messages from agreement replicas and at least one valid REPLY message from an execution replica.

As in PBFT [2], a client normally sends its request only to the primary agreement replica, which it learns based on previous replies. If many agreement replicas receive retransmissions of a client request for which a valid sequence number has not been assigned, they will eventually suspect the primary to be faulty and trigger a view change electing a new primary. Note that although

clients can cause view changes in ZZ, they can not trigger the recovery of standby execution replicas.

4.3 Checkpointing

Execution replicas periodically construct checkpoints of application state in order to garbage collect their pending request logs. The checkpoints are constructed at predetermined request sequence numbers, e.g., when the sequence number is exactly divisible by 1024.

Unlike [30], the checkpoint protocol is not completely contained within the execution cluster and requires coordination with the agreement cluster. To appreciate why, consider that in existing BFT systems including [30], if digital signatures are used, it suffices for an execution replica to obtain a checkpoint certificate $\langle \text{CHECKPOINT}, n, \overline{C} \rangle_{E|f+1}$ signed by $f+1$ execution replicas. It can use this certificate to prove to a recovering replica during a state transfer that the checkpoint is valid. However, if message authentication codes (MACs) are used, then $f+1$ checkpoint messages are insufficient for a replica to prove to a recovering replica that the checkpoint is correct. With $2f+1$ execution replicas in [30], a recovering replica is guaranteed to get at least $f+1$ valid and matching checkpoint messages from other execution replicas. However, with just $f+1$ execution replicas in ZZ, a recovering replica may get only one checkpoint message from other execution replicas.

To address this problem, the execution cluster coordinates with the agreement cluster during checkpoints. Each execution replica i sends a checkpoint message $\langle \text{CHECKPOINT}, n, \overline{C} \rangle_i$ to the agreement cluster. An agreement replica waits to receive $f+1$ valid, matching checkpoint messages. If $g+1$ or more agreement replicas fail to receive $f+1$ valid and matching checkpoint messages, they issue a recovery request to a subset of the hypervisors controlling the standby execution replicas.

In practice, applications store state in both memory and on disk. While previous approaches considered all types of state equivalently, ZZ differentiates between disk and memory state objects to allow for optimizations during state transfer to recovering replicas. Memory state is fully checkpointed as in the BASE library, and is transferred in full to recovering replicas before they start processing requests. Disk state objects, in contrast, are obtained by recovering replicas on demand using the amortized state transfer procedure described in the following sections. We describe how we can exploit file system level snapshots to produce the disk checkpoints in Section 5.3.

4.4 Fault Detection

The agreement cluster is responsible for detecting faults in the execution cluster. In the normal case, an agreement replica j waits for an execution report certificate $\langle \text{EXEC-REPORT}, n, \overline{R} \rangle_{E|f+1}$ for each request, i.e., valid and matching execution reports for the request from all $f+1$ execution replicas. Replica j inserts this certificate into a local log ordered by the sequence number of requests. When j does not obtain an execution report certificate for a request within a predetermined timeout, j sends a *recovery request* $\langle \text{RECOVER}, n \rangle_j$ to a subset of the hypervisors controlling the f standby execution replicas. The number of recovery requests issued by an agreement replica is at least as large as $f+1$ minus the size of the smallest set of valid and matching execution reports for the request that triggered the recovery.

When the hypervisor controlling an execution replica i receives a recovery certificate $\langle \text{RECOVER}, n \rangle_{A|g+1}$, i.e., valid and matching recovery requests from $g+1$ or more agreement replicas, it starts up the local execution replica.

4.5 Replica Recovery

When an execution replica k starts up, it neither has any application state nor a pending log of requests. Replica k's situation is similar to that of a long-lost replica that may have missed several previous checkpoints and wishes to catch up in existing BFT systems [2, 21, 30, 12]. Since these systems are designed for an asynchronous environment, they can correctly reintegrate up to f such recovering replicas provided there are at least $f+1$ other correct execution replicas. The recovering replica must obtain the most recent checkpoint of the entire application state from existing replicas and verify that it is correct. Unfortunately, checkpoint transfer and verification can take an unacceptably long time for applications with a large application state. Worse, unlike previous BFT systems that can leverage copy-on-write techniques and incremental cryptography schemes to transfer only the objects modified since the last checkpoint, a recovering ZZ replica does not have any previous application state.

To reduce recovery latency, ZZ uses a novel recovery scheme that amortizes the cost of state transfer across many requests. A recovering execution replica k first obtains an ordered log of committed requests since the most recent checkpoint from the agreement cluster. Let m denote the sequence number corresponding to the most recent checkpoint and let $n \geq m+1$ denote the sequence number of the most recent committed request. Some of the requests in the interval $[m+1, n]$ involve writes to application state while others do not. Replica k begins to replay in order the requests in $[m+1, n]$ involving writes to application state.

4.6 Amortized State Transfer

How does replica k begin to execute requests without any application state? The key insight is to fetch and verify the state necessary to execute a request on demand. After fetching the pending log of committed requests from the agreement cluster, k obtains a checkpoint certificate $\langle \text{CHECKPOINT}, n, \overline{C} \rangle_{E|g+1}$ from the agreement cluster.

Replica k further obtains valid digests for each object in the checkpoint from any execution replica. The checkpoint digest $\overline{C}$ is computed over the digests of individual object digests, so k can verify that it has received valid state objects from a correct execution replica. As in previous BFT systems [2, 21, 30, 12], execution replicas optimize the computation of checkpoint and object digests using copy-on-write and incremental cryptography schemes.

After obtaining the above checkpoints, replica k begins to execute in order the committed requests $[m+1, n]$ that involve writes. 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 the object on demand from any execution replica that can provide an object consistent with p's digest that k has already verified. Replica k continues executing requests in sequence number order and fetches objects on demand until it obtains a stable checkpoint.

4.7 Safety and Liveness Properties

We state the safety and liveness properties ensured by ZZ and outline the proofs. Due to space constraints, we defer formal proofs to a technical report [29].

ZZ ensures the safety property that if a client receives a reply $\langle \text{REPLY}, v, n, t, c, i, R \rangle_i$ from $f+1$ execution replicas or a reply from at least one execution replica and $\langle \text{REPLY-A}, v, b, t, c, j, \overline{R} \rangle_j$ from $g+1$ agreement replicas, then 1) the client issued a request $\langle \text{REQUEST}, o, t, c \rangle_c$ earlier, 2) all correct replicas agree on the order of all requests with sequence numbers in $[1, n]$, 3) the value of the reply R is the reply that a single correct replica would have produced if it started with a default initial state S_0 and executed each operation $o_i, 1 \leq i \leq n$, in that order, where o_i denotes the operation requested by the request to which sequence number i was assigned.

The first claim follows from the fact that the agreement cluster only generates valid request certificates for valid client requests and the second follows from the safety of the agreement protocol [3] which ensures that no two requests are assigned the same sequence number. To show the third claim, consider a single correct execution replica e that starts with a default state S_0 and executes all operations $o_i, 1 \leq i \leq n$, in order. We first show that the state S^i_{n-1} of at least one correct active execution replica i after executing o_n is equal to the corresponding state S^e_n of the single correct replica. The proof is an inductive argument over epochs where a new epoch begins each time the agreement cluster detects a fault and starts up standby replicas. Just before a new epoch begins, there are at least $f+1$ replicas with a full copy of the most recent stable checkpoint. The agreement cluster shuts down execution replicas that responded incorrectly. At least one replica that responded is correct as there can be at most f faults with an infinite window of vulnerability.

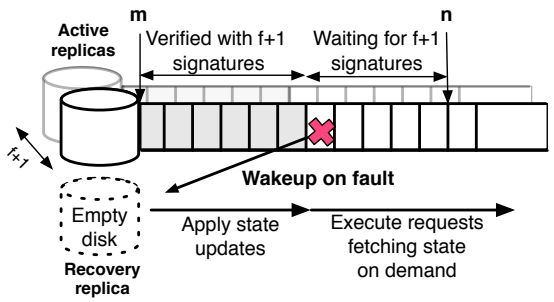


Figure 4: ZZ separates request execution from updating state during recovery.

ZZ ensures the liveness property that if a client sends a request with a timestamp exceeding previous requests and repeatedly retransmits the request, then it will eventually receive a valid reply certificate. Liveness of the agreement protocol follows from existing work [3]. The execution cluster's liveness property relies on the fact that there always must be at least one correct execution replica which will cause the agreement cluster to initialize additional execution replicas if necessary. Once all $2f+1$ execution replicas are active, liveness is guaranteed since the correct replicas will provide a sufficient majority to make progress.

4.8 Optimizations

4.8.1 Reducing the Window of Vulnerability

Our implementation of ZZ currently assumes an infinite window of vulnerability, however, this assumption can be relaxed through the use of proactive recovery [3]. By periodically forcing replicas to recover to a known clean state, proactive recovery allows for a configurable window of vulnerability in which f faults can occur. Efficient proactive recovery is crucial to ZZ because it runs only $f+1$ execution replicas during correct operation, and putting one down for recovery can render the system unavailable. Fortunately, ZZ can perform proactive recovery cheaply since it runs on a virtualization platform and can exploit this to begin proactive recovery of a replica as a new VM running in parallel to an existing replica [19]. Such a technique incurs minimal downtime if the replica being recovered is non-faulty, while the faulty case can be detected and dealt with like an ordinary fault in the system.

4.8.2 Separating State Updates and Execution

A novel optimization enabled by ZZ's recovery approach is the separation of state updates from the rest of request execution to further speed up recovery. A ZZ replica need only apply state updates without actually re-executing requests since the last checkpoint.

To this end, execution replicas include information about state updates in their execution reports.

During graceful execution, an execution replica i includes additional information in its execution report $\langle \text{EXEC-REPORT}, n, \overline{W}, H_n \rangle$ where W identifies the writes performed during the execution of the request and $H_n = [H_{n-1}, \overline{W}]$ summarizes the write history. Let m be the sequence number of the last stable checkpoint and let n be the sequence number that caused a fault to be detected. Replica i also stores the list of writes $W_{m+1}, \ldots, W_{n-1}$ performed while executing requests with the corresponding sequence numbers.

Upon recovery, an execution replica j obtains a checkpoint certificate $\langle \text{CHECKPOINT}, m, \overline{C} \rangle_{A|g+1}$ for the most recent successful checkpoint, and the execution report certificate $\langle \text{EXEC-REPORT}, n, \overline{W}, H_{n-1} \rangle$ for the sequence number immediately preceding the fault. Replica j obtains the list of writes $[W_{m+1}, \ldots, W_{n-1}]$ matching the execution report certificate from an execution replica. Then, instead of actually executing the requests, j simply applies the writes in sequence number order while fetching the most recent checkpoint of the objects being written to on demand. This approach is illustrated in Figure 4. For compute-intensive applications with infrequent and small writes, this optimization can significantly reduce recovery latency.

5 ZZ Implementation

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

5.1 Replica Control Daemon

Xen does not allow arbitrary code to be run within the hypervisor. As a result, we have implemented a replica control daemon which runs in Domain-0, a "privileged" domain responsible for managing the other virtual machines on the host. One replica control daemon runs on each physical host and is responsible for starting and stopping replicas after faults are detected. The control daemon uses the certificate scheme described in Section 4.4 to ensure that it only starts or stops replicas when enough non-faulty replicas agree that it should do so.

The inactive replicas can be maintained in either a paused state where they have no CPU cost, but incur a small memory overhead on the system, or hibernated to disk which utilizes no resources other than disk space. Paused replicas can be very quickly initialized after a fault, while a naive approach to hibernated VMs can take tens of seconds and is proportional to the amount of memory allocated to a VM. ZZ uses a paged-out restore technique that exploits the fact that hibernated replicas initially have no useful application state in memory, and thus can be created with a bare minimum allocation of 128MB of RAM. After being restored, their memory allocation is increased to the desired level. Although the VM will immediately have access to its expanded memory allocation, there may be an application dependent period of reduced performance if data needs to be paged in.

5.2 Recovery

The implementation of ZZ's recovery protocol described in Section 4.5 proceeds in the following steps.
1. Fault Detection: When an agreement replicas receive $f + 1$ output messages from execution replicas that are not all identical, it sends wake-up messages to the replica control daemons on f servers in the free pool.
2. VM Wake-up: When a replica control daemon receives $f+1$ wake-up messages it attempts to "unpause" a replica if available, and if not, spawns a new VM by loading a replica hibernated on disk.
3. Checkpoint metadata transfer: A replica upon startup obtains the log of committed requests, checkpoint metadata, and any memory state corresponding the most recent stable checkpoint. The replica also obtain access to the latest disk snapshots created by all replicas.
4. Replay: The replica replays requests prior to the fault that modified any application state.
5. On-demand verification: The replica attempts to get the state required for each request from another replica's disk snapshot, and verifies the file contents against the disk hashes contained in the checkpoint metadata. Upon retrieving a valid file, the replica copies it to its local disk and directs all future accesses to that version of the file.
6. Eliminate faulty replicas: With $2f + 1$ replies, the agreement cluster determines which of the original execution replicas were faulty. Agreement replicas send shutdown messages to the replica control daemons listing the faulty replicas to be terminated.

5.3 Exploiting File System Snapshots

Typical data center applications utilize a relatively small amount of critical in memory state, plus a potentially much larger amount of disk state. To implement checkpointing in ZZ we rely on the existing mechanisms in the BASE library to save the protocol state of the agreement nodes and any memory state used by the application on the execution nodes. In addition ZZ exploits the snapshot mechanism supported by many modern file systems to make efficient disk checkpoints, lowering memory overheads since duplicate copies do not need to be kept in memory if they are modified between checkpoints and reducing the number of modifications required to make existing applications support BFT.

Some modern file systems allow for very efficient snapshot creation, typically by maintaining a lof of updates in a journal [31, 18]. Creating snapshots is efficient because copy-on-write techniques prevent the need for duplicate disk blocks to be created. Snapshots size and creation time is independent of the size of data stored

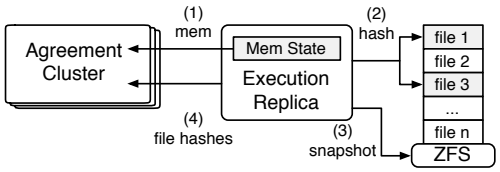


Figure 5: For each checkpoint an execution replica (1) sends any modified memory state, (2) creates hashes for any modified disk files, (3) creates a ZFS snapshot, and (4) returns the list of hashes to the agreement cluster.

since only meta information needs to be updated. ZZ relies on ZFS for snapshot support, and works with both the native Solaris and user-space Linux 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 5. 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 files from the disk snapshots and copy it to the replica's own disk on demand. When a recovering replica first tries to open a file, it calls bft_fopen(foo), but the replica will not yet have a local copy of the file. The recovery replica fetches a copy of the file from any replica and verifies it against the hash contained in the most recent checkpoint. If the hashes do not match, the recovery replica requests the file from a different replica, until a matching copy is found and copied to its own disk.

6 Experimental Evaluation

In this section, we evaluate our ZZ prototype under graceful execution and in the presence of faults. Specifically, we quantify the recovery times seen by a ZZ replica, the efficacy of obtaining state on demand, ZZ's overhead compared to BASE under correct operation, and the costs of using virtualization and ZFS snapshots.

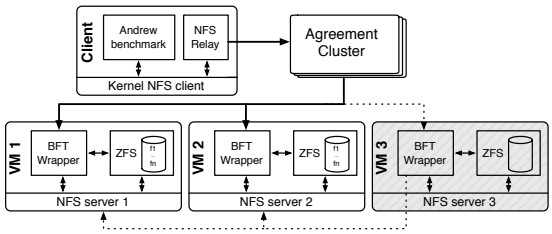


Figure 6: Experimental setup for NFS. The recovery replica obtains state on demand from the active servers.

6.1 Experimental Setup

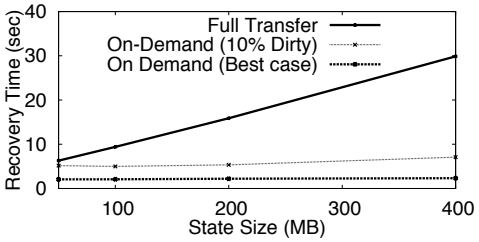
Our experimental setup consists of a mini fault-tolerant data center that uses a cluster of 2.12 GHz 64-bit Xeon quad-core Dell servers, each with 4GB RAM. Each machine runs a Xen v3.1 hypervisor and Xen virtual machines. Both domain-0 (the controller domain in Xen) as well as the individual VMs run the CentOS 5.1 Linux distribution with the 2.6.18 Linux kernel. For ease of deployment, we use the Linux implementation of ZFS which runs as a userspace file system on a separate storage node. Each execution replica is given an independent ZFS file system on the storage node (note that, we place all ZFS file systems on a single node due to constraints on the number of machines at our disposal; a true BFT system would use a separate storage node for each execution replica). All machines are interconnected over gigabit ethernet.

We use $f + 1$ machines to house our execution replicas, each of which runs inside its own virtual machine. An additional f machines are assumed to belong to the free server pool; we experiment with both paused and hibernated pre-spawned execution replicas. The agreement replicas run on a separate set of machines from the execution nodes.

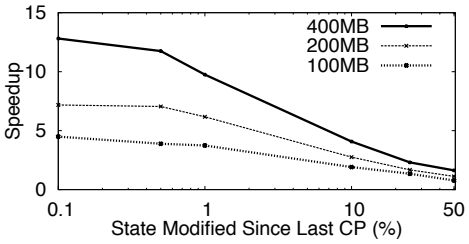
Our experiments involve 1) a toy client-server application and 2) a fault-tolerant NFS, based on ZZ.

Toy client-server: We modified the generic client server application from the BASE library to maintain a configurable amount of state on disk rather than in memory. Recovery replicas can use either ZZ's amortized state transfer technique to fetch state on demand, or a full transfer scheme which fetches and verifies all state immediately after fault detection.

Fault-tolerant NFS: BASE provides an NFS client relay and a BFT wrapper for the standard NFS server. We extendeded this wrapper so that a recovering replica can communicate with multiple NFS servers. Recovery replicas fetche disk state objects by requesting access to ZFS snapshots through the NFS servers of existing replicas. Figure 6 illustrates the system setup for an NFS server capable of handling one faulty replica, where VM3 is an initially sleeping recovery replica. We use a combination of synthetic workloads and the Andrew filesystem benchmark in our experiments.



(a) Recovery Time



(b) On Demand Speedup Factor

Figure 7: The worst case recovery time depends on the amount of state updated between the last checkpoint and the fault.

6.2 Recovery Time

This experiment uses our client-server application to study the recovery time after faults are caused in applications with different state sizes. The disk state at the server is varied between 50MB and 400MB by varying the sizes of 500 files. We define recovery time as the delay from when the agreement cluster detects a fault until the client receives the correct response.

Figure 7(a) compares the recovery time for a replica that performs full state transfer to one that fetches and verifies state on-demand. In the best case, a fault occurs immediately after a checkpoint, meaning that no additional requests need to be replayed by the recovery node. While the naive approach can take over 30 seconds to fully verify 400MB of state, ZZ's on demand scheme can recover and begin processing requests in less than 3s. In practice, a fault is unlikely to occur immediately after a checkpoint, forcing the recovery replica to replay requests and update its local state before it can respond to the client. The "10% Dirty" line in Figure 7(a) shows the recovery cost when the fault occurs immediately before a checkpoint such that 10% of the application's state needs to be fetched during replay. While this increases the recovery latency, ZZ's on-demand transfer scheme still sees significant benefits.

Figure 7(b) shows the speedup gained from getting state on demand as the amount of state objects modified since the last checkpoint increases. Having more modified state objects incurs greater cost since more objects need to be immediately fetched. Fortunately, checkpoints can be efficiently performed every few seconds, during which time most applications will only modify a small fraction of their total application state. For applications

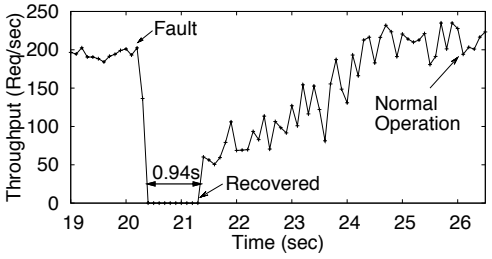


Figure 8: The recovery period lasts for less than a second, after which new requests can be processed. At first, requests see higher latency since state must be fetched on demand.

with large amounts of state, the on demand scheme can decrease the recovery time by at least five times when less than 10% of the state is modified between the checkpoint and a fault.

6.3 Fault Mode Latency

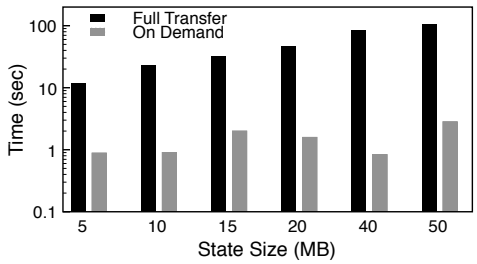
While obtaining state on-demand significantly reduces the initial recovery time, it may increase the latency for subsequent requests since they must fetch and verify state. In this experiment we examine the throughput and latency of requests to the client-server application after a fault has occurred.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

As shown in Figure 8, we inject a fault after 20.2 seconds. The faulty request experiences a latency of about one second for the initial recovery, after which the application can handle new requests. The mean request latency prior to the fault is 5 milliseconds with very little variation. The latency of requests after the fault has a bimodal distribution depending on whether the request accesses a file that has already been fetched or one which needs to be fetched and verified. The long requests, which include state verification and transfer, take an average of 20 milliseconds. As the recovery replica rebuilds its local state, the throughput rises since the proportion of slow requests decreases. After 26 seconds, the full application state has been loaded by the recovery replica, and the throughput prior to the fault is once again maintained.

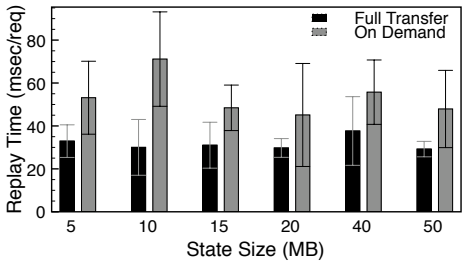
6.4 NFS Recovery Time Breakdown

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

We have found that 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. We split the recovery time into two portions: the time to verify a checkpoint and build the initial file system, and



(a) Loading Checkpoints



(b) Replaying Requests

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

the time to replay requests which occurred between the last checkpoint and the faulty request. Figure 9(a) shows the time for processing the checkpoints when using full transfer or ZZ's on demand approach (note the log scale). When transferring the full checkpoint, state sizes greater than a mere 25 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. Figure 9(b) shows the time required to replay requests. Since different state sizes may require different numbers of requests to be replayed, we report the average time per request replayed and the standard deviation. The ZZ 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.

6.5 NFS Performance

The Andrew filesystem benchmark emulates a software development workload of reading and modifying disk files. In this experiment we measure the time to complete the benchmark with and without faults. We use a scaled up version of the first four phases of the benchmark which include recursively creating source directories, copying source code files, examining file attributes, and reading files.[2] We use a modified version of the benchmark that can be scaled—instead of creating a single source directory, our enhanced benchmark creates N such directories and performs file operations on each set.

[2]Phase 5 of the benchmark, which compiles a set of source files, was omitted since it included old libraries that do not compile without substantial changes on modern Linux distributions.

Phase	ZZ	ZZ Fault	BASE	BASE Fault
1	3.09	3.10	2.85	2.85
2	62.70	61.37	50.01	50.26
3	32.01	47.90	25.37	35.95
4	47.77	71.62	35.51	33.95
Total	145.6	184.0	113.8	123.01

Table 2: Completion times for the Andrew benchmark. A fault is injected during phase 2.

We run the benchmark with $N = 50$ and inject a fault during phase 2 (file creation). This represents the worst possible time to inject a fault since a smaller amount of state would need to be rebuilt if it occurred earlier. Table 2 compares the completion time of the benchmark with and without faults; we report the average of three runs. While the fault occurs earlier on, it is not detected until phase 3 when the file is accessed, at which point ZZ's recovery replica takes an average of 16 seconds to build and verify the file system structure, and to replay any requests between the last checkpoint and the fault. Since phase 3 only examines the attributes of files, no files are copied to the recovery replica until phase 4. The additional latency of fetching files on demand in this phase increases the completion time by 23 seconds. In total, the fault adds 39 seconds of latency, an overhead of 26% across the full run.

For comparison, we present the completion time for BASE running the same benchmark. We again introduce a fault during phase 2 which forces BASE to restart a replica. Since BASE uses a full $3f + 1$ replicas, it is able to mask the fault and continue operating while the extra replica recovers. The recovery process slows down the correct replicas since they must assist the faulty node, adding 10 seconds to the total completion time. While BASE is capable of completing the benchmark with only an 8% performance drop, it does so with significantly higher cost since all $3f + 1$ replicas (or $2f + 1$ with separation of agreement and execution) must run at all times. ZZ only keeps $f + 1$ replicas active for the majority of the benchmark; $2f + 1$ replicas are only needed for a few seconds while the additional replica recovers and the faulty server is identified. Note that in this experiment the graceful performance of ZZ relative to BASE is reduced because the workload is driven by a single client, preventing ZZ from exploiting pipelining and batching to reduce overheads as described in the following section.

6.6 Graceful Performance

We next examine ZZ's performance under correct operating conditions. To find the minimal cost of the system, we use our client-server application to send "null" re-

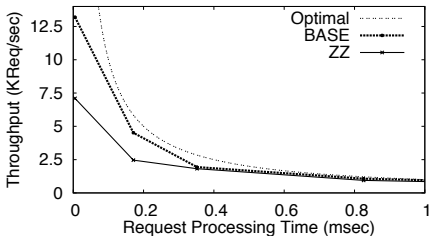
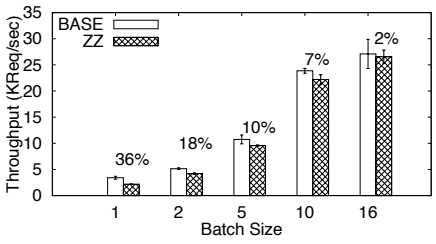
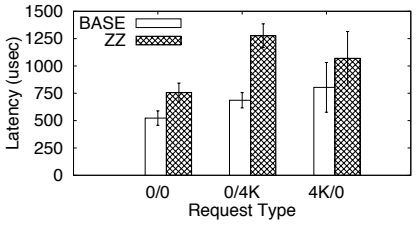


Figure 10: ZZ experiences some overhead compared to BASE, however, this quickly becomes negligible as the processing time for a request passes 0.5 msec.



(a) Throughput & ZZ Overhead

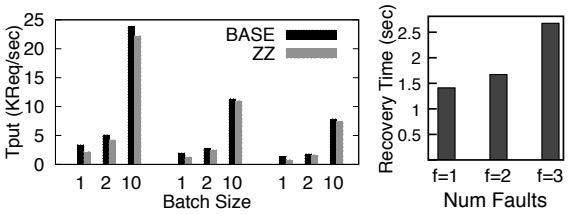


(b) Latency

Figure 11: Graceful mode performance of ZZ and BASE with different batch sizes and request types.

quests, which include no processing time or state modifications. Figure 11(a) illustrates the maximum throughput obtained for BASE and ZZ under various batch sizes; the labels indicate the percent performance loss seen by ZZ. Both systems see a large benefit from employing batching. With small batch sizes, ZZ's performance lags behind BASE due to overheads associated with separating the agreement and execution clusters. However, the extra cryptographic and network operations required in ZZ are applied per batch, so increasing the batch size significantly reduces this overhead. Figure 11(b) shows that ZZ sees only a modest latency overhead for requests with different request/reply sizes. This overhead is due to the additional cryptographic operations required in ZZ, and a high throughput can still be maintained with batching.

For realistic applications, the performance of "null" requests has little meaning since actual requests will incur processing at the execution nodes. Figure 10 demonstrates how ZZ and BASE perform when each request is required to perform a small amount of computation. The computation cost is varied by adjusting the size of a buffer which is repeatedly encrypted. While ZZ is initially slower than BASE due a small batch size, they are indistinguishable when the processing cost per request exceeds 250 microseconds. The Optimal curve represents the maximum throughput achievable for a given processing time if there are no network or cryptographic overheads. As the processing cost rises, it quickly becomes the dominant factor in determining throughput and both protocols perform very close to this upper bound.



(a) Graceful Performance (b) Recovery Time

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

6.7 Impact of Multiple Faults

Here we examine how ZZ's graceful performance and recovery time changes as we adjust f, the number of faults supported by the system. Figure 12(a) shows that ZZ's graceful mode performance scales similarly to BASE as the number of faults increases. This is expected because the number of cryptographic and network operations rises similarly in each system.

We next examine the recovery latency of the client-server application for up to three faults. In each case, we maintain f paused recovery replicas. We inject a fault to f of the active execution replicas and measure the recovery time to handle the faulty request. Figure 12(b) shows how the recovery time for $f = 1$ to $f = 3$ increases slightly due to increased message passing and because the ZFS server needs to export snapshots for a larger number of file systems. We believe that the communication costs could be decreased with hardware multicast and the use of multiple ZFS storage nodes.

6.8 Replay Vs State Updates

In this experiment we motivate separating computation from state updates with a microbenchmark comparing the latency of requests when full requests need to be replayed versus when only state updates need to be applied. We test this with a server application which in the ordinary case, receives a request from a client, performs a computation, and then writes the result to disk. If full requests must be replayed, then recovery replicas will have to perform this potentially expensive computation. Figure 13 illustrates how the replay time of requests changes as the cost of computation increases. By having replicas record and verify state updates as they are performed, recovery replicas only need to apply the state update (a disk write).

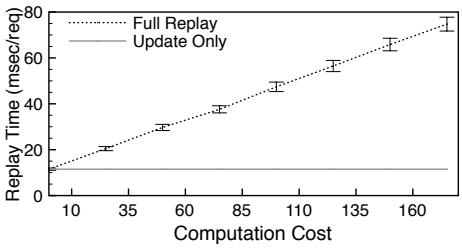


Figure 13: Separating state from execution can reduce latency when replaying requests.

Operation	Time (sec)
Naive Restore (2GB)	44.0
Paged-out Restore (128→2GB)	5.88
Unpause VM	0.29
ZFS Snapshot	0.03
ZFS Clone	0.60

Table 3: Using paused VMs results in the lowest latency, but ZZ's Paged-out approach can be used for applications without strong latency requirements.

6.9 VM Startup and ZFS Overheads

In our previous experiments, the recovery VMs are kept in a paused state; here we evaluate the feasibility of keeping recovery VMs hibernated on disk for applications with less stringent latency requirements, and present the latency incurred by different ZFS operations.

With a naive approach, maintaining replicas hibernated to disk can increase recovery latency by a factor proportional to the amount of memory allocated to each VM. This is because restoring a hibernated VM involves loading the VM's full memory contents from disk. Table 3 shows how our paged-out restore technique can reduce the startup time for a VM with a 2GB memory allocation from over 40 seconds to less than 6 seconds.

ZZ utilizes ZFS to simplify checkpoint creation at low cost. Table 3 illustrates the cost of the snapshot and clone procedures. Snapshots must be created at each checkpoint interval, and ZFS can efficiently support snapshot creation several times per second. The clone operation is used during recovery to make snapshots from the previous checkpoint available to the recovery VMs. This can be done in parallel with initializing the recovery VMs, and incurs only minimal latency.

7 Related Work

This section discusses related work not covered elsewhere. Lamport, Shostak, and Pease [15] introduced the problem of Byzantine agreement. Lamport also introduced the state machine replication approach [13] (with a popular tutorial by Schneider [22]) that relies on consensus to establish an order on requests. Consensus in the presence of asynchrony and faults has seen almost three decades of research. Dwork et al. [7] 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 [8] result showed that no agreement protocol is guaranteed to terminate with even one (benignly) faulty node in an asynchronous environment. Viewstamped replication [17] and Paxos [14] describe an agreement protocol for implementing an arbitrary state machine in an asynchronous benign environment that is safe when $n > 3f+1$ and live with synchrony assumptions.

Early BFT systems [20, 11] 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 [2] introduced a BFT SMR-based system that relied on synchrony only for liveness. The three-phase protocol at the core of PBFT is similar to viewstamped replication [17] or Paxos [14] and incurs a replication cost of at least $3f + 1$. More importantly, they showed that the latency and throughput overhead of BFT can be low enough to be practical. ZZ draws inspiration from Cheap Paxos [16], which advocated the use of cheaper auxiliary nodes used only to handle crash failures of main nodes. Our contribution lies in adapting the approach to tolerate Byzantine faults, demonstrating its feasibility through a prototyped system, and making it cheap and practical by leveraging a number of modern systems mechanisms.

Virtualization has been used in several BFT systems recently since it provides a clean way to isolate services. The VM-FIT systems exploits virtualization for isolation and to allow for more efficient proactive recovery [19]. The idea of "reactive recovery", where faulty replicas are replaced after fault detection, was used in [24], which also employed virtualization to provide isolation between different types of replicas. In ZZ, reactive recovery is not an optional optimization, but a requirement since in order to make progress it must immediately instantiate new replicas after faults are detected.

Rapid activation of virtual machine replicas for dynamic capacity provisioning and has been studied in [10, 25]. In contrast, ZZ uses VM replicas for high availability rather than scale. Live migration of virtual machines was proposed in [4] and its use for load balancing in a data center has been studied in [28]. Such techniques can be employed by ZZ to intelligently manage its free server pool, although we leave an implementation to future work. Virtualization has also been employed for security. Potemkin uses dynamically invocation of virtual machines to serve as a honeypot for security attacks [27]. Terra is a virtual machine platform for trusted computing that employs a trusted hypervisor [9]; ZZ only requires a trusted hypervisor if multiple replicas from an application are to be hosted on a single physical machine.

8 Conclusions

In this paper, we presented ZZ, a novel approach to construct general BFT services with a replication cost of practically $f + 1$, halving the $2f + 1$ or higher cost incurred by state-of-the-art approaches. Analogous to Remus [6] which uses virtualization to provide fail-stop fault tolerance at a minimal cost of $1 + \epsilon$, ZZ exploits virtualization to provide BFT at a low cost. Our key insight was to use $f+1$ execution replicas in the normal case and to activate additional VM replicas only upon failures. ZZ uses a number of novel mechanisms for rapid recovery of replicas, including the use of filesystem snapshots for efficient, application independent checkpoints, replaying state updates instead of full requests, and the use of an amortized state transfer mechanism that fetches state on-demand. We implemented ZZ using the BASE library, Xen and ZFS, and demonstrated the efficacy of our approach via an experimental evaluation.

References

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

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

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

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

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

[6] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asyncronous virtual machine replication. April 2008.

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

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

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

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

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

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

[13] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

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

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

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

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

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

[19] Hans P. Reiser and Rudiger Kapitza. Hypervisor-based efficient proactive recovery. In *26th IEEE International Symposium on Reliable Distributed Systems*, 2007.

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

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

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

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

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

[25] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning for multi-tier internet applications. In *Proceedings of the 2nd IEEE International Conference on Autonomic Computing (ICAC-05), Seattle, WA*, June 2005.

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

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

[28] T. Wood, P. Shenoy, A. Venkataramani, and M. Yousif. Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the Usenix Symposium on Networked System Design and Implementation (NSDI), Cambridge, MA*, April 2007.

[29] Timothy Wood, Rahul Singh, Arun Venkataramani, and Prashant Shenoy. Zz: Cheap practical bft using virtualization. Technical report, U. of Massachusetts Dept. of Computer Sciences, May 2008.

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

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