Zzyzx: Scalable Fault Tolerance through Byzantine Locking

James Hendricks[†*] Shafeeq Sinnamohideen[†] Gregory R. Ganger[†] Michael K. Reiter[‡]
[†]Carnegie Mellon University [‡]University of North Carolina at Chapel Hill

Abstract

Zzyzx is a Byzantine fault-tolerant replicated state machine protocol that outperforms prior approaches and provides near-linear throughput scaling. Using a new technique called Byzantine Locking, Zzyzx allows a client to extract state from an underlying replicated state machine and access it via a second protocol specialized for use by a single client. This second protocol requires just one round-trip and $2f+1$ responsive servers—compared to Zyzzyva, this results in 39–43% lower response times and a factor of 2.2–2.9× higher throughput. Furthermore, the extracted state can be transferred to other servers, allowing non-overlapping sets of servers to manage different state. Thus, Zzyzx allows throughput to be scaled by adding servers when concurrent data sharing is not common. When data sharing is common, performance can match that of the underlying replicated state machine protocol.

1. Introduction

As distributed systems grow in size and importance, they must tolerate complex software bugs and hardware misbehavior in addition to simple crashes and lost messages. Byzantine fault-tolerant protocols can tolerate arbitrary problems, making them an attractive building block, but system designers continue to worry that the performance overheads and scalability limitations are too great. Recent research has improved performance by exploiting optimism to improve common cases, but a significant gap still exists.

Zzyzx narrows that gap with a new technique called *Byzantine Locking*.[1] Layered atop a Byzantine fault-tolerant replicated state machine protocol (e.g., PBFT [7] or Zyzzyva [20]), Byzantine Locking temporarily gives a client exclusive access to state in the replicated state machine. It uses the underlying replicated state machine protocol to extract the relevant state and, later, to re-integrate it. Unlike locking in non-Byzantine fault-tolerant systems, Byzantine Locking is only a performance tool. To ensure liveness, locked state is kept on servers, and a client that tries to access objects locked by another client can request that the locks be revoked, forcing both clients back to the underlying replicated state machine to ensure consistency.

Byzantine Locking provides unprecedented scalability and efficiency for the common case of infrequent concurrent data sharing. Locked state is extracted to a set of *log servers*, which can execute on distinct physical computers from the replicas for the underlying replicated state machine. Thus, multiple log server groups, each running on distinct physical computers, can manage independently locked state, allowing throughput to be scaled by adding computers. Even when log servers and replicas share the same computers, exclusive access allows clients to execute operations much more efficiently—just one round-trip with only $2f+1$ responses, while tolerating f faulty servers.

Experiments, described in Section 6, show that Zzyzx can provide 39–43% lower latency and a factor of 2.2–2.9× higher throughput when using the same servers, compared to Zyzzyva, for operations on locked objects. Postmark [18] completes 60% more transactions on a Zzyzx-based file system than one based on Zyzzyva, and Zzyzx provides a factor of 1.6× higher throughput for a trace-based metadata workload. The benefits of locking outweigh the cost of unlocking after as few as ten operations. Operations on concurrently shared data objects do not use the Byzantine Locking layer—clients just execute the underlying protocol directly. Thus, except when transitioning objects from unshared to shared, the common case (unshared) proceeds with maximal efficiency and the uncommon case is no worse off than the underlying protocol.

Though Zzyzx provides the same correctness and liveness guarantees as PBFT and Zyzzyva under any workload, Byzantine Locking is most beneficial for services whose state consists of many objects that are rarely shared. This characterizes many critical services for which both scalability and Byzantine fault tolerance is desirable. For example, the metadata service of most distributed file systems contains a distinct object for each file or directory, and concurrent sharing is rare [3, 22].

2. Context and related work

Recent years have seen something of an arms race among researchers seeking to provide application writers with efficient Byzantine fault-tolerant substrates. Perhaps unintentionally, Castro and Liskov [7] initiated this race by

*James Hendricks is currently affiliated with Google.

[1]Pronounced *zai-ziks*, like *Isaac's*, Zzyzx is a populated settlement in San Bernardino County, California.

978-1-4244-7501-8/10/$26.00 ©2010 IEEE

	PBFT	Q/U	HQ	Zyzzyva	**Zzyzx**	RSM Lower Bound
Total servers required	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1**	3f+1 [28]
Responsive servers required	**2f+1**	4f+1	**2f+1**	3f+1	**2f+1**	2f+1
MAC ops at bottleneck server per request	2+(8f+1)/B	2+8f	4+4f	2+3f/B	**2**	1
Critical-path network 1-way latencies per req.	4	**2**	4	3	**2**	2
Throughput scales with added servers	No	Some*	Some*	No	**Yes**	–

Figure 1: **Comparison of Byzantine fault-tolerant replicated state machine protocols in the absence of faults and contention, along with commonly accepted lower bounds.** Data for PBFT, Q/U, HQ, and Zyzzyva are taken from [20]. Bold entries identify best-known values. f denotes the number of server faults tolerated, and B denotes the request batch size (see Section 6). "Responsive servers needed" refers to the number of servers that must respond in order to achieve good performance. *The throughput scalability provided by quorum protocols is limited by the requirement for overlap between valid quorums [23].

proposing a new protocol, PBFT, and labeling it "practical," because it performed better than most expected could be achieved with Byzantine fault-tolerant systems. Their protocol replaces the digital signatures common in previous protocols with message authentication codes (MACs) and also increases efficiency with request batching, link-level broadcast, and optimistic reads [6]. Still, the protocol requires four message delays and all-to-all communication for mutating operations, leaving room for improvement.

Abd-el-Malek et al. [1] proposed Q/U, a quorum-based Byzantine fault-tolerant protocol that exploits speculation and quorum constructions to provide throughput that can increase somewhat with addition of servers. Q/U provides Byzantine fault-tolerant operations on a collection of objects. Operations are optimistically executed in just one round-trip, and object histories are used to resolve issues created by concurrency or failures. Fortunately, concurrency and failures are expected to be rare in many important usages, such as the file servers that have been used as concrete examples in papers on this topic (e.g., [7]). Matching conventional wisdom, analysis of NFS traces from a departmental server [11] confirms that most files are used by a single client and that, when a file is shared, there is almost always only one client using it at a time.

Cowling et al. [8] proposed HQ, which uses a hybrid approach to achieve the benefits of Q/U without increasing the minimum number of servers ($3f$+1 for HQ vs. $5f$+1 for Q/U). An efficient quorum protocol executes operations unless concurrency or failures are detected. Each operation that encounters such issues then executes a second protocol to achieve correctness. In reducing the number of servers, HQ increases the common case number of message delays for mutating operations to four.

Kotla et al. [20] proposed Zyzzyva, which avoids all-to-all communication without additional servers, performs better than HQ under contention, and requires only three message delays. Unlike other protocols, however, Zyzzyva requires that all $3f+1$ nodes are responsive in order to achieve good performance, making Zyzzyva as slow as the slowest server. Also, requiring that all $3f+1$ servers respond to avoid extra work precludes techniques that reduce the number of servers needed in practice. For example, if only $2f+1$ servers need be responsive, the f "non-responsive" servers can be shared by neighboring clusters.

In a recent study of several Byzantine fault-tolerant replicated state machine protocols, Singh et al. concluded that "one-size-fits-all protocols may be hard if not impossible to design in practice" [26]. They note that "different performance trade-offs lead to different design choices within given network conditions." Indeed, there are several parameters to consider, including the total number of replicas, the number of replicas that must be responsive for good performance, the number of message delays in the common case, the performance under contention, and the throughput, which is roughly a function of the numbers of cryptographic operations and messages per request. Unfortunately, none of the above protocols score well on all of these metrics, as shown in Figure 1. PBFT requires four message delays and all-to-all communication, Q/U requires additional replicas, HQ requires four message delays and performs poorly under contention, and Zyzzyva performs poorly unless all nodes are responsive.

How Zzyzx fits in: Like prior systems, Zzyzx is optimized to perform well in environments where faults are rare and concurrency is uncommon, while providing correct operation under harsher conditions. During benign periods, Zzyzx performs and scales better than all of the prior approaches, requiring the minimum possible numbers of message delays (two, which equals one round-trip), responsive servers ($2f$+1), and total servers ($3f$+1). Zzyzx provides unprecedented scalability, because it does not require overlapping quorums as in prior protocols (HQ and Q/U) that provide any scaling; non-overlapping server sets can be used for frequently unshared state. When concurrency is common, Zzyzx performs similarly to its underlying protocol (e.g., Zyzzyva).

Zzyzx takes inspiration from the locking mechanisms used by many distributed systems to achieve high performance in benign environments. For example, GPFS uses distributed locking to provide clients byte-range locks that

enable its massive parallelism [25]. In benign fault-tolerant environments, where lockholders may crash or be unresponsive, other clients or servers must be able to break the lock. To tolerate Byzantine faults, Zzyzx must also ensure that lock semantics are not violated by faulty servers or clients and that a broken lock is always detected by correct clients.

By allowing clients to acquire locks, and then only allowing clients that have the lock on given state to execute operations on it, Zzyzx achieves much higher efficiency for sequences of operations from that client. Each replica can proceed on strictly local state, given evidence of lock ownership, thus avoiding all inter-replica communication. Also, locked state can be transferred to other servers, allowing non-overlapping sets of servers to handle independently locked state.

3. Definitions and system model

This paper makes the same assumptions about network asynchrony and the security of cryptographic primitives (e.g., MACs, signatures, and hash functions), and offers the same guarantees of liveness and correctness (linearizability), as the most closely related prior works [7, 8, 20].

Zzyzx tolerates up to f Byzantine faulty servers and any number of Byzantine faulty clients, given $3f + 1$ servers. As will be discussed, Zzyzx allows physical servers to take on different roles in the protocol, namely as *log servers* or state machine *replicas*. A log server and replica can be co-located on a single physical server, or each can be supported by separate physical servers. Regardless of the mapping of roles to physical servers, the presentation here assumes that there are $3f + 1$ log servers, at most f of which fail, and $3f + 1$ replicas, at most f of which fail.

Byzantine Locking requires no assumptions about the behavior of faulty nodes (i.e., Byzantine faults), except that they are unable to defeat the cryptographic primitives that correct nodes use to authenticate each others' messages (i.e., message authentication codes (MACs) and digital signatures). Moreover, it requires no assumptions about the synchrony of the network, beyond what the substrate replicated state-machine protocol requires. Because a (deterministic) replicated state machine cannot be guaranteed to make progress in an asynchronous network environment, even if only a single benign fault might occur [12], such protocols generally require the network to be eventually synchronous [10] in order to ensure liveness. In general, Byzantine Locking inherits the liveness properties of the underlying protocol.

As in prior protocols [1, 7, 8, 20], Zzyzx satisfies linearizability [17] from the perspective of correct clients. Linearizability requires that correct clients issue operations sequentially, leaving at most one operation outstanding at a time. The presentation in this paper also assumes this, but this requirement can be relaxed. Each operation applies to one or more *objects*, which are individual components of state within the state machine.

Two operations are *concurrent* if neither operation's response precedes the other's invocation. This paper makes a distinction between concurrency and contention. An object experiences *contention* if distinct clients submit concurrent requests to the object or interleave requests to it (even if those requests are not concurrent). For example, an object experiences frequent contention if two clients alternate writing to it. Low contention can be characterized by long *contention-free runs*, where multiple operations on an object are issued by a single client. It is precisely such contention-free runs on objects for which Byzantine Locking is beneficial, since it provides exclusive access to those objects and enables an optimized protocol to be used to invoke operations on them. As such, it is important for performance that objects be defined so as to minimize contention.

Zzyzx, HQ [8], and Q/U [1] provide an object-based state machine interface [15, Appendix A.1.1], which differs from other protocols [7, 20] in that each request must include the list of objects that it touches. Specifying which objects are touched in advance may complicate some operations (e.g., dereferencing pointers), but it poses no problems for many applications (e.g., distributed metadata services).

Many replication protocols elect a server as a leader, calling it the *primary* [7, 20] or *sequencer* [24, 27]. For simplicity and concreteness, this paper assumes Byzantine Locking on top of PBFT or Zyzzyva, so certain activities can be relegated to the primary to simplify the protocol. But, Byzantine Locking is not dependent on a primary-based protocol, and can build on a variety of underlying replicated state machine protocols.

4. Byzantine Locking and Zzyzx

This section describes Byzantine Locking and Zzyzx at a high level. Details and a more formal treatment are provided in the technical report [15, Chapter 5]. Byzantine Locking provides a client an efficient mechanism to modify replicated objects by providing the client temporary exclusive access to the object. A client that holds temporary exclusive access to an object is said to have *locked* the object. Zzyzx implements Byzantine Locking on top of PBFT [7] or Zyzzyva [20], as illustrated in Figure 2. In Zzyzx, objects are unlocked by default. At first, each client sends all operations through PBFT or Zyzzyva (the *substrate interface* labeled *A* in Figure 2). Upon realizing that there is little contention, the client sends a request through the substrate interface to lock a set of objects. The substrate interface and the locking operation are described in Section 4.1.

For subsequent operations that touch only locked objects, the client uses the *log interface* (*B* in Figure 2). The performance of Zzyzx derives from the simplicity of the log interface, which is little more than a replicated append-only

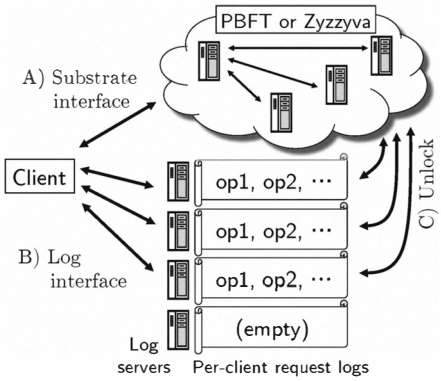

Figure 2: **Zzyzx components.** *The execution of Zzyzx can be divided into three subprotocols, described in Section 4. A) If a client has not locked the objects needed for an operation, the client uses a substrate protocol such as PBFT or Zyzzyva (Section 4.1). B) If a client holds locks for all objects touched by an operation, the client uses the log interface (Section 4.2). C) If a client tries to access an object for which another client holds a lock, the unlock subprotocol is run (Section 4.3).*

log. To issue a request, a client increments a sequence number and sends the request to $3f+1$ *log servers*, which may or may not be physically co-located with the substrate interface's replicas. If the request is in order, each log server appends the request to its per-client *request log*, executes the request on its local state, and returns a response to the client. If $2f+1$ log servers provide matching responses, the operation is complete. The log interface is described in Section 4.2.

If another client attempts to access a locked object through the substrate interface, the primary initiates the *unlock subprotocol* (*C* in Figure 2). The primary sends a message to each log server to unlock the object. Log servers reach agreement on their state using the substrate interface, mark the object as unlocked, and copy the updated object back into the replicas. If the client that locked the object subsequently attempts to access the object through the log interface, the log server replies with an error code, and the client retries its request through the substrate interface. The unlock subprotocol is described in Section 4.3.

4.1. The substrate interface and locking

In Zzyzx, each client maintains a list of locked objects that is the client's current best guess as to which objects it has locked. The list may be inaccurate without impacting correctness. Each replica maintains a special state machine object called the *lock table*. The lock table provides an authoritative description of which client, if any, has currently locked each object. The lock table also provides some per-client state, including a counter, vs.

Upon invoking an operation in Zzyzx, a client checks if any object touched by the operation is not in its list of locked objects, in which case the client uses the substrate interface. As in PBFT and Zyzzyva, the client sends its request to the primary replica. The primary checks if any object touched by the request is locked. If not, the primary resumes the substrate protocol, batching requests and sending ordering messages to the other replicas.

If an object touched by the request is locked, the primary initiates the unlock subprotocol, described in Section 4.3. The request is enqueued until all touched objects are unlocked. As objects are unlocked, the primary dequeues each enqueued request for which all objects touched by the request have been unlocked, and resumes the substrate protocol as above.

Note that a client can participate in Zzyzx using only the substrate protocol, and in fact does not need to be aware of the locking mechanism at all. In general, a replicated state machine protocol can be upgraded to support Byzantine Locking without affecting legacy clients.

A client can attempt to lock its working set to improve its performance. To do so, it sends a *lock request* for each object using the substrate protocol. The replicas evaluate a deterministic *locking policy* to determine whether to grant the lock. If granted, the client adds the object to its list of locked objects. The replicas also return the value of the per-client vs counter, which is incremented upon unlock and used to synchronize state between the log servers and replicas. If there is little concurrency across a set of objects, the entire set can be locked in one operation. For example, if each file in a file system is an object, then a client's entire home directory subtree could be locked upon login and the efficient log interface used for nearly all operations. The Zzyzx prototype uses a simple policy to decide to lock an object: each replica counts how often a single client accesses an object without contention. (The evaluation in Section 6 uses a threshold of ten.)

4.2. The log interface

Upon invoking an operation in Zzyzx, a client may find that all objects touched by the operation are in its list of locked objects, in which case the client uses the log interface. The client increments its request number, which is a local counter used for each operation issued through the log interface, and builds a message containing the request, the request number, and the vs. It then computes a MAC of the message for each log server. (Large requests are hashed, and the hash is used in the MACs.) The client sends the message and all MACs to each log server.

Upon receiving a request, each log server verifies its MAC. The log server then verifies that the request is in order as follows: If the request number is lower than the most recent request number for the client, the request is a duplicate and is ignored. If the request number matches the most recent number, the most recent response is re-sent. If the re-

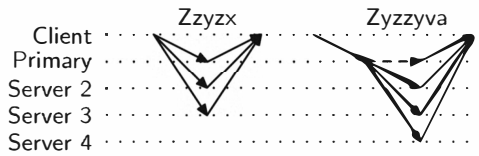

Figure 3: **Basic communication pattern of Zzyzx versus Zyzzyva.** *Operations on locked objects in Zzyzx complete in a single round-trip to $2f+1$ log servers. Zyzzyva requires three message delays, if all $3f+1$ replicas are responsive, or more message delays, if some replicas are unresponsive.*

quest number is greater than the next in sequence, or if the vs value is greater than the log server's value, the log server must have missed a request or an unlock, so it initiates state transfer (Section 5.1). If the log server has promised not to access an object touched by the request (since the object is in the process of being unlocked, as described in Section 4.3), it returns failure.

If the request number is next in sequence, the log server tries to execute the request. It lazily fetches objects from replicas as needed by invoking the substrate interface. Of course, if a log server is co-located with a replica, pointers to objects may be sufficient. If fetching an object fails because the object is no longer locked by the client, the log server returns failure. Otherwise, the log server has a local copy of each object that is touched by the request. It executes the request on its local copy, appends the request, vs, request number, and the set of MACs to its request log, and returns a MACed message with the response and the client's MACs. (If the returned MACs do not match those sent, the client re-sends the MACs and a MAC of the MACs [15, Appendix B.7].) Upon receiving $2f+1$ non-failure responses, the client returns the majority response.

If any log server returns failure, the client sends a *retry request* through the substrate interface, which includes both the request and the request number. Each replica checks if the request completed at the log servers before the last execution of the unlock subprotocol, in which case the replicas tell the client to wait for a response from a log server. Otherwise, the replicas execute the request.

Figure 3 shows the basic communication pattern of the log interface in Zzyzx versus Zyzzyva. Zyzzyva requires 50% more network hops than Zzyzx, and Zyzzyva requires all $3f+1$ servers to be responsive to perform well, f more than the $2f+1$ responsive servers required by Zzyzx. Zzyzx improves upon Zyzzyva further, though, by removing the bottleneck primary and requiring less cryptography at servers. The latter improvement obviates the need for batching, a technique used in previous protocols [6, 19, 20, 24] where the primary accumulates requests before sending them to other replicas. Batching amortizes the cryptographic overhead of the agreement subprotocol over many requests, but waiting to batch requests before execution increases latency in Zyzzyva. Because Byzantine Locking provides clients temporary exclusive access to objects, each client can order its own requests for locked objects, avoiding the need for an agreement subprotocol.

4.3. Handling contention

The protocol, as described so far, is a simple combination of operations issued to PBFT or Zyzzyva (Section 4.1) and requests appended to a log (Section 4.2). The unlock subprotocol is what differentiates Byzantine Locking from prior lease- and lock-like mechanisms in systems such as Farsite [2] and Chubby [5].

If a request touches an unlocked object (Section 4.1) or is retried because a log server returned failure (Section 4.2), then the client sends the request to the primary using the substrate interface. The primary checks if the request touches any locked objects and, if so, initiates the unlock subprotocol described in this section. In general, the unlock subprotocol can unlock multiple objects in a single execution, but, for clarity, this section describes unlocking a single object.

The unlock subprotocol consists of a fast path and a full path, both shown in Figure 4. The fast path requires just a single round-trip between the primary and $2f+1$ log servers. Full unlock requires additional communication, but it is required only when a client or log server is faulty, or when request logs do not match due to concurrency.

Fast unlock: In the fast unlock path (*A* in Figure 4), the primary sends a "Try unlock" message to each log server, describing the object (or set of objects) being unlocked. Each log server constructs a message containing the hash of its request log and a hash of the object. A designated replier includes the value of the object in its message (as in replies for PBFT [7]). Once again, if log servers are co-located with replicas, only a pointer to the object need be sent. Each log server sends its response to the primary formatted as a request to the substrate interface.

Upon receiving $2f+1$ responses with matching object and request log hashes and at least one object that matches the hashes, the primary sends the responses through the substrate protocol, batched with any requests enqueued due to object contention (see Section 4.1). Each replica marks the object unlocked, and vs is incremented before the next lock or full unlock operation. Before sending a response to the primary, each log server adds the object to a list of objects it promises not to touch until the next instantiation of the full unlock subprotocol. This list prevents the log server from touching potentially unlocked objects in concurrent appends (see Section 4.2).

The full unlock subprotocol: If request log hashes do not match in the fast path, the full unlock subprotocol is executed (*B* in Figure 4). The primary fetches signed request logs from $2f+1$ log servers ("Break lock" in Figure 4).

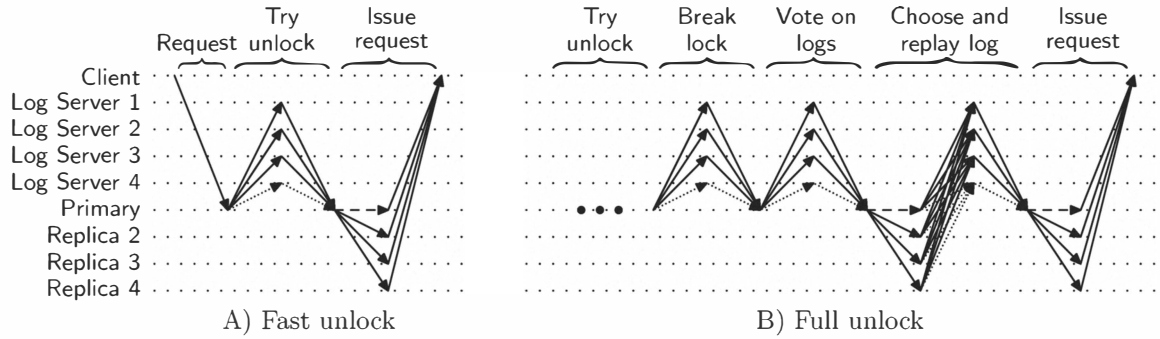

Figure 4: **Unlock message diagram.** A) The primary fetches a hash of the request log at $2f+1$ log servers ("Try Unlock"). If hashes match, the primary sends the hashes and the conflicting request through the substrate interface ("Issue request"), which unlocks the object. B) Otherwise, the primary fetches request logs from $2f+1$ log servers ("Break lock"). The primary then asks each log server to vote on which requests have valid MACs ("Vote on logs"). Each log server sends its votes via the substrate interface. Replicas choose the longest sequence of requests with $f+1$ votes, and return the chosen request log as a substrate protocol response. Each log server replays that request log to reach a consistent state ("Choose and replay log"). Finally, as above, the log servers send the primary matching hashes, which the primary sends with the conflicting request through the substrate interface ("Issue request"). (See Section 4.3.)

(Signatures can be avoided using standard techniques, but full unlock is rare.) Before sending its request log, a log server adds the object (or set of objects) being unlocked to its list of objects that it promises not to touch until the next full unlock, as in the fast path. Unfortunately, the replicas cannot verify the MACs stored with each request in the request logs (Section 4.2). Thus, the primary sends the signed request logs to the log servers, which "vote" on each request log entry to prove whether the client invoked each request, as follows.

Each log server sends a substrate interface request that lists which request log entries have valid MACs. ("Vote on logs" in Figure 4.) Replicas create a new log consisting of the longest sequence of request log entries such that each touched object is locked and each request has at least $f+1$ votes, ensuring that the client invoked each request. Replicas return this log to each log server in a substrate interface response. Each log server replays this request log as needed, thus reaching a consistent state that matches the state at other correct log servers. ("Choose and replay log" in Figure 4.) Each log server then marks the object unlocked, increments vs, and clears the list of objects it promised not to touch. Finally, as above, correct log servers send the primary matching hash values describing their state and the object to be unlocked. The primary sends these hashes, and any requests enqueued due to object contention (see Section 4.1), in a batch through the substrate protocol. Each replica marks the object unlocked and increments vs.

The primary sends and each log server checks vs before each "Try unlock", "Break lock", and "Replay log" message. If the log server's vs is greater than the primary's, then the message is stale. If the primary's vs is greater than the log server's, the log server missed an unlock, so it initiates state transfer (Section 5.1).

5. Protocol details

The log servers use checkpointing and state transfer mechanisms, described in Section 5.1, similar to those in PBFT [7], HQ [8], and Zyzzyva [20]. Section 5.2 describes optimizations for read-only requests, more aggressive locking, lower contention, and preferred quorums. Section 5.3 discusses how Zzyzx can provide near-linear scalability by deploying additional replicas. In contrast, the throughput of most Byzantine fault-tolerant protocols cannot be increased significantly by adding additional replicas, because all requests flow through a bottleneck node (e.g., the primary in PBFT and Zyzzyva) or overlapping quorums (which provides limited scalability).

Though this paper assumes that at most f servers (log servers or replicas) fail, Byzantine Locking (and many other Byzantine fault-tolerant protocols) can support a hybrid failure model that allows for different classes of failures. As in Q/U [1], suppose that at least $n-t$ servers are correct, and that at least $n-b$ are *honest*, i.e., either correct or fail only by crashing; as such, $t \geq b$. Then, the total number of servers is $b+2t+1$ rather than $3f+1$, and the quorum size is $b+t+1$ rather than $2f+1$. Of course, when $f=b=t$, it is the case that $b+2t+1=3f+1$ and $b+t+1=2f+1$. The benefit of such a hybrid model is that one additional server can provide some Byzantine fault-tolerance. (More generally, b additional servers can tolerate b simultaneous Byzantine faults.) A hybrid model suits deployments where arbitrary faults, such as faults due to soft errors, are less common than crash faults.

5.1. Checkpointing and state transfer

Log servers should checkpoint their state periodically to truncate their request logs and to limit the amount of work needed for a full unlock. The full unlock operation acts as

a checkpointing mechanism, because log servers reach an agreed-upon state. Thus, upon full unlock, requests prior to the unlock can be purged. The simplest checkpoint protocol is for log servers to execute the full unlock subprotocol for a null object at fixed intervals. Zzyzx can also use standard checkpointing techniques found in Zyzzyva [20] and similar protocols, which may be more efficient.

If a correct client sends a request number greater than the next request number in order, the log server must have missed a request. The log server sends a message to all $3f$ other log servers, asking for missed requests. Upon receiving matching values for the missing requests and the associated MACs from $f+1$ log servers, the log server replays the missed requests on its local state to catch up. Since $2f+1$ log servers must have responded to each of the client's previous requests, at least $f+1$ correct log servers must have these requests in their request logs. A log server may substitute a checkpoint in place of prior requests.

5.2. Optimizations

Read-only requests: A client can read objects locked by another client if all $3f+1$ log servers return the same value, as in Zyzzyva. If $2f+1$ log servers return the same value and the object was not modified since the last checkpoint, the client can return that value. If the object was modified, the client can request a checkpoint, which may be less expensive than the unlock subprotocol.

Aggressive locking: If an object is locked but never fetched, there is no need to run the unlock subprotocol. The primary just sends conflicting requests through the standard substrate protocol, which will deny future fetch requests pertaining to the previous lock. Thus, aggressively locking a large set of objects does not lower performance.

Pre-serialization: Section 6.5 finds that Zzyzx outperforms Zyzzyva for contention-free runs as short as ten operations. The pre-serializer technique of Singh et al. [27] could make the break-even point even lower.

Preferred quorums: As in Q/U [1] and HQ [8], Zzyzx takes advantage of *preferred quorums*. Rather than send requests to all $3f+1$ log servers for every operation, a client can send requests to $2f+1$ log servers if all $2f+1$ servers provide matching responses. This optimization limits the amount of data sent over the network, which is useful when the network is bandwidth- or packet-limited, or when the remaining f replicas are slow. It also frees f servers to process other tasks or operations in the common case, thus allowing a factor of up to $\frac{3f+1}{2f+1}$ higher throughput.

5.3. Scalability through log server groups

There is nothing in Section 4 that requires the group of replicas used in the Byzantine fault-tolerant replicated state machine protocol to be hosted on the same servers as the log servers. Thus, a system can deploy replicas and log servers on distinct servers. Similarly, the protocol can use multiple distinct groups of log servers. An operation that spans multiple log server groups can always be completed through the substrate interface. The benefit of multiple log server groups is near linear scalability in the number of servers, which far exceeds the scalability that can be achieved by adding servers in prior protocols.

6. Evaluation

This section evaluates the performance of Zzyzx and compares it with that of Zyzzyva and that of an unreplicated server. Zyzzyva is measured because it outperforms prior Byzantine fault-tolerant protocols [20, 26]. Zzyzx is implemented as a module on top of Zyzzyva, which in turn is a modified version of the PBFT library [7]. For evaluation, MD5 was replaced with SHA1 in Zyzzyva and Zzyzx, because MD5 is no longer considered secure [29].

Since Zyzzyva does not use Byzantine Locking, replicas must agree on the order of requests before a response is returned to the client (rather than the client ordering requests on locked objects). Agreeing on order requires that the primary generate and send MACs to all $3f$ other replicas, which would be expensive if done for every operation. Thus, the primary in Zyzzyva accumulates a batch of B requests before ordering them all by generating a single set of MACs that is sent to all $3f$ other replicas, amortizing the network and cryptographic cost over several requests. This section considers Zyzzyva with batch sizes of B=1 and B=10.

The micro-benchmark workload used in Sections 6.2–6.4 consists of each client process performing a null request and receiving a null reply. Each client accesses an independent set of objects, avoiding contention. A client running Zzyzx locks each object on first access. The workload is meant to highlight the overhead found in each protocol, as well as to provide a basis for comparison by reproducing prior experiments. Section 6.5 considers the effects of contention, and Section 6.6 evaluates a benchmark and trace-based workload on a file system that uses Zzyzx for metadata operations.

6.1. Experimental setup

All experiments are performed on a set of computers that each have a 3.0 GHz Intel Xeon processor, 2 gigabytes of memory, and an Intel PRO/1000 network card. All computers are connected to an HP ProCurve Switch 2848, which has a specified internal bandwidth of 96 Gbps (69.3 Mpps). Each computer runs Linux kernel 2.6.28-7 with default networking parameters. Experiments use the Zyzzyva code released by the protocol's authors, configured to use all optimizations. Both Zyzzyva and Zzyzx use UDP multicast. After accounting for the performance difference between SHA1 and MD5, the this section's evaluation of Zyzzyva agrees with that of Kotla et al. [20].

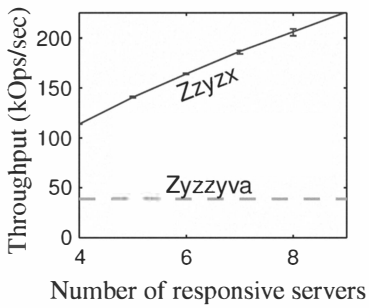

Figure 5: **Throughput vs. servers.** *Zzyzx's throughput scales nearly linearly as servers are added.*

A Zyzzyva replica process runs on each of $3f+1$ server computers. For Zzyzx, except where noted, one Zyzzyva replica process and one log server process run on each of $3f+1$ server computers. Zzyzx is measured both with the preferred quorum optimization of Section 5.2 enabled (labeled "Zzyzx") and with preferred quorums disabled (labeled "Zzyzx-NPQ").

Each physical client computer runs 10 instances of the client process. This number was chosen so that the client computer does not become processor-bound. All experiments are run for 90 seconds, with measurements taken from the middle 60 seconds. The mean of at least 3 runs is reported, and the standard deviation for all results is within 3% of the mean.

6.2. Scalability

Figure 5 shows the throughput of Zzyzx and Zyzzyva as the number of servers increases when tolerating one fault. Zyzzyva cannot use additional servers to improve throughput, so the dashed line repeats its 4-server throughput for reference. Although data is only shown for $f = 1$, the general shape of the curve applies when tolerating more faults.

Even with the minimum number of servers (4), Zzyzx outperforms Zyzzyva by a factor of 2.9× higher throughput. For Zzyzx, the first $3f+1$ log servers are co-located with the Zyzzyva replicas. Additional log servers run on dedicated computers. Since only $2f+1$ log servers are involved in each operation and independent log server sets do not need to overlap, the increase in usable quorums results in nearly linear scalability.

6.3. Throughput

Figure 6 shows the throughput achieved, while varying the number of clients, when tolerating a single fault and when all servers are correct and responsive. Zzyzx significantly outperforms all Zyzzyva configurations. Zzyzx's maximum throughput is 2.9× that of Zyzzyva's with B=10, and higher still compared to Zyzzyva without batching. When Zzyzx is run on f+1 additional servers (6 total), it's maximum throughput is 3.9× that of Zyzzyva with B=10. Even without preferred quorums, Zzyzx's maximum throughput is 2.2× that of Zyzzyva with B=10, due to Zzyzx's lower network and cryptographic overhead.

Due to the preferred quorums optimization, Zzyzx provides higher maximum throughput than the unreplicated server, which simply generates and verifies a single MAC, because each log server processes only a fraction ($\frac{2f+1}{3f+1}$) of the requests. With preferred quorums disabled (Zzyzx-NPQ), Zzyzx provides lower throughput than the unreplicated server due to checkpoint, request log, and network overheads.

Zzyzx performs as well or better than Zyzzyva for larger request and response sizes. For larger request sizes, such as the 4 kB request and null reply found in the 4/0 micro-benchmark of Castro and Liskov [7], Zzyzx provides 1.3× higher throughput ($\frac{3f+1}{2f+1}$) than Zyzzyva because each log server processes only a fraction of requests.

6.4. Latency

Figure 7 shows the average latency for a single operation under varied load. When serving one request, Zzyzx exhibits 39–43% lower latency than Zyzzyva, and Zzyzx continues to provide lower latency as load increases. The lower latency is because Zzyzx requires only 2 one-way message delays (33% fewer than Zyzzyva), each server computes fewer MACs, and log servers in Zzyzx never wait for a batch of requests to accumulate before executing a request and returning its response. (Though important for high throughput, batching increases latency in Zyzzyva.)

6.5. Performance under contention

Figure 8 shows the performance of Zzyzx under contention. For this workload, each client accesses an object a fixed number of times before the object is unlocked. The client then procures a new lock and resumes accessing the object. The experiment identifies the *break-even* point of Zzyzx, i.e., the length of the shortest contention-free run for which Zzyzx outperforms Zyzzyva.

When batching in Zyzzyva is disabled to improve latency, Zzyzx outperforms Zyzzyva for contention-free runs that average ≥ 10 operations. Zzyzx outperforms Zyzzyva when batching is enabled (B=10) for contention-free runs that average ≥ 20 operations. Zzyzx achieves 85–90% of its top throughput for contention-free runs of 160 operations.

6.6. Postmark and trace-based execution

To compare Zzyzyva and Zzyzx in a full system, we built a memory-backed file system that can use either Zzyzyva or Zzyzx for its metadata storage. We focus on file system metadata rather than data because efficient protocols specialized for fault-tolerant block data storage already exist [13, 16]. Zzyzx completes 566 Postmark [18] transactions per second (TPS) compared to Zyzzyva's 344 TPS, an increase of 60%. Postmark produces a workload with

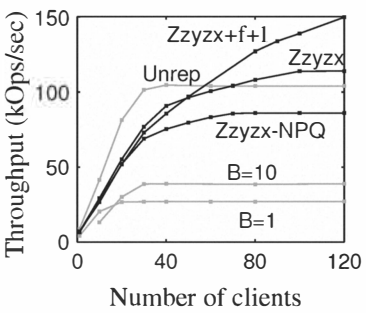

Figure 6: **Throughput vs. clients**

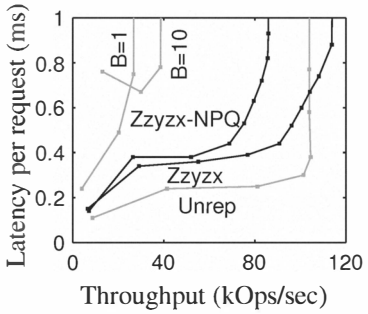

Figure 7: **Latency vs. throughput**

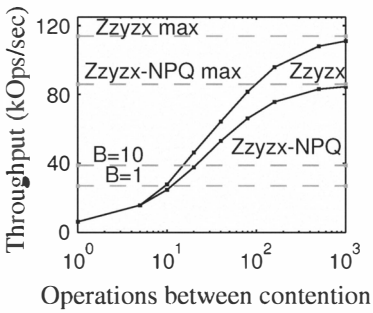

Figure 8: **Throughput vs. contention**

$f = 1$ and all servers are responsive. The dashed lines in Figure 8 show throughput with no contention.

many small files, similar to the workload found in a mail server, and each transaction consists of one or more Zzyzx or Zyzzyva operations plus some processing time. Because Postmark is single-threaded, its performance depends mainly on request response time and does not benefit from batching.

Metadata operations were extracted from NFS traces of a large departmental server. Over 14 million metadata operations were considered from the Harvard EECS workload between Monday 17–Friday 21 of February 2003 [11]. A matching operation mix was then executed on Zzyzx and Zyzzyva. Zzyzx was able to use the log interface for 82% of operations, with an average contention-free run length of 4926 operations. Of the remaining 18% of operations executed through the Zyzzyva substrate interface, 56% were read-only and used Zyzzyva's one round-trip read optimization. When all operations were executed through Zyzzyva, the read optimization was used for 55% of operations. Zzyzx completed operations at a rate of 104.7 kOps/sec, and Zyzzyva completed operations at a rate of 64.8 kOps/sec. Thus, Zzyzx provides a factor of 1.6× higher throughput for this trace-based metadata workload.

7. Additional related work

Recent Byzantine fault-tolerant replicated state machine protocols build upon several years of prior research. SecureRing uses an unreliable failure detector to provide an asynchronous Byzantine fault-tolerant group communication abstraction [19], which can be used to build a replicated state machine. The Rampart toolkit implements an asynchronous Byzantine fault-tolerant replicated state machine [24]. The technical report provides further discussion [15, Section 5.1.3].

Guerraoui et al. introduce a modular framework for Byzantine fault-tolerant replicated state machine protocols and the Quorum and Chain subprotocols [14]. As in Zyzzyva, Quorum and Chain require that all $3f + 1$ replicas are responsive. Quorum [14] looks like Zyzzyva [20] without a pre-serializer [27]. Chain is a pipelined version of Quorum, which increases throughput but also increases latency by increasing the number of message delays per request. Zzyzx is more similar to PBFT and HQ than Quorum or Chain. Zzyzx, PBFT, and HQ each require $2f + 1$ responsive servers of $3f + 1$ total. Each have a setup phase that determines which client will issue the next request on a set of objects (lock for Zzyzx, write-1 for HQ, and pre-prepare for PBFT). The difference is that Zzyzx allows multiple operations on a set of objects after a single setup phase. Put another way, Zzyzx batches operations for a single client.

Farsite [2, 4, 9] uses a Byzantine fault-tolerant replicated state machine to manage metadata in a distributed file system. Farsite issues leases to clients for metadata such that clients can update metadata locally, which can increase scalability. Upon conflict or timeout, leases are recalled, but updates may be lost if the client is unreachable. Leasing schemes do not provide the strong consistency guarantees expected of replicated state machines (linearizability [17]), so leasing is not acceptable for some applications. Also, choosing lease timeouts presents an additional challenge: a short timeout increases the probability that a client will miss a lease recall or renewal, but a long timeout may stall other clients needlessly in case of failure. Farsite expires leases after a few hours [9], which is acceptable only because Farsite is not designed for large-scale write sharing [4].

To scale metadata further, Farsite hosts metadata on multiple independent replicated state machines called directory groups. To ensure namespace consistency, Farsite uses a special-purpose subprotocol to support Windows-style renames across directory groups [9]. This subprotocol allows Farsite to scale, but it is inflexible and does not generalize to other operations. For example, the subprotocol cannot handle POSIX-style renames [9]. Byzantine Locking would allow Farsite and similar protocols to maintain scalability without resorting to a special-purpose protocol.

Yin et al. [30] describe an architecture in which agreement on operation order is separated from operation execution, allowing execution to occur on distinct servers. But, the replicated state machine protocol is never relieved of the task of ordering operations, and as such, remains a bottle-

neck. Dividing the state machine into objects, as in Zzyzx, has been used in many previous systems. For example, Kotla et al. [21] describe CBASE, which partitions a state machine into objects and allows each replica to execute operations known to involve only independent state concurrently.

8. Conclusion

Byzantine Locking allows creation of efficient and scalable Byzantine fault-tolerant services. Compared to the state-of-the-art (Zyzzyva), Zzyzx delivers a factor of 2.2×–2.9× higher throughput during concurrency-free and fault-free operation, given the minimum number of servers ($3f+1$). Moreover, unlike previous Byzantine fault-tolerant replicated state machine protocols, Zzyzx offers near-linear scaling of throughput as additional servers are added.

9. Acknowledgments

We thank the members and companies of the CyLab Corporate Partners and the PDL Consortium (including APC, EMC, Facebook, Google, Hewlett-Packard Labs, Hitachi, IBM, Intel, LSI, Microsoft Research, NEC Laboratories, NetApp, Oracle, Seagate, Symantec, VMware, and Yahoo! Labs) for their interest, insights, feedback, and support. We thank Ramakrishna Kotla et al. and Miguel Castro et al. for sharing source code for Zyzzyva and PBFT. This material is based on research sponsored in part by the National Science Foundation, via grants CCR-0326453 and CNS-0910483, the Army Research Office, under grant number DAAD19-02-1-0389, and the Air Force Office of Scientific Research, under grant number F49620-01-1-0433.

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. In *Proc. SOSP 2005*, pages 59–74.

[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. OSDI 2002*, pages 1–14.

[3] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. In *Proc. SOSP 1991*, pages 198–212.

[4] W. J. Bolosky, J. R. Douceur, and J. Howell. The Farsite project: A retrospective. *SIGOPS OSR*, 41(2):17–26, 2007.

[5] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. OSDI 2006*, pages 335–350.

[6] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, January 2001. MIT-LCS-TR-817.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. OSDI 1999*, pages 173–186.

[8] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. OSDI 2006*, pages 177–190.

[9] J. R. Douceur and J. Howell. Distributed directory service in the Farsite file system. In *Proc. OSDI 2006*, pages 321–334.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[11] D. Ellard and M. Seltzer. New NFS tracing tools and techniques for system analysis. In *Proc. LISA 2003*, pages 73–86.

[12] 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.

[13] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. DSN 2004*, pages 135–144.

[14] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The next 700 BFT protocols. Technical Report 2008-008, EPFL.

[15] J. Hendricks. *Efficient Byzantine Fault Tolerance for Scalable Storage and Services.* PhD thesis, July 2009. CMU-CS-09-146.

[16] J. Hendricks, G. R. Ganger, and M. K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proc. SOSP 2007*, pages 73–86.

[17] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM TOPLAS*, 12(3):463–492, 1990.

[18] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Appliance, October 1997.

[19] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proc. HICSS 1998*, pages 317–326.

[20] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. SOSP 2007*, pages 45–58.

[21] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proc. DSN 2004*, pages 575–584.

[22] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *Proc. USENIX 2008 ATC*, pages 213–226.

[23] D. Malkhi, M. K. Reiter, and A. Wool. The load and availability of Byzantine quorum systems. *SIAM J. Comp.*, 29(6):1889–1906, 2000.

[24] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Selected Papers from Int. Workshop on Theory and Practice in Dist. Systems*, pages 99–110, 1995.

[25] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. FAST 2002*, pages 231–244.

[26] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. BFT protocols under fire. In *Proc. NSDI 2008*, pages 189–204.

[27] A. Singh, P. Maniatis, P. Druschel, and T. Roscoe. Conflict-free quorum based BFT protocols. Technical Report TR-2007-2, Max Planck Institute for Software Systems.

[28] S. Toueg. Randomized Byzantine agreements. In *Proc. PODC 1984*, pages 163–178.

[29] X. Wang and H. Yu. How to break MD5 and other hash functions. In *EUROCRYPT '05*, pages 19–35.

[30] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. SOSP 2003*, pages 253–267.