Zyzzyva: Speculative Byzantine Fault Tolerance

Ramakrishna Kotla *
Microsoft Research
Silicon Valley, USA
kotla@microsoft.com

Allen Clement, Edmund Wong,
Lorenzo Alvisi, and Mike Dahlin
Dept. of Computer Sciences
University of Texas, Austin, USA
{aclement,elwong,lorenzo,dahlin}@cs.utexas.edu

ABSTRACT

A longstanding vision in distributed systems is to build reliable systems from unreliable components. An enticing formulation of this vision is Byzantine fault tolerant (BFT) state machine replication, in which a group of servers collectively act as a correct server even if some of the servers misbehave or malfunction in arbitrary ("Byzantine") ways. Despite this promise, practitioners hesitate to deploy BFT systems at least partly because of the perception that BFT must impose high overheads.

In this article, we present Zyzzyva, a protocol that uses speculation to reduce the cost of BFT replication. In Zyzzyva, replicas reply to a client's request without first running an expensive three-phase commit protocol to agree on the order to process requests. Instead, they optimistically adopt the order proposed by a primary server, process the request, and reply immediately to the client. If the primary is faulty, replicas can become temporarily inconsistent with one another, but clients detect inconsistencies, help correct replicas converge on a single total ordering of requests, and only rely on responses that are consistent with this total order. This approach allows Zyzzyva to reduce replication overheads to near their theoretical minima and to achieve throughputs of tens of thousands of requests per second, making BFT replication practical for a broad range of demanding services.

1. INTRODUCTION

Mounting evidence suggests that real systems must contend not only with simple crashes but also with more complex failures ranging from hardware data corruption [22] to nondeterministic software errors [25] to security breaches. Such failures can cause even highly-engineered services to become unavailable or to lose data. For example, a single corrupted bit in a handful of messages recently brought down the Amazon S3 storage service for several hours [4], and several well-known email service providers have occasionally lost customer data [14].

Byzantine fault tolerant (BFT) state machine replication is a promising approach to masking many such failures and constructing highly reliable and available services. In BFT replication, $n \geq 3f + 1$ servers collectively act as a *correct* server even if up to f servers misbehave or malfunction in arbitrary ("Byzantine") ways [15, 16].

Today, three trends make real-world deployment of BFT increasingly attractive. First, as noted above, there is mounting evidence of non-fail-stop behaviors in real systems, motivating the use of new techniques to improve robustness. Second, the growing value of data and the falling costs of hardware make it advantageous for service providers to trade increasingly inexpensive hardware for the peace of mind potentially provided by BFT replication. Third, improvements to the state of the art in BFT algorithms [1, 3, 6, 13, 23, 26] have narrowed the gap between BFT replication costs and the costs already being paid for non-BFT replication by many commercial services. For example, by default, the Google file system uses 3-way replication of storage [9], which is roughly the cost of tolerating one Byzantine failure by using 3 full replicas plus one additional lightweight node to help the replicas coordinate their actions [26].

Unfortunately, practitioners hesitate to deploy BFT systems at least partly because of the perception that BFT must impose high overheads. This concern motivates our work, which seeks to answer a simple question: *Can we build a system that tolerates a broad range of faults while meeting the demands of high performance services?*

To answer this question, this article presents Zyzzyva[1]. Zyzzyva seeks to make BFT replication deployable for the widest range of practical services by implementing the extremely general abstraction of a replicated state machine at an extremely low cost.

The basic idea of BFT state machine replication is simple: a client sends a request to a replicated service and the service's distributed agreement protocol ensures that correct servers execute the same requests in the same order [24]. If the service is deterministic, each correct replica thus traverses the same series of states and produces the same reply to each request. The servers send their replies back to the client, and the client accepts a reply that matches across a sufficient number of servers.

Zyzzyva builds on this basic approach, but reduces its cost through *speculation*. As is common in existing BFT replication protocols, an elected *primary* server proposes an order on client requests to the other server *replicas* [3]. However, unlike in traditional protocols, Zyzzyva replicas then immediately execute requests speculatively, without running an expensive agreement protocol to definitively establish the order. As a result, if the primary is faulty, correct replicas' states may diverge, and they may send different responses to a client. Nonetheless, Zyzzyva preserves correctness because a correct client detects such divergence and avoids acting on a reply until the reply and the sequence of preceding requests are *stable* and guaranteed to eventually be adopted by all correct

*This work was done when the author was at the University of Texas at Austin.

[1]Zyzzyva (ZIZ-uh-vuh) is the last word in the dictionary. According to dictionary.com, a zyzzyva is "any of various tropical American weevils of the genus Zyzzyva, often destructive to plants."

servers. Thus, applications at correct clients observe the traditional abstraction of a replicated state machine that executes requests in a linearizable [10] order.

Essentially, Zyzzyva "rethinks the sync" [19] for BFT. Whereas past BFT systems have pessimistically enforced the condition that *a correct server only emits replies that are stable*, Zyzzyva recognizes that this condition is stronger than required. Instead, Zyzzyva enforces the weaker condition: *a correct client only acts on replies that are stable*. This change allows us to move the output commit from the servers to the client, which in the optimized case allows servers to avoid expensive all-to-all communication that they would otherwise require to ensure the stronger condition.

Leveraging the client in this way allows us to minimize server overheads and maximize throughputs in the optimized, failure-free case. As a result, Zyzzyva's peak measured throughput of over 86K requests/second on 3.0GHz Pentium-IV machines makes it feasible to utilize BFT replication in a broad range of demanding services. Despite these aggressive optimization to the fault-free case, Zyzzyva retains good performance of over 82K requests/second even when up to f backup replicas crash. In fact, Zyzzyva's replication costs, processing overheads, and communication latencies approach their theoretical lower bounds.

2. SYSTEM MODEL

To maximize fault tolerance, BFT replication assumes what is essentially an adversarial failure model. Under this model, faulty nodes (servers or clients) may deviate from their intended behavior in arbitrary ways, representing problems such as hardware faults, software faults, node misconfigurations, or even malicious attacks. This model further assumes a strong adversary that can coordinate faulty nodes to compromise the replicated service. Note, however, that our model assumes the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures; we denote a message m signed by principal q's public key as $\langle m \rangle_{\sigma_q}$. Zyzzyva ensures its safety and liveness properties if at most f replicas are faulty, and it assumes a finite client population, any number of which may be faulty.

It makes little sense to build a system that can tolerate Byzantine replicas/servers[2] and clients but that can be corrupted by an unexpectedly slow node or network link, hence we design Zyzzyva so that its safety properties hold in any asynchronous distributed system where nodes operate at arbitrary speeds and are connected by a network that may fail to deliver messages, corrupt them, delay them, or deliver them out of order.

Unfortunately, ensuring both safety and liveness for consensus in an asynchronous distributed system is impossible if any server can crash [8], let alone if servers can be Byzantine. Zyzzyva's liveness, therefore, is ensured only during intervals in which messages sent to correct nodes are processed within some arbitrarily large fixed (but potentially unknown) worst case delay from when they are sent. This assumption appears easy to meet in practice if broken links are eventually repaired.

Zyzzyva implements a BFT service using state machine replication [16, 24]. Traditional state machine replication techniques can be applied only to deterministic services. Zyzzyva copes with the non-determinism present in many real-world applications such as file systems and databases using standard techniques to abstract the observable application state at the replicas and to resolve non-deterministic choices via the agreement stage [23].

If a client of a service issues an erroneous or malicious request, Zyzzyva's job is to ensure the request is processed consistently at all correct replicas; the replicated service, itself, is responsible for protecting its application state from such erroneous requests. Services typically limit the damage by authenticating clients and enforcing access control. For example, in a replicated file system, if a client tries to write a file without appropriate credentials, correct replicas could all process the request by returning an error code.

[2]We use the terms *replica* and *server* interchangeably.

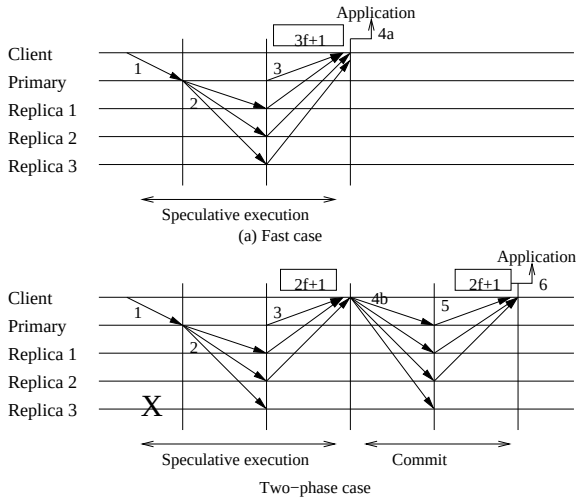

Figure 1: Protocol communication pattern for agreement within a view for (a) the fast case and (b) the two-phase faulty replica case. The numbers refer to the main steps of the protocol in the text.

3. AGREEMENT PROTOCOL

Zyzzyva is a state machine replication protocol executed by $3f+1$ replicas and based on three sub-protocols: (1) agreement, (2) view change, and (3) checkpoint. The *agreement* sub-protocol orders requests for execution by the replicas. Agreement operates within a sequence of *views*, and in each view a single replica, designated the *primary*, is responsible for leading the agreement sub-protocol. The *view change* sub-protocol coordinates the election of a new primary when the current primary is faulty or the system is running slowly. The *checkpoint* sub-protocol limits the state that must be stored by replicas and reduces the cost of performing view changes.

For simplicity, this article focuses on the agreement sub-protocol. The view change and execution sub-protocols are similar to those used previously [3, 26]. Interested readers may refer to Kotla et al. [11] for the full protocol.

Figure 1 shows the communication pattern for a single instance of Zyzzyva's agreement sub-protocol. In the fast, no-fault case (Figure 1-a), a client simply sends a request to the primary, the primary forwards the request to the replicas, and the replicas execute the request and send their responses to the client.

A request *completes* at a client when the client has a sufficient number of matching responses to ensure that all correct replicas will eventually execute the request and all preceding requests in the same order, thus guaranteeing that all correct replicas process the request in the same way, issue the same reply, and transition to the same subsequent system state. To allow a client to determine when a request *completes*, a client receives from replicas *responses* that include both an application-level *reply* and the *history* on which the reply depends. The *history* is the sequence of all requests executed

Label	Meaning
c	Client ID
CC	Commit certificate
d	Digest (cryptographic 1-way hash) of client request message: $d = H(m)$
i, j	Server IDs
h_n	History through sequence number n encoded as cryptographic 1-way hash: $h_n = H(h_{n-1}, d)$
m	Message containing client request
max_n	Max sequence number accepted by replica
n	Sequence number
ND	Selection of nondeterministic values needed to execute a request
o	Operation requested by client
OR	Order Request message
POM	Proof Of Misbehavior
r	Application reply to a client operation
t	Timestamp assigned to an operation by a client
v	View number

Table 1: Labels given to fields in messages.

by a replica prior to and including this request.

As Figure 1 illustrates, a request *completes* at a client in one of two ways. First, if the client receives $3f + 1$ matching responses (Figure 1-a), then the client considers the request *complete* and acts on it. Second, if the client receives between $2f + 1$ and $3f$ matching responses (Figure 1-b), then the client gathers $2f + 1$ matching responses and distributes this *commit certificate* to the replicas. A commit certificate includes cryptographic proof that $2f + 1$ servers agree on a linearizable order for the request and all preceding requests, and successfully storing a commit certificate to $2f+1$ servers (and thus at least $f+1$ correct servers) ensures that no other ordering can muster a quorum of $2f + 1$ servers to contradict this order. Therefore, once $2f + 1$ replicas acknowledge receiving a commit certificate, the client considers the request *complete* and acts on the corresponding reply.

Zyzzyva then ensures the following safety condition:

SAF If a request with sequence number n and history h_n completes, then any request that completes with a higher sequence number $n' \geq n$ has a history $h_{n'}$ that includes h_n as a prefix.

If fewer than $2f+1$ responses match, then to ensure liveness the client retransmits the request to all replicas at increasing intervals, and replicas demand that the primary order retransmitted requests. If the primary orders requests too slowly or orders requests inconsistently, a replica will suspect that the primary is faulty. If a sufficient number of replicas suspect that the primary is faulty, then a view change occurs and a new primary is elected.

Zyzzyva thereby ensures the following liveness condition assuming eventual synchrony[3] [7]:

LIV Any request issued by a correct client eventually completes.

In the rest of this section, we detail Zyzzyva's agreement subprotocol by considering three cases: (1) the *fast case* when all nodes act correctly and no timeouts occur, (2) the *two-phase case* that can occur when a non-primary replica is faulty or some timeouts occur, and (3) the *view change* case that can occur when the primary is faulty or more serious timeouts occur. Table 1 summarizes the labels we give fields in messages. Most readers will be happier if on their first reading they skip the text marked Additional Pedantic Details.

[3]In practice eventual synchrony can be achieved by using exponentially increasing timeouts [3].

3.1 Fast case

Figure 1-(a) illustrates the basic flow of messages in the fast case. We trace these messages through the system to explain the protocol, with the numbers in the figure corresponding to the numbers of major steps in the text. As the figure illustrates, the fast case proceeds in four major steps:

1. Client sends request to the primary.

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

4a. Client receives $3f + 1$ matching responses and completes the request.

To ensure correctness, the messages are carefully constructed to carry sufficient information to link these steps with one another and with past system actions. We now detail the contents of each message and describe the steps each node takes to process each message.

3.1.1 Message processing details

1. Client sends request to the primary.

A client c requests an operation o be performed by the replicated service by sending a message $\langle$REQUEST, o, t, $c\rangle_{\sigma_c}$ to the replica it believes to be the primary (i.e., the primary for the last response the client received).

Additional Pedantic Details: If the client guesses the wrong primary, the retransmission mechanisms discussed in step 4c below forwards the request to the current primary. The client's timestamp t is included to ensure exactly-once semantics of execution of requests [3].

2. Primary receives request, assigns sequence number, and forwards ordered request to replicas.

A view's primary has the authority to propose the order in which the system should execute requests. It does so by producing ORDER-REQ messages in response to client REQUEST messages.

In particular, when the primary p receives message $m =\langle$REQUEST, o, t, $c\rangle_{\sigma_c}$ from client c, the primary assigns to the request a sequence number n in the current view v and relays a message $\langle\langle$ORDER-REQ, v, n, h_n, d, $ND\rangle_{\sigma_p}$, $m\rangle$ to the backup replicas where n and v indicate the proposed sequence number and view number for m, digest $d = H(m)$ is the cryptographic one-way hash of m, $h_n = H(h_{n-1}, d)$ is a cryptographic hash summarizing the history, and ND is a set of values for non-deterministic application variables (time in file systems, locks in databases, etc.) required for executing the request.

Additional Pedantic Details: The primary only takes the above actions if $t > t_c$ where t_c is the highest timestamp previously received from c.

3. Replica receives ordered request, speculatively executes it, and responds to the client.

When a replica receives an ORDER-REQ message, it optimistically assumes that the primary is correct and that other correct repli-

cas will receive the same request with the same proposed order. It therefore speculatively executes requests in the order proposed by the primary and produces a SPEC-RESPONSE message that it sends to the client.

In particular, upon receipt of a message $\langle\langle$ORDER-REQ, $v, n, h_n, d, ND\rangle_{\sigma_p}, m\rangle$ from the primary p, replica i accepts the ordered request if m is a well-formed REQUEST message, d is a correct cryptographic hash of m, v is the current view, $n = max_n + 1$ where max_n is the largest sequence number in i's history, and $h_n = H(h_{n-1}, d)$. Upon accepting the message, i appends the ordered request to its history, executes the request using the current application state to produce a reply r, and sends to c a message $\langle\langle$SPEC-RESPONSE, $v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$ where $OR = \langle$ORDER-REQ, $v, n, h_n, d, ND\rangle_{\sigma_p}$.

Additional Pedantic Details: A replica may only accept and speculatively execute requests in sequence-number order, but message loss or a faulty primary can introduce holes in the sequence number space. Replica i discards the ORDER-REQ message if $n \leq max_n$. If $n > max_n + 1$, then i discards the message, sends a message $\langle$FILL-HOLE, $v, max_n+1, n, i\rangle_{\sigma_i}$ to the primary, and starts a timer. Upon receipt of a message $\langle$FILL-HOLE, $v, k, n, i\rangle_{\sigma_i}$ from replica i, the primary p sends a $\langle\langle$ORDER-REQ, $v, n', h_{n'}, d, ND\rangle_{\sigma_p}, m'\rangle$ to i for each request m' that p ordered in $k \leq n' \leq n$ during the current view; the primary ignores fill-hole requests from other views. If i receives the valid ORDER-REQ messages needed to fill the holes, it cancels the timer. Otherwise the replica broadcasts the FILL-HOLE message to all other replicas and initiates a view change when the timer fires. Any replica j that receives a FILL-HOLE message from i sends the corresponding ORDER-REQ message, if it has received one. If, in the process of filling in holes in the replica sequence, replica i receives conflicting ORDER-REQ messages, then the conflicting messages form a proof of misbehavior as described in protocol step 4d.

> 4a. Client receives $3f + 1$ matching responses and completes the request.

In the absence of faults and timeouts, the client receives matching SPEC-RESPONSE messages from all $3f + 1$ replicas. The client can then consider the request and its history to be *complete* and delivers the reply r to the application.

$3f + 1$ identical replies with identical histories suffice to ensure that a client can rely on a response. In particular, $3f + 1$ matching responses means all correct servers have executed the request and all preceding requests in the same order, so correct servers can always form a majority to vote to keep this response, even across view changes [11]. In particular, the view change sub-protocol executes across $2f + 1$ responsive servers, but any group of $2f + 1$ servers must include at least $f + 1$ correct servers and at most f faulty servers. Thus, the correct servers are always able to vote to keep this response, including both the application reply and the history of previous actions.

Therefore, upon receiving $3f+1$ distinct messages $\langle\langle$SPEC-RESPONSE, $v, n, h_n, H(r), c, t\rangle_{\sigma_i}, i, r, OR\rangle$, where i identifies the replica issuing the response, a client determines if they match. SPEC-RESPONSE messages from distinct replicas *match* if they have identical $v, n, h_n, H(r), c, t, OR$, and r fields.

3.2 Two-phase case

If the network, primary, or some replicas are slow or faulty, the client c may not receive matching responses from all $3f + 1$ replicas. The two-phase case applies when the client receives between $2f + 1$ and $3f$ matching responses. As Figure 1-(b) illustrates,

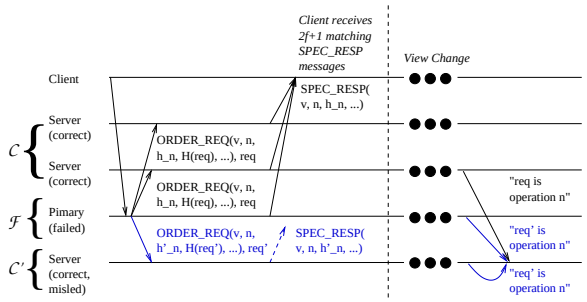

Figure 2: Example of a problem that could occur if a client were to rely on just $2f + 1$ matching responses without depositing a commit certificate with the servers.

steps 1-3 occur as described above, but step 4 is different:

> 4b. Client receives between $2f + 1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

The commit certificate is cryptographic proof that a majority of correct servers agree on the ordering of requests up to and including the client's request. Protocol steps 5 and 6 complete the second phase of agreement by ensuring that enough servers have this proof.

> 5. Replica receives a COMMIT message from a client containing a commit certificate and acknowledges with a LOCAL-COMMIT message.

> 6. Client receives a LOCAL-COMMIT messages from $2f + 1$ replicas and completes the request.

Again, the details of message construction and processing are designed to allow clients and replicas to link the system's actions together into a single linearizable history.

3.2.1 *Message processing details*

> 4b. Client receives between $2f + 1$ and $3f$ matching responses, assembles a commit certificate, and transmits the commit certificate to the replicas.

A client c sets a timer when it first issues a request. When this timer expires, if c has received matching speculative responses from between $2f + 1$ and $3f$ replicas, then c has a proof that a majority of correct replicas agree on the order in which the request should be processed. Unfortunately, the replicas, themselves, are unaware of this quorum of matching responses—they only know of their local decision, which may not be enough to guarantee that the request completes in this order.

Figure 2 illustrates the problem. A client receives $2f + 1$ matching speculative responses indicating that a request req was executed as the nth operation in view v. Let these responses come from $f + 1$ correct servers $\mathcal{C}$ and f faulty servers $\mathcal{F}$ and assume the remaining f correct servers $\mathcal{C}'$ received an ORDER-REQ message from a faulty primary proposing to execute a different request req' at sequence number n in view v. Suppose a view change occurs at this time. The view change sub-protocol must determine what requests were executed with what sequence numbers in view v so that the state in view $v + 1$ is consistent with the state in view v. Furthermore, since up to f replicas may be faulty, the view change sub-protocol must be able to complete using information from only

$2f + 1$ replicas. Suppose now that the $2f + 1$ replicas contributing state to a view change operation are one correct server from $\mathcal{C}$, f faulty servers from $\mathcal{F}$, and f correct but misled servers from $\mathcal{C}'$. In this case, only one of the replicas initializing the new view is guaranteed to vote to execute req as operation n in the new view, while as many as $2f$ replicas may vote to execute req' in that position. Thus, the system cannot ensure that view $v + 1$'s state reflects the execution of req as the operation with sequence number n.

Before client c can rely on this response, it must take additional steps to ensure the stability of this response. The client therefore sends a message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ where CC is a commit certificate consisting of a list of $2f + 1$ replicas, the replica-signed portions of the $2f + 1$ matching SPEC-RESPONSE messages from those replicas, and the corresponding $2f + 1$ replica signatures.

Additional Pedantic Details: CC contains $2f + 1$ signatures on the SPEC-RESPONSE message and a list of $2f + 1$ nodes, but, since all the responses received by c from replicas are identical, c only needs to include *one* replica-signed portion of the SPEC-RESPONSE message. Also note that, for efficiency, CC does not include the body r of the reply but only the hash $H(r)$.

5. Replica receives a COMMIT message from a client containing a commit certificate and acknowledges with a LOCAL-COMMIT message.

When a replica i receives a message $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$ containing a valid commit certificate CC proving that a request should be executed with a specified sequence number and history in the current view, the replica first ensures that its local history is consistent with the one certified by CC. If so, replica i (1) stores CC if CC's sequence number exceeds the stored certificate's sequence number and (2) sends a message $\langle \text{LOCAL-COMMIT}, v, d, h, i, c \rangle_{\sigma_i}$ to c.

Additional Pedantic Details: If the local history simply has holes encompassed by CC's history, then i fills them as described in 3. If, however, the two histories contain different requests for the same sequence number, then i initiates the view change sub-protocol. Note that as the view change protocol executes, correct replicas converge on a single common history, and those replicas whose local state reflect the "wrong" history (e.g., because they speculatively executed the "wrong" requests) restore their state from a cryptographically signed distributed checkpoint [11].

6. Client receives a LOCAL-COMMIT messages from $2f + 1$ replicas and completes the request.

The client resends the COMMIT message until it receives corresponding LOCAL-COMMIT messages from $2f + 1$ distinct replicas. The client then considers the request to be *complete* and delivers the reply r to the application.

$2f + 1$ local commit messages suffice to ensure that a client can rely on a response. In particular, at least $f + 1$ correct servers store a commit certificate for the response, and since any commit or view change requires participation by at least $2f + 1$ of the $3f + 1$ servers, any subsequent committed request or view change includes information from at least one correct server that holds this commit certificate. Since the commit certificate includes $2f + 1$ signatures vouching for the response, even a single correct server can use the commit certificate to convince all correct servers to accept this response (including the application reply and the history.)

Additional Pedantic Details: When the client first sends the COMMIT message to the replicas it starts a timer. If this timer expires before the client receives $2f + 1$ LOCAL-COMMIT messages then the client moves on to protocol steps described in the next subsection.

3.2.2 Client trust

At first glance, it may appear imprudent to rely on clients to transmit commit certificates to replicas (4b): what if a faulty client sends an altered commit certificate (threatening safety) or fails to send a commit certificate (imperiling liveness)?

Safety is ensured even if clients are faulty because commit certificates are authenticated by $2f + 1$ replicas. If a client alters a commit certificate, correct replicas will ignore it.

Liveness is ensured *for correct clients* because commit certificates are cumulative: successfully storing a commit certificate for request n at $2f+1$ replicas commits those replicas to a linearizable total order for all requests up to request n. So, if a faulty client fails to deposit a commit certificate, that client may not learn when its request *completes*, and a replica whose state has diverged from its peers may not immediately discover this fact. However, if at any future time a correct client issues a request, that request (and a linearizable history of earlier requests on which it depends) will either (i) complete via $3f + 1$ matching responses (4a), (ii) complete via successfully storing a commit certificate at $2f + 1$ replicas (4b-6), or (iii) trigger a view change (4c or 4d below).

3.3 Timeout and view change cases

Cases 4a and 4b allow a client c's request to complete with $2f + 1$ to $3f + 1$ matching responses. However, if the primary or network is faulty, c may not receive matching SPEC-RESPONSE or LOCAL-COMMIT messages from even $2f + 1$ replicas. Cases 4c and 4d therefore ensure that a client's request either completes in the current view or that a new view with a new primary is initiated. In particular, case 4c is triggered when a client receives fewer than $2f + 1$ matching responses and case 4c occurs when a client receives responses indicating inconsistent ordering by the primary.[4]

4c. Client receives fewer than $2f+1$ matching SPEC-RESPONSE messages and resends its request to all replicas, which forward the request to the primary in order to ensure the request is assigned a sequence number and eventually executed.

A client sets a second timer when it first issues a request. If the second timer expires before the request *completes*, the client suspects that the primary may not be ordering requests as intended, so it resends its REQUEST message through the remaining replicas so that they can track the request's progress and, if progress is not satisfactory, initiate a view change. This case can be understood by examining the behavior of a non-primary replica and of the primary.

Replica. When non-primary replica i receives a message $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ from client c, then if the request has a higher timestamp than the currently cached response for that client, i sends a message $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$ where $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ to the primary p and starts a timer. If the replica accepts an ORDER-REQ message for this request before the timeout, it processes the ORDER-REQ message as described above. If the timer fires before the primary orders the request, the replica initiates a view change.

Primary. Upon receiving the message $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$ from replica i, the primary p checks the client's timestamp for the request. If the request is new, p sends a new ORDER-REQ message using a new sequence number as described in step 2.

Additional Pedantic Details: If replica i does not receive the

[4]Note that cases 4b and 4c are not exclusive of 4d; a client may receive messages that are both sufficient to complete a request and also a proof a misbehavior against the primary.

ORDER-REQ message from the primary, the replica sends the CONFIRM-REQ message to all other replicas. Upon receipt of a CONFIRM-REQ message from another replica j, replica i sends the corresponding ORDER-REQ message it received from the primary to j; if i did not receive the request from the client, i acts as if the request came from the client itself. To ensure eventual progress, a replica doubles its current timeout in each new view and resets it to a default value if a view succeeds in executing a request.

Additionally, to retain exactly-once semantics, replicas maintain a cache that stores the reply to each client's most recent request. If a replica i receives a request from a client and the request matches or has a lower client-supplied timestamp than the currently cached request for client c, then i simply resends the cached response to c. Similarly, if the primary p receives an old client request from replica i, p sends to i the cached ORDER-REQ message for the most recent request from c. Furthermore, if replica i has received a commit certificate or stable checkpoint for a subsequent request, then the replica sends a LOCAL-COMMIT to the client even if the client has not transmitted a commit certificate for the retransmitted request.

> 4d. Client receives responses indicating inconsistent ordering by the primary and sends a proof of misbehavior to the replicas, which initiate a view change to oust the faulty primary.

If client c receives a pair of SPEC-RESPONSE messages containing valid messages $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_j}$ for the same request ($d = H(m)$) in the same view v with differing sequence number n or history h_n or ND, then the pair of ORDER-REQ messages constitutes a proof of misbehavior (POM) [2] against the primary. Upon receipt of a POM, c sends a message $\langle \text{POM}, v, POM \rangle_{\sigma_c}$ to all replicas. Upon receipt of a valid POM message, a replica initiates a view change and forwards the POM message to all other replicas.

4. EVALUATION

This section examines the performance of Zyzzyva and compares it with existing approaches. We run our experiments on 3.0 GHz Pentium-4 machines with the Linux 2.6 kernel. We use MD5 for hashing and UMAC [3] for message authentication codes (MACs). MD5 is known to be vulnerable, but we use it to make our results comparable with those in the literature. Since Zyzzyva uses fewer MACs per request than any of the competing algorithms, our advantages over other algorithms would be increased if we were to use the more secure, but more expensive, SHA-256.

For comparison, we run Castro et al.'s implementation of PBFT [3] and Cowling et al.'s implementation of HQ [6]; we scale up HQ's measured throughput for the small request/response benchmark by 9% to account for their use of SHA-1 rather than MD5. We include published throughput measurements for Q/U [1]; we scale Q/U's reported performance up by 7.5% to account for our use of 3.0 GHz rather than 2.8GHz machines. We also compare against the measured performance of an unreplicated server.

To stress-test Zyzzyva we use the micro-benchmarks devised by Castro et al. [3]. In the 0/0 benchmark, clients send null requests and receive null replies. In the 4/0 benchmark, clients send 4KB requests and receive a null replies. In the 0/4 benchmark, clients send null requests and receive 4KB replies. In all experiments, we configure all BFT systems to tolerate $f = 1$ faults; we examine performance for other configurations elsewhere [11].

In the preceding sections, we describe a simplified version of the protocol. In our extended paper [12], we detail a number of optimizations, all implemented in the prototype measured here, that (1) reduce encryption costs by replacing public key signatures with message authentication codes (MACs) [3], (2) improve throughput by agreeing on the order of batches of requests [3], (3) reduce the impact of lost messages by caching out-of-order messages, (4) improve read performance by optimizing read-only requests [3], reduce bandwidth by allowing most replicas to send hashes rather than full replies to clients [3], (5) improve the performance of Zyzzyva's two-phase case by using a commit optimization in which replicas use client hints to initiate and complete the second phase to commit the request before they execute the request and send the response (with the committed history) back to the client, and (6) reduce overheads by including MACs only for a preferred quorum [6]. In the extended paper we also describe Zyzzyva5, a variation of the protocol that requires $5f + 1$ agreement replicas but that improves performance in the presence of faulty replicas by completing in three one-way message exchanges as in Figure 1(a) even when up to f non-primary replicas are faulty.

In the following experiments, unless noted otherwise, we use all of the optimizations other than preferred quorums for Zyzzyva. PBFT [3] does not implement the preferred quorum optimization, but HQ does [6]. We do not use the read-only optimization for Zyzzyva and PBFT unless we state so explicitly.

4.1 Cost model

Our evaluation focuses on three metrics that BFT replication must optimize to be practical for a broad range of services: replication cost, throughput, and latency. Before we dive into experimental evaluation in the following sections, Table 2 puts our results in perspective by providing a high-level analytic model of of Zyzzyva and of several other recent BFT protocols. The table also shows lower bounds on BFT state machine replication overheads for each of these dimensions.

In the first row of the table body, replication *cost* refers to the number of replicas required to construct a system that tolerates f Byzantine faults. The importance of minimizing this metric for practical services is readily apparent. We show two values: *replicas with application state* indicates the number of replicas that must both participate in the coordination protocol and also maintain application state for executing application requests. Conversely, *total replicas* indicates the total number of machines that must participate in the protocol including, for some protocols, "witness nodes" that do not maintain application state or execute application requests. This distinction is important because witness nodes may be simpler or less expensive than nodes that must also execute requests to run the replicated service.

Zyzzyva and PBFT (with Yin et al.'s optimization for separating agreement and execution [26]) meet the replication cost lower bounds of $2f + 1$ application replicas (so a majority of nodes are correct) [24] and $3f + 1$ total replicas (so agreement on request order can be reached) [21].

In the next row of the table body, *throughput* is determined by the processing overhead per request. Our simple model focuses on CPU-intensive cryptographic operations. All of the systems we examine use Castro's MAC authenticator construct [3] to avoid using expensive asymmetric cryptography operations.

The (trivial) lower bound on processing overhead is for each server to process two MAC operations per client request: one to verify the client's request and one to authenticate its reply. Zyzzyva and PBFT approach this bound by using a *batching* optimization in which the primary accumulates a batch of b client requests and leads agreement on that batch rather than on each individual request. Zyzzyva's speculative execution allows it to avoid several

		PBFT [3]	Q/U [1]	HQ [6]	Zyzzyva	State Machine Repl. Lower Bound
Cost	Total replicas	**3f+1**	5f+1	**3f+1**	**3f+1**	**3f+1** [21]
	Replicas with application state	**2f+1** [26]	5f+1	3f+1	**2f+1**	**2f+1** [24]
Throughput	MAC ops at bottleneck server	2+(8f+1)/b	2+8f	4+4f	**2+3f/b**	**2**†
Latency	NW 1-way latencies on critical path	4	**2**	4	**3**	**2/3**‡

Table 2: Properties of state-of-the-art and optimal Byzantine fault tolerant service replication systems tolerating f faults, using MACs for authentication [3], and using a batch size of b [3]. Bold entries denote protocols that match known lower bounds or those with the lowest known cost. †It is not clear that this trivial lower bound is achievable. ‡The distributed systems literature typically considers 3 one-way latencies to be the lower bound for agreement on client requests [17]; 2 one-way latencies is achievable if no request contention is assumed.

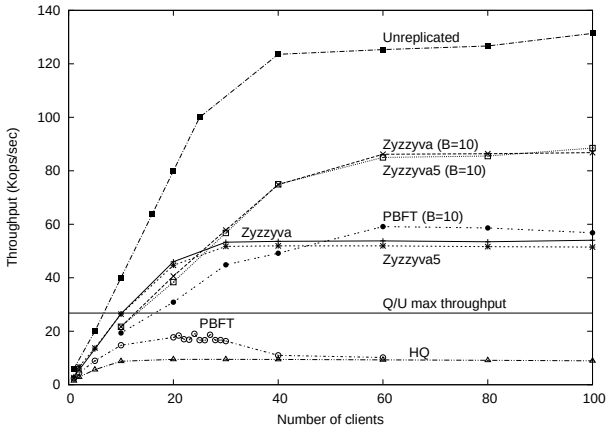

Figure 3: Realized throughput for the 0/0 benchmark as the number of client varies. Q/U throughput is scaled from [1].

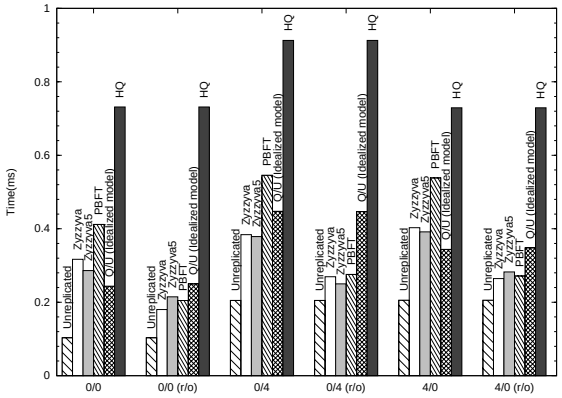

Figure 4: Latency for 0/0, 0/4, and 4/0 benchmarks.

rounds of all-to-all communication among servers, so it requires fewer MAC operations per batch than PBFT.

In the last row of the table body, *latency* counts the number of one-way message delays from when a client issues a request until it receives a reply. In the general case, agreement requires 3 message delays [17], and Zyzzyva matches this bound by having requests go from the client to the primary to the replicas to the client. Q/U circumvents this bound by optimizing for the case of no request contention so that requests go directly from the client to the replicas to the client. We chose to retain the extra hop through the primary in Zyzzyva because it facilitates batching, which we consider to be an important throughput optimization.

The models described in this subsection focus on what we regard as important factors for understanding the performance trade-offs of different algorithms, but they necessarily omit details present in implementations. Also, as is customary [1, 3, 6, 23, 26], Table 2 compares the protocols' performance during the optimized case of fault-free, timeout-free execution. In the rest of this section we experimentally examine these protocols' throughput, latency, and performance during failures.

4.2 Throughput

Figure 3 shows the throughput measured for the 0/0 benchmark for Zyzzyva, Zyzzyva5 [11], PBFT, and HQ (scaled as noted above). For reference, we also show the peak throughput reported for Q/U [1] in the $f = 1$ configuration, scaled to our environment as described above.

Zyzzyva executes over 50K requests per second without batching, and this number rises to 86K requests per second when batching is activated with 10 requests per batch. As the figure illustrates, Zyzzyva enjoys a significant throughput advantage over the other protocols.

It is also worth noting that when batching is enabled, Zyzzyva's throughput is within 35% of the throughput of an unreplicated server that simply replies to client requests over an authenticated channel. Furthermore, this gap would fall if the service being replicated were more demanding than the null service examined here. Overall, we speculate that Zyzzyva's throughput is sufficient to support BFT replication for a broad range of demanding services.

4.3 Latency

Figure 4 shows the latencies of Zyzzyva, Zyzzyva5, HQ, and PBFT for the 0/0, 0/4, and 4/0 workloads with a single client issuing one request at a time. We examine both the default read/write requests that use the full protocol and read-only requests that can exploit Zyzzyva and PBFT's read-only optimization [3].

We did not succeed in getting Abd-El-Malek et al.'s implementation of Q/U running in our environment. However, because Table 2 suggests that Q/U may have a latency advantage over other protocols, for comparison we implement an idealized model of Q/U designed to provide an optimistic estimate of Q/U's latency in our environment. In our idealized implementation, a client simply generates and sends $4f + 1$ MACs with a request, each replica verifies $4f + 1$ MACs (1 to authenticate the client and $4f + 1$ to validate the reported state), each replica in a preferred quorum [6] generates and sends $4f + 1$ MACs (1 to authenticate the reply to the client

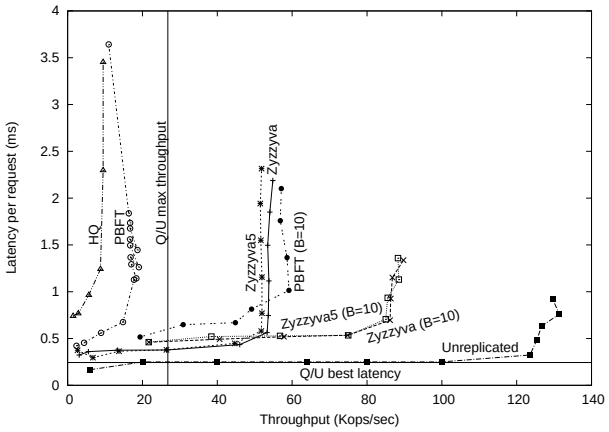

Figure 5: Latency vs. throughput for the 0/0 benchmark. Q/U throughput is scaled from [1]. *Q/U best latency* is the measured latency for our idealized model implementation of Q/U under low offered load.

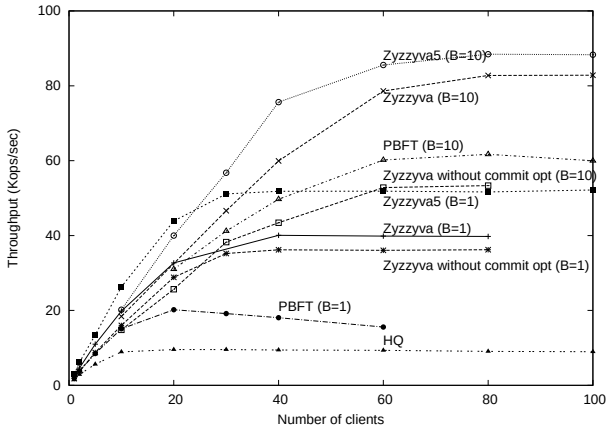

Figure 6: Realized throughput for the 0/0 benchmark as the number of clients varies when $f = 1$ non-primary replicas fail to respond to requests.

and $4f$ to authenticate the new state) with a reply to the client, and the client verifies $4f + 1$ MACs.

For the read/write 0/0 and 4/0 benchmarks, Q/U does have a modest latency advantage over Zyzzyva as predicted by Table 2. For the read-only benchmarks, the situation is reversed with Zyzzyva exhibiting modestly lower latency than Q/U because Zyzzyva's read-only optimization completes read-only requests in two message delays (like Q/U) but uses fewer cryptographic operations.

Figure 5 shows latency and throughput as we vary offered load for the 0/0 benchmark. As the figure illustrates, batching in Zyzzyva, Zyzzyva5, and PBFT increases latency but also increases peak throughput. Adaptively setting the batch size in response to workload characteristics is an avenue for future work.

Overall, all of the BFT protocols do add service latency compared to an unreplicated server, but Zyzzyva is generally competitive with the best protocols by this metric. We speculate that the additional 120 to 250 microseconds that Zyzzyva requires compared to an unreplicated server will be a significant barrier for only the most demanding services, and we note that the relative gap would shrink for services that do more than execute the null request.

4.4 Performance during failures

Zyzzyva guarantees correct execution with any number of faulty clients and up to f faulty replicas. However, its performance is optimized for the case of failure-free operation, and a single faulty replica can force Zyzzyva to execute the slower two-phase protocol.

One solution is to buttress Zyzzyva's fast 1-phase path by employing additional servers. Zyzzyva5 [11] uses a total of $5f + 1$ servers ($2f + 1$ full replicas and $3f$ additional witnesses) to allow the system to complete requests via the fast communication pattern shown in Figure 1-(a) when the client receives $4f + 1$ (out of $5f + 1$) matching replies.

Surprisingly, however, even when running with $3f + 1$ replicas, Zyzzyva remains remains competitive with existing protocols even when it is falls back on two-phase operation. In particular, Zyzzyva's cryptographic overhead at the bottleneck replica increases from $2 + \frac{3f+1}{b}$ to $3 + \frac{5f+1}{b}$ operations per request if we simply execute the two-phase algorithm described above. Furthermore, our implementation also includes a *commit optimization* [12] that reduces cryptographic overheads to $2 + \frac{5f+1}{b}$ cryptographic operations per request by having replicas that suspect a faulty primary initiate and complete the second phase to commit the request before they execute the request and send the response (with the committed history) back to the client.

Figure 6 compares throughputs of Zyzzyva, Zyzzyva5, PBFT, and HQ in the presence of f non-primary-server fail-stop failures. We do not include a discussion of Q/U in this section as the throughput numbers of Q/U with failures are not reported [1], but we would not expect a fail-stop failure by a replica to significantly reduce the performance shown for Q/U in Figure 3. Also, we do not include a line for the unreplicated server case as the throughput falls to zero when the only server suffers a fail-stop failure.

As Figure 6 shows, without the commit optimization, falling back on two-phase operation reduces Zyzzyva's maximum throughput from 86K requests per second (Figure 3) to 52K requests per second. Despite this extra overhead, Zyzzyva's "slow case" performance remains within 13% of PBFT's performance, which is less highly tuned for the failure-free case and which suffers no slowdown in this scenario. Zyzzyva's commit optimization repairs most of the damage caused by a fail-stop replica, maintaining a throughput of 82K requests/second which is within 5% of the peak throughput achieved for the failure free case. For systems that can afford extra witness replicas, Zyzzyva5's throughput is not significantly affected by the fail-stop failure of a replica, as expected.

5. RELATED WORK

Zyzzyva stands on the shoulders of recent efforts that have dramatically cut the costs and improved the practicality of BFT replication. Castro and Liskov's Practical Byzantine Fault Tolerance (PBFT) protocol [3] devised techniques to eliminate expensive signatures and potentially fragile timing assumptions, and it demonstrated high throughputs of over ten thousand requests per second. This surprising result jump started an arms race in which researchers reduced replication costs [26], and improved performance [1, 6, 13] of BFT service replication. Zyzzyva incorporates many of the ideas developed in these protocols and folds in the new idea of speculative execution to construct an optimized fast path that significantly outperforms existing protocols and that has replication cost, processing overhead, and latency that approach the theoretical minima for these metrics.

Numerous BFT agreement protocols [3, 6, 13, 18, 23, 26] have

used *tentative execution* to reduce the latency experienced by clients. This optimization allows replicas to execute a request tentatively as soon as they have collected the equivalent of a Zyzzyva commit certificate for that request. This optimization may appear similar to Zyzzyva's support for *speculative execution*, but there are two fundamental differences. First, Zyzzyva's speculative execution allows requests to complete at a client after a single phase, without the need to compute a commit certificate: this reduction in latency is not possible with traditional tentative executions. Second, and more importantly, in traditional BFT systems a replica can execute a request tentatively only after the replica knows that all previous requests have been committed. In Zyzzyva, replicas continue to execute requests speculatively, without waiting to know that requests with lower sequence numbers have completed. This difference is what lets Zyzzyva leverage speculation to achieve not just lower latency but also higher throughput.

Speculator [20] allows clients to speculatively complete operations at the application level and perform client level rollback. A similar approach could be used in conjunction with Zyzzyva to support clients that want to act on a reply optimistically rather than waiting on the specified set of responses.

Zyzzyva's focus is on maximizing the peak performance of BFT replication. Conversely, Clement et al. [5] argue that BFT systems should seek not only to ensure safety but also good performance in the presence of Byzantine faults, and their Aardvark system eliminates *fragile optimizations* that maximize best-case performance but that can allow a faulty client or server to drive the system down expensive execution paths.

6. CONCLUSION

By systematically exploiting speculation, Zyzzyva exhibits significant performance improvements over existing BFT protocols. The throughput overheads and latency of Zyzzyva approach the theoretical lower bounds for any BFT state machine replication protocol.

Looking forward, although we expect continued progress in improving the performance (for example, by making additional assumptions about the application characteristics) and robustness (in the presence of broader range of failure scenarios) of BFT replication, we believe that Zyzzyva demonstrates that BFT overheads should no longer be regarded as a barrier to using BFT replication for even many highly demanding services.

7. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CNS-0720649, CNS-0509338, and CNS-0411026. We thank Rodrigo Rodrigues, James Cowling, and Michael Abd-El-Malek for sharing source code for PBFT, HQ, and Q/U respectively. We are grateful for Maurice Herlihy's feedback on earlier drafts of this article.

8. REFERENCES

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. SOSP*, Oct. 2005.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. of SOSP '05*, pages 45–58, Oct. 2005.

[3] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, Nov. 2002.

[4] Amazon s3 availability event: July 20, 2008. http://status.aws.amazon.com/s3-20080720.html.

[5] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. Making Byzantine fault tolerant services tolerate Byzantine faults. Technical Report 08-27, UT Austin Dept. of Computer Sciences, May 2008.

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

[7] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 1988.

[8] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[9] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proceedings of 19th ACM Symp. on Operating Systems Principles*, October 2003.

[10] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Prog. Lang. Sys.*, 12(3), 1990.

[11] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *ACM Symposium on Operating Systems Principles*, 2007.

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

[13] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *DSN*, June 2004.

[14] R. Kotla, M. Dahlin, and L. Alvisi. SafeStore: A durable and practical storage system. In *USENIX07*, June 2007.

[15] Lamport, Shostak, and Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, July 1982.

[16] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, Apr. 1984.

[17] L. Lamport. Lower bounds for asynchronous consensus. In *Proc. FUDICO*, pages 22–23, June 2003.

[18] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. *IEEE TODSC*, 3(3):202–215, July 2006.

[19] E. Nightingale, K. Veeraraghavan, P. Chen, and J. Flinn. Rethink the sync. In *Proc. OSDI*, 2006.

[20] E. B. Nightingale, P. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. SOSP*, Oct. 2005.

[21] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2), April 1980.

[22] V. Prabhakaran, L. Bairavasundaram, N. Agrawal, H. G. A. Arpaci-Dusseau, and R. Arpaci-Dusseau. IRON file systems. In *Proc. SOSP*, 2005.

[23] R. Rodrigues, M. Castro, and B. Liskov. BASE : Using abstraction to improve fault tolerance. In *Proc. SOSP*, October 2001.

[24] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4), 1990.

[25] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proc. OSDI*, 2006.

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