Tolerating latency in replicated state machines through client speculation

Benjamin Wester[⋆] James Cowling[†] Edmund B. Nightingale[⋄]
Peter M. Chen[⋆] Jason Flinn[⋆] Barbara Liskov[†]

University of Michigan[⋆] *MIT CSAIL*[†] *Microsoft Research*[⋄]

Abstract

Replicated state machines are an important and widely-studied methodology for tolerating a wide range of faults. Unfortunately, while replicas should be distributed geographically for maximum fault tolerance, current replicated state machine protocols tend to magnify the effects of high network latencies caused by geographic distribution. In this paper, we examine how to use speculative execution at the clients of a replicated service to reduce the impact of network and protocol latency. We first give design principles for using client speculation with replicated services, such as generating early replies and prioritizing throughput over latency. We then describe a mechanism that allows speculative clients to make new requests through replica-resolved speculation and predicated writes. We implement a detailed case study that applies this approach to a standard Byzantine fault tolerant protocol (PBFT) for replicated NFS and counter services. Client speculation trades in 18% maximum throughput to decrease the effective latency under light workloads, letting us speed up run time on single-client micro-benchmarks 1.08–19× when the client is co-located with the primary. On a macro-benchmark, reduced latency gives the client a speedup of up to 5×.

1 Introduction

As more of society depends on services running on computers, tolerating faults in these services is increasingly important. Replicated state machines [34] provide a general methodology to tolerate a wide variety of faults, including hardware failures, software crashes, and malicious attacks. Numerous examples exist for how to build such replicated state machines, such as those based on agreement [8, 11, 22, 25] and those based on quorums [1, 11].

For replicated state machines to provide increased fault tolerance, the replicas should fail independently. Various aspects of failure independence can be achieved by using multiple computers, independently written software [2, 33], and separate administrative domains. Geographic distribution is one important way to achieve failure independence when confronted with failures such as power outages, natural disasters, and physical attacks.

Unfortunately, distributing the replicas geographically increases the network latency between replicas, and many protocols for replicated state machines are highly sensitive to latency. In particular, protocols that tolerate Byzantine faults must wait for multiple replicas to reply, so the effective latency of the service is limited by the latency of the slowest replica being waited for. Agreement-based protocols further magnify the effects of high network latency because they use multiple message rounds to reach agreement. Some implementations may also choose to delay requests and batch them together to improve throughput.

Our work uses speculative execution to allow clients of replicated services to be less sensitive to high latencies caused by network delays and protocol messages. We observe that faults are generally rare, and, in the absence of faults, the response from even a single replica is an excellent predictor of the final, collective response from the replicated state machine. Based on this observation, clients in our system can proceed after receiving the first response, thereby hiding considerable latency in the common case in which the first response is correct, especially if at least one replica is located nearby. When responses are completely predictable, clients can even continue before they receive any response.

To provide safety in the rare case in which the first response is incorrect, a client in our system may only continue executing *speculatively*, until enough responses are collected to confirm the prediction. By tracking all effects of the speculative execution and not externalizing speculative state, our system can undo the effects of the speculation if the first response is later shown to be incorrect.

Because client speculation hides much of the latency of the replicated service from the client, replicated

servers in our system are freed to optimize their behavior to maximize their throughput and minimize load, such as by handling agreement in large batches.

We show how client speculation can help clients of a replicated service tolerate network and protocol latency by adding speculation to the Practical Byzantine Fault Tolerance (PBFT) protocol [8]. We demonstrate how performance improves for a counter service and an NFSv2 service on PBFT from decreased effective latency and increased concurrency in light workloads. Speculation improves the client throughput of the counter service 2–58× across two different network topologies. Speculation speeds up the run time of NFS micro-benchmarks 1.08–19× and up to 5× on a macro-benchmark when co-locating a replica with the client. When replicas are equidistant from each other, our benchmarks speed up by 1.06–6× and 2.2×, respectively. The decrease in latency that client speculation provides does have a cost: under heavy workloads, maximum throughput is decreased by 18%.

We next describe our general approach to adding client speculation to a system with a replicated service.

2 Client speculation in replicated services

2.1 Speculative execution

Speculative execution is a general latency-hiding technique. Rather than wait for the result of a slow operation, a computer system may instead predict the outcome of that operation, checkpoint its state, and speculatively execute further operations using the predicted result. If the speculation is correct, the checkpoint is committed and discarded. If the speculation is incorrect, it is aborted, and the system rolls back its state to the checkpoint and re-executes further operations using the correct result.

In general, speculative execution is beneficial only if the time to checkpoint state is less than the time to perform the operation that generates the result. Further, the outcome of that operation must be predictable. Incorrect speculations waste resources since all work that depends on a mispredicted result is thrown away. This waste lowers throughput, especially when multiple entities are participating in a distributed system, since the system might have been able to service other entities in lieu of doing work for the incorrect speculation. Thus, the decision of whether or not to speculate on the result of an operation often boils down to determining which operations will be slow and which slow operations have predictable results.

2.2 Applicability to replicated services

Replicated services are an excellent candidate for client-based speculative execution. Clients of replicated state machine protocols that tolerate Byzantine faults must wait for multiple replicas to reply. That may mean waiting for multiple rounds of messages to be exchanged among replicas in an agreement-based protocol. If replicas are separated by geographic distances (as they should be in order to achieve failure independence), network latency introduces substantial delay between the time a client starts an operation and the time the client receives the reply that commits the operation. Thus, there is substantial time available to benefit from speculative execution, especially if one replica is located near the client.

Replicated services also provide an excellent predictor of an operation's result. Under the assumption that faults are rare, a client's request will generate identical replies from every replica, so the first reply that a client receives is an excellent predictor of the final, collective reply from the replicated state machine (which we refer to as the *consensus reply*). After receiving the first reply to any operation, a client can speculate *based on 1 reply* with high confidence. For example, when an NFS client tries to read an uncached file, it cannot predict what data will be returned, so it must wait for the first reply before it can continue with reasonable data.

The results of some remote operations can be predicted even before receiving any replies; for instance, an NFS client can predict with high likelihood of success that file system updates will succeed and that read operations will return the same (possibly stale) values in its cache [28]. For such operations, a client may speculate *based on 0 replies* since it can predict the result of a remote operation with high probability.

2.3 Protocol adjustments

Based on the above discussion, it becomes clear that some replicated state machine protocols will benefit more from speculative execution than others. For this reason, we propose several adjustments to protocols that increase the benefit of client-based speculation.

2.3.1 Generate early replies

Since the maximum latency that can be hidden by speculative execution, in the absence of 0-reply speculation, is the time between when the client receives the first reply from any replica and when the client receives enough replies to determine the consensus response, a protocol should be designed to get the first reply to the client as quickly as possible. The fastest reply is realized when the client sends its request to the closest replica, and that replica responds immediately. Thus, a protocol that supports client speculation should have one or more replicas immediately respond to a client with the replica's best guess for the final outcome of the operation, as long as that guess can accurately predict the consensus reply.

Assuming each replica stores the complete state of the service, the closest replica can always immediately perform and respond to a read-only request. However, that reply is not guaranteed to be correct in the presence of

concurrent write operations. It could be wrong if the closest replica is behind in the serial order of operations and returns a stale value, or in quorum protocols where the replica state has diverged and is awaiting repair [1]. We describe optimizations in Section 3.2.2 that allow early responses from any replica in the system, along with techniques to minimize the likelihood of an incorrect speculative read response.

It is more difficult to allow any replica to immediately execute a modifying request in an agreement protocol. Backup replicas depend on the primary replica to decide a single ordering of requests. Without waiting for that ordering, a backup could guess at the order, speculatively executing requests as it receives them. However, it is unlikely that each replica will perceive the same request ordering under workloads with concurrent writers, especially with geographic distribution of replicas. Should the guessed order turn out wrong (beyond acceptable levels [23]), the replica must roll back its state and re-execute operations in the committed order, hurting throughput and likely causing its response to change.

For agreement protocols like PBFT, a more elegant solution is to have only the primary execute the request early and respond to the client. As we explain in Section 3.3, such predictions are correct unless the primary is faulty. This solution enables us to avoid speculation or complex state management on the replicas that would reduce throughput. Used in this way, the primary should be located near the most active clients in a system to reduce their latency.

2.3.2 Prioritize throughput over latency

There exist a myriad of replicated state machine protocols that offer varying trade-offs between throughput and latency [1, 8, 11, 22, 30, 32, 37]. Given client support for speculative execution, it is usually best to choose a protocol that improves throughput over one that improves latency. The reason is that speculation can do much to hide replica latency but little to improve replica throughput.

As discussed in the previous section, speculative execution can hide the latency that occurs between the receipt of an early reply from a replica and the receipt of the reply that ends the operation. Thus, as long as a speculative protocol provides for early replies from the closest or primary replica, reducing the latency of the overall operation does not ordinarily improve user-perceived latency.

Speculation can only improve throughput in the case where replicas are occasionally idle by allowing clients to issue more operations concurrently. If the replicas are fully loaded, speculation may even decrease throughput because of the additional work caused by mispredictions or the generation of early replies. Thus, it seems prudent to choose a protocol that has higher latency but higher potential throughput, perhaps through batching, and stable performance under write contention [8, 22], rather than protocols that optimize latency over throughput [1, 11].

An important corollary of this observation is that client speculation allows one to choose simpler protocols. With speculation, a complex protocol that is highly optimized to reduce latency may perform approximately the same as a simpler, higher latency protocol from the viewpoint of a user. A simpler protocol has many benefits, such as allowing a simpler implementation that is quicker to develop, is less prone to bugs, and may be more secure because of a smaller trusted computing base.

2.3.3 Avoid speculative state on replicas

To ensure correctness, speculative execution must avoid *output commits* that externalize speculative output (e.g., by displaying it to a user) since such output can not be undone once externalized. The definition of what constitutes external output, however, can change. For instance, sending a network message to another computer would be considered an output commit if that computer did not support speculation. However, if that computer could be trusted to undo, if necessary, any changes that causally depend on the receipt of the message, then the message would not be an output commit. One can think of the latter case as enlarging the *boundary of speculation* from just a single computer to encompass both the sender and receiver.

What should be the boundary of speculation for a replicated service? At least three options are possible: allow all replicas and clients of the service to share speculative state, allow replicas to share speculative state with individual clients but not to propagate one client's speculative state to other clients, and disallow replicas from storing speculative state.

Our design uses the third option, with the smallest boundary of speculation, for several reasons. First, the complexity of the system increases as more parts participate in a speculation. The system would need to use distributed commit and rollback [14] to involve replicas and other clients in the speculation, and the interaction between such a distributed commit and the normal replicated service commit would need to be examined carefully. Second, as the boundary of speculation grows larger, the cost of a misprediction is higher; all replicas and clients that see speculative state must roll back all actions that depend on that state when a prediction is wrong. Finally, it may be difficult to precisely track dependencies as they propagate through the data structures of a replica, and any false dependencies in a replica's state may force clients to trust each other in ways not required by the data they share in the replicated service.

For example, if the system takes the simple approach of tainting the entire replica state, then one client's misprediction would force the replica to roll back all later operations, causing unrelated clients to also roll back.

2.3.4 Use replica-resolved speculation

Even with this small boundary of speculation, we would still like to allow clients to issue new requests that depend on speculative state (which we call *speculative requests*). Speculative requests allow a client to continue submitting requests when it would otherwise be forced to block. These additional requests can be handled concurrently, increasing throughput when the replicas are not already fully saturated.

One complication here is that, to maintain correctness, if one of the prior operations on which the client is speculating fails, any dependent operations that the client issues must also abort. There is currently no mechanism for a replica to determine whether or not a client received a correct speculative response. Thus, the replica is unable to detect whether or not to execute subsequent dependent speculative requests.

To overcome this flaw, we propose *replica-resolved speculation through predicated writes*, in which replicas are given enough information to determine whether the speculations on which requests depend will commit or abort. With predicated writes, an operation that modifies state includes a list of the active speculations on which it depends, along with the predicted responses for those speculations. Replicas log each committed response they send to clients and compare each predicted response in a predicated write with the actual response sent. If all predicated responses match the saved versions, the speculative request is consistent with the replica's responses, and it can execute the new request. If the responses do not match, the replica knows that the client will abort this operation when rolling back a failed speculation, so it discards the operation. This approach assumes a protocol in which all non-faulty replicas send the same response to a request.

Note that few changes may need to be made to a protocol to handle speculative requests that modify data. An operation O that depends on a prior speculation O_s, with predicted response r, may simply be thought of as a single deterministic request to the replicated service of the predicated form: `if` $response(O_s) = r$, `then do` O. This predicate must be enforced on the replicas. However, as shown in Section 5, predicate checking may be performed by a shim layer between the replication protocol and the application without modifying the protocol itself.

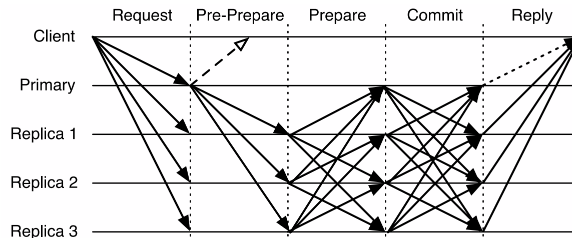

Figure 1: PBFT-CS Protocol Communication. The early response from the primary is shown with a dashed hollow arrow, which replaces its response from the Reply phase (dotted filled arrow) in PBFT.

3 Client speculation for PBFT

In this section, we apply our general strategy for supporting client speculative execution in replicated services to the Practical Byzantine Fault Tolerance (PBFT) protocol. We call the new protocol we develop PBFT-CS (CS denotes the additional support for client speculation).

3.1 PBFT overview

PBFT is a Byzantine fault tolerant state machine replication protocol that uses a primary replica to assign each client request a sequence number in the serial order of operations. The replicas run a three-phase agreement protocol to reach consensus on the ordering of each operation, after which they can execute the operation while ensuring consistent state at all non-faulty replicas. Optionally, the primary can choose and attach *nondeterministic data* to each request (for NFS, this contains the current time of day).

PBFT requires $3f + 1$ replicas to handle f concurrent faulty replicas, which is the theoretical minimum [5]. The protocol guarantees liveness and correctness with up to f failures, and runs a *view change* sub-protocol to move the primary to another replica in the case of a bad primary.

The communication pattern for PBFT is shown in Figure 1. The client normally receives a commit after five one-way message delays, although this may be shortened to four delays by overlapping the *commit* and *reply* phases using a *tentative execution* optimization [8]. To reduce the overhead of the agreement protocol, the primary may collect a number of client requests into a *batch* and run agreement once on the ordering of operations within this batch.

In our modified protocol, PBFT-CS, the primary responds immediately to client requests, as illustrated by the dashed line in Figure 1.

3.2 PBFT-CS base protocol

In both PBFT and PBFT-CS, the client sends each request to all replicas, which buffer the request for execu-

tion after agreement. Unlike the PBFT agreement protocol, the primary in PBFT-CS executes an operation immediately upon receiving a request and sends the early reply to the client as a speculative response. The primary then forms a pre-prepare message for the next batch of requests and continues execution of the agreement protocol. Other replicas are unmodified and reply to the client request once the operation has committed.

Since the primary determines the serial ordering of all requests, under normal circumstances the client will receive at least f committed responses from the replicas matching the primary's early response. This signifies that the speculation was correct because the request committed with the same value as the speculative response. If the client receives $f + 1$ matching responses that differ from the primary's response, the client rolls back the current speculation and resumes execution with the consensus response.

3.2.1 Predicated writes

A PBFT-CS client can issue subsequent requests immediately after predicting a response to an earlier request, rather than waiting for the earlier request to commit. To enable this without requiring replicas themselves to speculate and potentially roll back, PBFT-CS ensures that a request that modifies state does not commit if it depends on the value of any incorrect speculative responses. To meet this requirement, clients must track and propagate the dependencies between requests.

For example, consider a client that reads a value stored in a PBFT-CS database (`op1`), performs some computation on the data, then writes the result of the computation back to the database (`op2`). If the primary returns an incorrect speculative result for `op1`, the value to be written in `op2` will also be incorrect. When `op1` eventually commits with a different value, the client will fail its speculation and resume operation with the correct value. Although the client cannot undo the send of `op2`, dependency tracking prevents `op2` from writing its incorrect value to the database.

Each PBFT-CS client maintains a log of the digests d_T of each speculative response issued at logical timestamp T. When an operation commits, its corresponding digest is removed from the tail of the log. If an operation aborts, its digest is removed from the log, along with the digests of any dependent operations.

Clients append any required dependencies to each speculative request, of the form $\{c, \langle t_i, d_i \rangle, ...\}$ for client c and each digest d_i at timestamp t_i.

Replicas also store a log of digests for each client with the committed response for each operation. The replica executes a speculative request only if all digests in the request's dependency list match the entries in the replica's log. Otherwise, the replica executes a no-op in place of the operation.

It is infeasible for replicas to maintain an unbounded digest log for each client in a long-running system, so PBFT-CS truncates these logs periodically. Replicas must make a deterministic decision on when to truncate their logs to ensure that non-faulty replicas either all execute the operation or all abort it. This is achieved by truncating the logs at fixed deterministic intervals.

If a client issues a request containing a dependency that has since been discarded from the log, the replicas abort the operation, replacing it with a no-op. The client recognizes this scenario when receiving a consensus response that contains a special *retry* result. It retries execution once all its dependencies have committed. In practice an operation will not abort due to missing dependencies, provided that the log is sufficiently long to record all operations issued in the time between a replica executing an operation and a quorum of responses being received by the client.

3.2.2 Read-only optimization

Many state machine replication protocols provide a read-only optimization [1, 8, 11, 22] in which read requests can be handled by each replica without being run through the agreement protocol. This allows reads to complete in a single communication round, and it reduces the load on the primary.

In the standard optimization, a client issues optimized read requests directly to each replica rather than to the primary. Replicas execute and reply to these requests without taking any steps towards agreement. A client can continue after receiving $2f + 1$ matching replies. Because optimized reads are not serialized through the agreement protocol, other clients can issue conflicting, concurrent writes that prevent the client from receiving enough matching replies. When this happens, the client retransmits the request through the agreement protocol. This optimization is beneficial to workloads that contain a substantial percentage of read-only operations and exhibit few conflicting, concurrent writes. Importantly, when a backup replica is located nearer a client than the primary, that replica's reply will typically be received by the client before the primary's.

PBFT-CS cannot use this standard optimization without modification. A problem arises when a client issues a speculative request that depends on the predicted response to an optimized read request. PBFT-CS requires all non-faulty replicas to make a deterministic decision when verifying the dependencies on an operation. However, since optimized reads are *not* serialized by the agreement protocol, one non-faulty replica may see a conflicting write before responding to an optimized read, while another non-faulty replica sees the write after responding to the read. These two non-faulty replicas will

thus respond to the optimized read with different values, and they will make different decisions when they verify the dependencies on a later speculative request. A non-faulty replica that sent a response that matches the first speculative response received by the client will commit the write operation, while other non-faulty replicas will not. Hence, writes may not depend on uncommitted optimized reads. This is enforced at each replica by not logging the response digest for such requests.

We address this problem by allowing a PBFT-CS client to resubmit optimized read requests through the full agreement protocol, forcing the replicas to agree on a common response. When write conflicts are low, the resubmitted read is likely to have the same reply as the initial optimized read, so a speculative prediction is likely to still be correct. After performing this procedure, we can send any dependent write requests, as they no longer depend on an optimized request.

There are three issues that must be considered for a read request to be submitted using this optimization.

- The request cannot read uncommitted state.
- The client should not follow a read with a write.
- The reply should not be completely predictable.

The first issue is required for consistency. A client cannot optimize a read request for a piece of state before all its write requests for that state are committed. Otherwise, it risks reading stale data when a sufficient number of backup replicas have not yet seen the client's previous writes. The data dependency tracking required to implement this policy is also used to propagate speculations, so no extra information needs to be maintained. Reads that do depend on uncommitted data may still be submitted through the agreement protocol as with write requests. Should a client desire a simpler policy for ensuring correctness, it can disable the read-only optimization while it has any uncommitted writes.

Second, consider a client that reads a value, performs a computation, and then writes back a new value. If the read request is initially sent optimized, issuing the write will force the read to be resubmitted. The "optimization" results in additional work. Clients that anticipate following a read by a write should decline to optimize the read.

Finally, if a client can predict the outcome of the request before receiving any replies (for instance, if it predicts that a locally-cached value has not become stale), then it should submit the request through the normal agreement protocol. Since the client does not need to wait for any replies, it is not hurt by the extra latency of waiting for agreement.

3.3 Handling failures

Speculation optimizes for reduced latency in the non-failure case, but it is important to ensure that correctness and liveness are maintained in the presence of faulty replicas. Failed speculations also increase the latency of a client's request, forcing it to roll back after having waited for the consensus response, and hurt throughput by forcing outstanding requests to become no-ops. It is important for our protocol to handle faults correctly in a way that still tries to preserve performance.

A speculation will fail on a client when the first reply it receives to a request does not match the consensus response. There are three cases in which this might happen:

- The most common case occurs when a write issued by another client conflicts with an optimized read. In an extreme instance, one replica's early reply could contain the stale data while all other replicas reply with current data.
- The second case occurs when there is a view change. PBFT ensures that committed requests will be ordered the same in the new view, but the client is speculating on uncommitted requests that the new replica could order differently. View changes may be the result of a bad primary, or they may be triggered by network conditions or proactive recovery [9].
- The third case occurs when the primary is faulty, and it either returns an incorrect speculative response or serializes a request differently when running the agreement protocol. We next examine this scenario further.

It is trivial for a client to detect a faulty primary: a request's early reply from the primary and the consensus reply will be in the same view and not match. If signed responses are used, the primary's bad reply can be given to other replicas as a proof of misbehavior. However, if simple message authentication codes (MACs) are used, the early reply cannot be used in this way since MACs do not provide non-repudiation.

The simplest solution to handling faults with MACs is for a client to stop speculating if the percentage of failed speculations it observes surpasses a threshold. PBFT-CS currently uses an arbitrary threshold of 1%. If a client observes that the percentage of failed speculations is greater than 1% over the past n early replies provided by a replica, it simply ceases to speculate on subsequent early replies from that replica. Although it will not speculate on subsequent replies, it can still track their accuracy and resume speculating on further replies if the percentage falls below a threshold. Our experimental results verify that at this threshold, PBFT-CS is still effective at reducing the average latency under light workloads.

3.4 Correctness

The speculative execution environment and PBFT protocol used in our system both have well-established correctness guarantees [7, 28]. We thus focus our attention

on the modifications made to PBFT, to ensure that this protocol remains correct.

Our modified version of PBFT differs from the original in several key ways:

- A client may be sent a speculative response that differs from the final consensus value.
- A client may submit an operation that depends on a failed speculation.
- The primary may execute an operation before it commits.

We evaluate each modification independently.

Incorrect speculation A bad primary may send an incorrect speculative response to a client, in that it differs on the value or ordering of the final consensus value. We also consider in this class an honest primary that sends a speculative response to a client but is unable to complete agreement on this response due to a view change. In either case, the client will only see the consensus response once the operation has undergone agreement at a quorum of replicas. If the speculative response was incorrect, it is safe for the client to roll back the speculative execution and re-run using the consensus value, since PBFT ensures that all non-faulty replicas will agree on the consensus value.

Dependent operations A further complication arises when the client has issued subsequent requests that depend on the value of a speculative response. Here, the speculation protocol on the client ensures that it rolls back execution of any operations that have dependencies on the failed speculation. We must ensure that all valid replicas make an identical decision to abort each dependent operation by replacing it with a no-op.

Replicas maintain a log of the digests for each committed operation and truncate this log at deterministic intervals so that all non-faulty replicas have the same log state when processing a given operation. Predicated writes in PBFT-CS allow the client to express the speculation dependencies to the replicas. A non-faulty replica will not execute any operation that contains a dependency that does not match the corresponding digest in the log, or that does not have a matching log entry. Since the predicated write contains the same information used by the client when rolling back dependent operations, the replicas are guaranteed to abort any operation aborted by the client. If a client submits a dependency that has since been truncated from the log, it will also be aborted.

The only scenario where replicas are unable to deterministically decide whether a speculative response matches its agreed-upon value is when a speculative response was produced using the read-only optimization. Here, different replicas may have responded with different values to the read request. We explicitly avoid this case by making it an error to send a write request that depends on the reply to an optimized read request; correct clients will never issue such a request. Replicas do not store the responses to optimized reads in their log and hence always ignore any request sent by a faulty client with a dependency on an optimized read.

Speculative execution In our modified protocol, the primary executes client requests immediately upon their receipt, before the request has undergone agreement. The agreement protocol dictates that all non-faulty replicas commit operations in the order proposed by the primary, unless they execute a view change to elect a new primary. After a view change, the new primary may reorder some uncommitted operations executed by the previous primary, however, the PBFT view change protocol ensures that any committed operations persist into the new view. It is safe for the old primary to restore its state to the most recent committed operation since any incorrect speculative response will be rolled back by clients where necessary.

4 Discussion and future optimizations

In this section, we further explore the protocol design space for the use of client speculation with PBFT. We compare and contrast possible protocol alternatives with the PBFT-CS protocol that we have implemented.

4.1 Alternative failure handling strategies

We considered two alternative strategies for dealing with faulty primaries. First, we could allow clients to request a view change without providing a proof of misbehavior. This scheme would seem to significantly compromise liveness in a system containing faulty clients since they can force view changes at will. However, this is an existing problem in BFT state machine replication in the absence of signatures. A bad client in PBFT is always able to force a view change by sending a request to the primary with a bad authenticator that appears correct to the primary or by sending different requests to different replicas [7]. We could mitigate the damage a given bad client can do by having replicas make a local decision to ignore all requests from a client that 'framed' them. In this way a bad client can not initiate a view change after incriminating f primaries.

Alternatively, we could require signatures in communications between client and replicas. This is the most straight-forward solution, but entails significant CPU overhead. Compared to these two alternative designs, we chose to have PBFT-CS revert to a non-speculative protocol due to the simplicity of the design and higher performance in the absence of a faulty primary.

4.2 Coarse-grained dependency tracking

PBFT-CS tracks and specifies the dependencies of a speculative request at fine granularity. Thus, message

size and state grow as the average number of dependencies for a given operation increases. To keep message size and state constant, we could use coarser-grained dependencies.

We could track dependencies on a per-client basis by ensuring that a replica executes a request from a client at logical timestamp T only if *all* outstanding requests from that client prior to time T have committed with the same value the client predicted.

Instead of maintaining a list of dependencies, each client would instead store a hash chained over all consensus responses and subsequent speculative responses. The client would append this hash to each operation in place of the dependency list. The client would also keep another hash chained only over consensus responses, which it would use to restore its dependency state after rolling back a failed speculation.

Each replica would maintain a hash chained over responses sent to the client and would execute an operation if the hash chain in the request matches its record of responses. Otherwise, it would execute a no-op.

We chose not to use this optimization in PBFT-CS since the use of chained hashes creates dependencies between all operations issued by a client even when no causal dependencies exist. This increases the cost of a failed speculation since the failure of one speculative request causes all subsequent in-progress speculative operations to abort. Coarse-grained dependency tracking also limits the opportunities for running speculative read operations while there are active speculative writes. Since speculative read responses are not serialized with respect to write operations, it is likely that the client will insert the read response in the wrong point in the hash chain, causing subsequent operations to abort.

4.3 Reads in the past

A read-only request need not circumvent the agreement protocol completely, as described in section 3.2.2. A client can instead take a hybrid approach for non-modifying requests: it can submit the request for full agreement and at the same time have the nearest replica immediately execute the request.

If the primary happens to be the nearest to the client, this is not a change from the normal protocol. When another replica is closer, the client can get a lower-latency first reply, plus having agreement eliminates the second consideration for optimized reads (in Section 3.2.2), that a client should not follow a read with a write.

However, this new optimization presents a problem when there are concurrent writes by multiple clients. A non-primary replica will execute an optimized request, and a client will speculate on its reply, in a sequential order that is likely different from the request's actual order in the agreement protocol. In essence, the read has been executed *in the past*, at a logical time when the replicas have not yet processed all operations that are undergoing agreement but when they still share a consistent state.

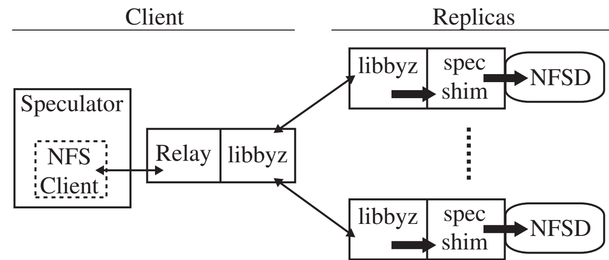

Figure 2: Speculative fault-tolerant NFS architecture

We could extend the PBFT-CS read-only optimization to also allow reads in the past. Under a typical configuration, there is only one round of agreement executing at any one time, with incoming requests buffered at the primary to run in the next batch of agreement. If we were to ensure that all buffered reads are reordered, when possible, to be serialized at the start of this next batch, it would be highly likely that no write will come between a read being received by a replica and the read being serialized after agreement.

Note that the primary may assign any order to requests within a batch as long as no operation is placed before one on which it depends. Recall that a PBFT-CS client will only optimize a read if the read has no outstanding write dependencies. Hence, the primary is free to move all speculative reads to the start of the batch. The primary executes these requests on a snapshot of the state taken before the batch began.

5 Implementation

We modified Castro and Liskov's PBFT library, *libbyz* [8], to implement the PBFT-CS protocol described in Section 3. We also modified BFS [8], a Byzantine-fault-tolerant replicated file service based on NFSv2, to support client speculation. The overall system can be divided into three parts as shown in Figure 2: the NFS client, a protocol relay, and the fault-tolerant service.

5.1 NFS client operation

Our client system uses the NFSv2 client module of the Speculator kernel [28], which provides process-level support for speculative execution. Speculator supports fine-grained dependency tracking and checkpointing of individual objects such as files and processes inside the Linux kernel. Local file systems are speculation-aware and can be accessed without triggering an output commit. Speculator buffers external output to the terminal, network, and other devices until the speculations on which they depend commit. Speculator rolls back pro-

cess and OS state to checkpoints and restarts execution if a speculation fails.

To execute a remote NFS operation, Speculator first attaches a list of the process's dependencies to the message, then sends it to a relay process on the same machine. The relay interprets this list and attaches the correct predicates when sending the PBFT-CS request.

The relay brokers communication between the client and replicas. It appears to be a standard NFS server to the client, so the client need not deal with the PBFT-CS protocol. When the relay receives the first reply to a 1-reply speculation, the reply is logged and passed to the waiting NFS client. The NFS client recognizes speculative data, creates a new speculation, and waits for a confirmation message from the relay. Once the consensus reply is known, the relay sends either a `commit` message or a `rollback{reply}` message containing the correct response.

Our implementation speculates based on 0 replies for `GETATTR`, `SETATTR`, `WRITE`, `CREATE`, and `REMOVE` calls. It can speculate on 1 reply for `GETATTR`, `LOOKUP`, and `READ` calls. This list includes the most common NFS operations: we observed that at least 95% of all calls in all our benchmarks were handled speculatively. Note that we speculate on both 0 replies and 1 reply for `GETATTR` calls. The kernel can speculate as soon as it has attributes for a file. When the attributes are cached, 0 replies are needed, otherwise, the kernel waits for 1 reply before continuing.

5.2 PBFT-CS client operation

Speculation hides latency by allowing a single client to pipeline many requests; however, our PBFT implementation only allows for each PBFT-CS client to have a single outstanding request at any time. We work around this limitation by grouping up to 100 logical clients into a single client process.

NFS with 0-reply speculation requires its requests to be executed in the order they were issued. A PBFT-CS client process can tag each request with a sequence number so that the primary replica will only process requests from that client process's logical clients in the correct order. Of course, two different clients' requests can still be interleaved in any order by the primary.

To support this additional concurrency, we designed the client to use an event-driven API. User programs pass requests to libbyz and later receive two callbacks: one delivers the first reply and another delivers the consensus reply. The user program is responsible for monitoring libbyz's communication channels and timers.

5.3 Server operation

On the replicas, libbyz implements an event-based server that performs upcalls into the service when needed: to re-

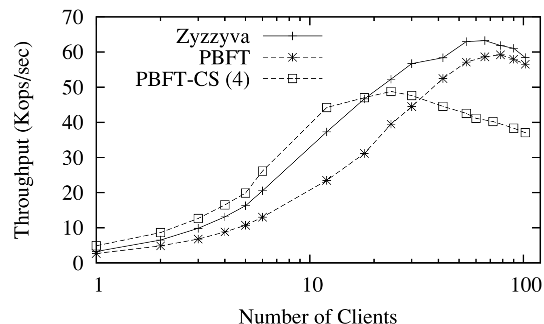

Figure 3: Server throughput in a LAN, measured on the **shared counter** service. PBFT-CS (4) is limited to four concurrent requests.

Overhead Source	Slowdown
Early replies	8.2%
Larger request	4.1%
Complex client	2.8%
Predicate checking	1.8%

Table 1: Major sources of overhead affecting throughput for PBFT-CS relative to PBFT.

quest non-deterministic data, to execute requests, and to construct error replies. The library handles all communication and state management, including checkpointing and recovery.

A shim layer is used to manage dependencies on replicas. When writes need to be quashed due to failed speculative dependencies, the shim layer issues a no-op to the service instead. Thus, the underlying service is not exposed to details of the PBFT-CS protocol.

The primary will batch together all requests it receives while it is still agreeing on earlier requests. Batching is a general optimization that reduces the number of protocol instances that must be run, decreasing the number of communications and authentication operations [8, 22, 23, 37]. This implementation imposes a maximum batch size of 64 requests, a limit our benchmarks do run up against.

6 Evaluation

In this section, we quantify the performance of our PBFT-CS implementation. We have implemented a simple shared counter micro-benchmark and several NFS micro- and macro-benchmarks.

We compare PBFT-CS against two other Byzantine fault-tolerant agreement protocols: PBFT [8] and Zyzzyva [22]. PBFT is the base protocol we extend make use of client speculation. Its overall structure is illustrated in Figure 1. We use the tentative reply optimization, so each request must go through 4 communication phases before the client acquires a reply that it can act on.

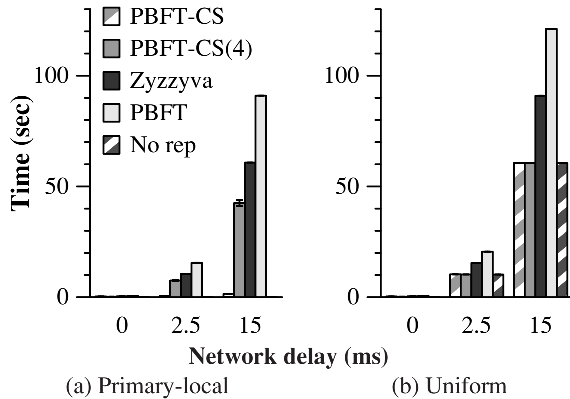

Figure 4: Time taken to run 2000 updates using the **shared counter** service. The primary-local topology (a) shows a client located at the same site as the primary. The uniform topology (b) shows a remote client equidistant from all sites. 0 ms (LAN) times for both graphs are (in bar order): 0.36 s, 0.27 s, 0.41 s, 0.54 s, and 0.16 s.

PBFT uses an adaptive batching protocol, allowing up to 64 requests to be handled in one agreement instance.

Zyzzyva is a recent agreement protocol that is heavily optimized for failure-free operation. When all replicas are non-faulty (as in our experiments), it takes only 3 phases for a client to possess a consensus reply. We run Kotla et al.'s implementation of Zyzzva, which uses a fixed batch size. We simulate an adaptive batching strategy by manually tuning the batch size as needed for best performance.

By comparison, a PBFT-CS client can continue executing speculatively after only 2 communication phases. We expect this to significantly reduce the effective latency of our clients. Note that requests still require 4 phases to *commit*, but we can handle those requests concurrently rather than sequentially. If we limit the number of in-flight requests to some number n, we call the protocol "PBFT-CS (n)."

6.1 Experimental setup

Each replica machine uses a single Intel Xeon 2.8 GHz processor with 512 MB RAM (sufficient for our applications). We always evaluate using four replicas without failures (unless noted). In our NFS comparisons, we use a single client that is identical in hardware to the replicas. Our counter service runs on an additional five client machines using Intel Pentium 4s or Xeons with clock speeds of 3.06–3.20 GHz and 1 GB RAM. All systems use a generic Red Hat Linux 2.4.21 kernel.

Our machines use gigabit Ethernet to communicate directly with a single switch. Experiments using the shared counter service were performed on a Cisco Catalyst 2970 gigabit switch; NFS used an Intel Express ES101TX 10/100 switch.

Our target usage scenario is a system that consists of several sites joined by moderate latency connections (but slower than LAN speeds). Each site has a high-speed LAN hosting one replica and several clients, and clients may also be located off-site from any replica. For comparison with other agreement protocols, we also consider using PBFT-CS in a LAN setting where all replicas and clients are on the same local segment.

Based on the above scenarios, we emulate a simplified test network using NISTNet [6] that inserts an equal amount of one-way latency between each site. We let this inserted *delay* be either 2.5 ms or 15 ms.

We also measure performance at clients located in different areas in our scenario. In the *primary-local topology*, the client is at the same site as the current primary replica. The *primary-remote topology* considers a client at different site hosting a backup replica. A client not present at any site is shown in the *uniform topology*, and we let the client have the same one-way latency to all replicas as between sites.

When comparing against a service with no replication in a given topology, we always assume that a client at a site can access its server using only the LAN. A client not at a site is still subject to added delay.

6.2 Counter throughput

We first examine the throughput of PBFT-CS using the counter service. Similar to Castro and Liskov's standard 0/0 benchmark [8], the counter's request and reply size are minimal. This service exposes only one operation: increment the counter and return its new value. Each reply contains a token that the client must present on its next request. This does add a small amount of processing time to each request, but it ensures that client requests must be submitted sequentially.

Our client is a simple loop that issues a fixed number of counter updates and records the total time spent. No state is externalized by the client, so we allow the client process to implement its own lightweight checkpoint mechanism. Checkpoint operations take negligible time, so our results focus on the characteristics of the protocol itself rather than our checkpoint mechanism.

We measure throughput by increasing the number of client processes per machine (up to 17 processes) until the server appears saturated. Graphs show the mean of at least 6 runs, and visible differences are statistically significant.

Figure 3 shows the measured throughput in a LAN configuration. We found that in this topology, a single PBFT-CS client gains no benefit from having more than 4 concurrent requests, and we enforce that limit on all clients. When we have 12 or fewer concurrent clients, PBFT-CS has 1.19–1.49× higher through-

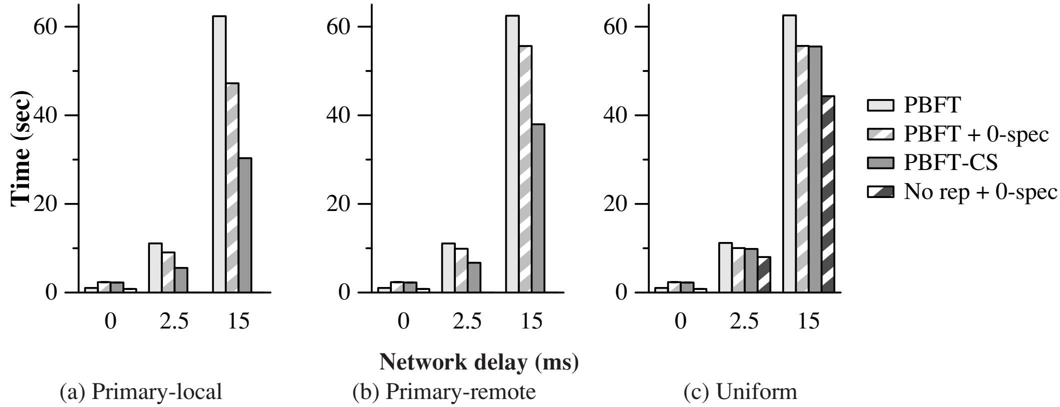

Figure 5: **Read-only** NFS micro-benchmark performance across different network topologies. The last three data sets use 0-reply speculation. At 0 ms, all three topologies are equivalent, so the same data is used for each graph. The *no rep* data show a lower bound for run time. There is only one *no rep* data set for primary-local and primary remote topologies, because the location of the server does not change with increasing latency. For these two graphs, the 0 ms bar applies to all latencies but is not repeated.

put than Zyzzyva and 1.79–2× higher throughput than PBFT.

In lightly loaded systems, the servers are not being fully utilized, and speculating clients can take advantage of the spare resources to decrease their own effective latency. As the server becomes more heavily loaded, those resources are no long free to use. As a result, PBFT-CS reaches its peak throughput before other protocols.

There is a trade-off of throughput for latency: PBFT-CS shows a peak throughput that is 17.6% lower than PBFT. We found four fundamental sources of overhead, summarized in Table 1. First, the client implementation for PBFT-CS uses an event-driven system to handle several logical clients, needed to support concurrent requests. This design does lead to a slower client than the one in PBFT, which can get by with a simpler blocking design. Second, we found that having the primary send early replies increases its time spent blocking while transmitting. Third, each predicate added to a request makes the request packet larger, and fourth, those predicates take additional work to verify on each replica.

6.3 Counter latency

We next examine how latency affects client performance under a light workload when the client is located at different sites. Figure 4 shows the time taken for a single counter client to issue 2000 requests in different topologies. In the LAN topology where no delay is added, a PBFT-CS client is able to complete the benchmark in 33% less time than PBFT, reflecting average run times of 357 ms and 538 ms respectively. When we increase the latency between sites, run time becomes dominated by number of communication phases. With a uniform topology (Figure 4b), PBFT-CS takes 50% less time than PBFT and 33% less time than Zyzzyva, and its runtime is only 1% slower than the unreplicated service. This matches our intuitive understanding of the protocol behavior described at the start of this section.

For PBFT-CS, the critical path is a round-trip communication with the primary replica. Moving to a primary-remote topology (bringing one backup replica closer) does not affect this critical path, and our measurements show no significant difference between primary-remote and uniform topologies.

Figure 4a presents results when using a primary-local topology. As latency increases and backup replicas move further from the client, performance does not degrade significantly, since the latency to the primary is fixed. At 15 ms latency, a client using PBFT takes 58× longer than with PBFT-CS. The combination of client speculation and a co-located primary achieves much of the performance benefit of a closely located non-replicated server, while providing all the guarantees of a geographically distributed replicated service that tolerates Byzantine faults.

These significant gains are directly attributable to the increased concurrency possible in the primary-local topology. When we limit PBFT-CS to only 4 outstanding requests, the client must then wait on requests to commit, reintroducing a dependence on communication delay. In topologies where the client does not have privileged access to the primary, as in the uniform topology, limiting concurrency has little effect.

6.4 NFS

We next examine PBFT-CS applied to an NFS server. Considering that the NFSv2 protocol is not explicitly designed for high-latency environments, we compare

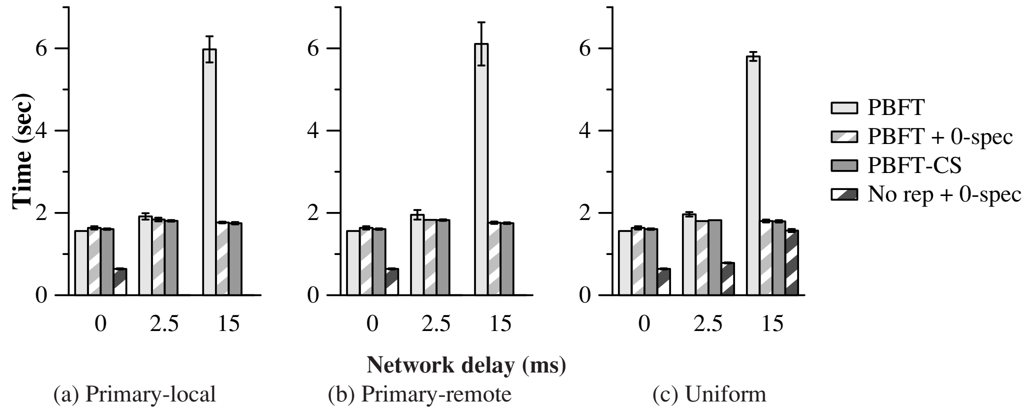

Figure 6: **Write-only** NFS micro-benchmark.

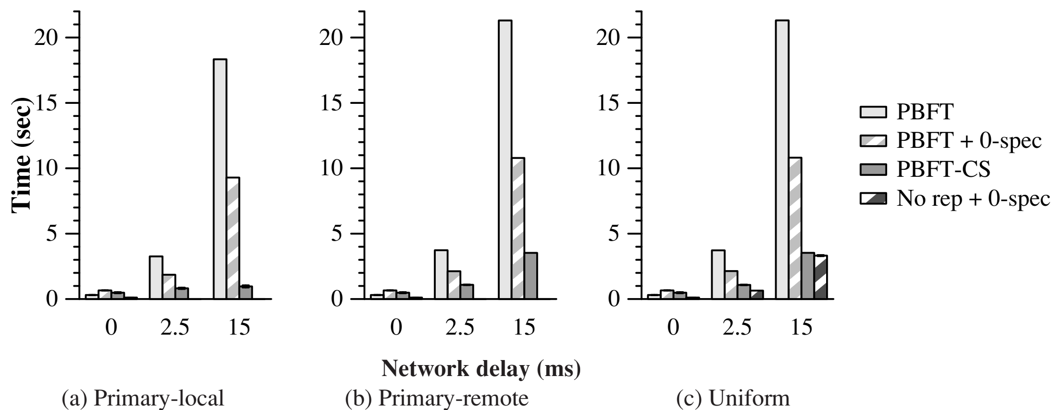

Figure 7: **Read/Write** NFS micro-benchmark.

against the variation of NFS that uses 0-reply speculation. All benchmarks begin with a freshly-mounted file system and an empty cache.

Unlike the counter service, this application has overhead associated with creating, committing, and rolling back to a checkpoint. Processes may have computation to perform between requests, and they may need to block before an output commit.

For comparison with non-speculative systems, we measure the performance of NFS under PBFT. Using our speculative NFS protocol, we measure PBFT using only 0-reply speculation (*PBFT + 0-spec*) and PBFT-CS. The difference between these two measurements show the benefit of 1-reply speculation. As a lower bound, we also measure the performance of a non-replicated NFS server that uses 0-reply speculation (*No rep + 0-spec*).

We use a vanilla kernel for evaluating non-speculative PBFT with a slight modification that increases the number of concurrent RPC requests allowed. Other benchmarks use the Speculator kernel.

In the *no replication* configuration, the NFS client uses a thin UDP relay on the local machine that stands in for the BFT relay.

Our modifications to the NFS client, the relay, and the replicated service have introduced additional overhead that is not present in the original PBFT. This inefficiency is particularly apparent in our 0 ms topologies, where PBFT-CS shows a 1.03–2.18× slowdown relative to PBFT across all our benchmarks. However, in all cases at higher latencies, client speculation results in a clear improvement, and we primarily address these configurations in the following sections.

At the time of publication, we had not yet ported our NFS server to use the Zyzzyva protocol, so we regretfully are unable to provide a direct comparison for these benchmarks.

All graphs show the mean of at least five measurements. Error bars are shown when the 95% confidence interval is above 1% of the mean value.

6.5 NFS: Read-only micro-benchmark

We first ran a read-only micro-benchmark that `greps` for a common string within the Linux headers. The total size of the searched files is about 9.1 MB. Most requests

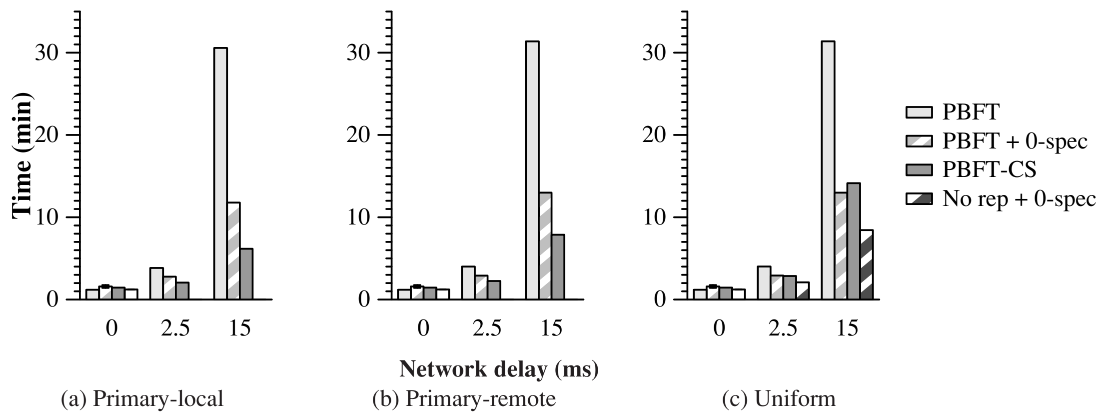

Figure 8: The **Apache build** NFS benchmark measures how long it takes to compile and link Apache 2.0.48.

in this benchmark are read-only and are optimized to circumvent agreement.

Figure 5 shows that PBFT takes 2.06× longer to complete than PBFT-CS at 15 ms. 0-reply speculation lets the client avoid blocking when revalidating a file after opening it. With PBFT-CS, we can additionally read from a file without delay: a nearby replica supplies all the speculative data. Without a nearby replica (in uniform topology), 1-reply speculation is not beneficial since optimized reads complete at about the same time the client gets its first reply.

6.6 NFS: Write-only micro-benchmark

We next ran a write-only micro-benchmark that writes 3.9 MB into an NFS file (Figure 6). All writes are issued asynchronously by the file system, and the client only blocks when the file is closed. In this case, speculation is not needed to increase the parallelism of the system.

There are a very small number of read requests in this benchmark, issued when first opening a file, so there is no practical opportunity to use 1-reply speculation. Speculation at 2.5 ms reduces the benchmark run time by only 6–7%. We found that within each latency (irrespective of topology), there is no statistical difference between PBFT+0-spec and PBFT-CS.

6.7 NFS: Read/write micro-benchmark

We next ran a read/write micro-benchmark that creates 100 4 KB files in a directory. For each file, the client creates and writes to a file; this includes read-only operations to read the directory entries. PBFT-CS never blocks on any of these operations.

In the primary-local topology, PBFT takes up to 19× longer to complete than PBFT-CS (Figure 7). Furthermore, PBFT-CS shows a resilience to changes in latency as it increases from 0-15 ms: PBFT-CS execution time doubles while PBFT takes 59× longer. On the primary-remote and uniform topologies, operations take longer to complete, but client speculation still speeds up run time by 6.03×.

6.8 NFS: Apache build macro-benchmark

Finally, we ran a benchmark that compiles and links Apache 2.0.48. This emulates the standard Andrew-style benchmark that has been widely used in the PBFT literature. This is intended to model a realistic and common workload, where speculation allows significant computation to be overlapped with I/O.

Within the primary-local topology, PBFT takes up to 5.0× longer to complete than PBFT-CS (Figure 8). In the uniform topology, PBFT takes up to 2.2× longer than PBFT-CS. Since files are often reused many times during the build process, there is less opportunity to benefit from 1-reply speculation. However, the relative difference in performance degradation as latency increases is still significant. With a co-located primary, PBFT-CS becomes 4.3× slower as delay increases to 15 ms, while PBFT slows down by a factor of 25.

6.9 Cost of failure / faulty primary

To measure the cost of speculation failures, we modified our PBFT-CS relay to inject faulty digests into early replies, simulating a primary that returns corrupted replies at a rate of 1%. Any speculation based on a corrupted reply will eventually be rolled back, and any dependent requests will be turned into no-ops on good replicas.

The results of this experiment are presented in Figure 9. We used the Apache build benchmark in the primary-local topology. The injected faults were responsible for slowdowns in PBFT-CS of 3%, 9%, and 29% at 0 ms, 2.5 ms, and 15 ms delay respectively.

These slowdowns are not identical because a client may have a greater number of requests in the pipeline for completion at a 15 ms delay than at a 0 ms delay. When one request fails, nearly all outstanding requests

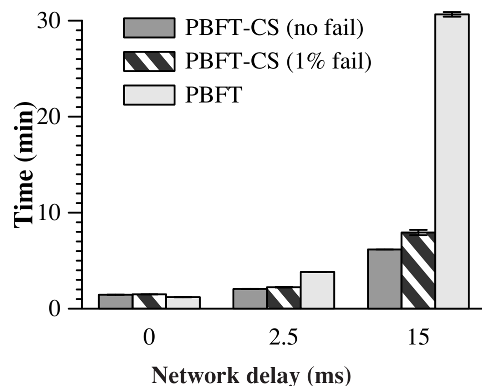

Figure 9: For the Apache build benchmark in the primary-local topology, PBFT-CS is at worst 29% slower when 1% of its speculations fail.

also fail. We observed that 1% of our speculations failed directly, and an additional 1%, 4%, and 5% of speculations (at 0 ms, 2.5 ms, and 15 ms respectively) failed due to their dependencies. These extra requests added unnecessary load to the replicas. By executing more requests in advance, clients must roll back a larger amount of state.

As discussed in section 3.3, once a client detects that 1% of requests are failing, it can stop trusting the primary to provide good first replies and disable its own speculation. If replies are signed, each primary can cause only a single failed speculation, and the resulting view change will dominate recovery time. For reference, over 100 failed speculations in this benchmark result from a 1% failure rate.

7 Related work

This paper contributes the first detailed design for applying client speculative execution to replicated state machine protocols. It also provides the first design and implementation that uses client speculation to hide latency in PBFT [8].

Speculator [28] was originally used to hide latency in distributed file systems, and thus our work shares many of Speculator's original goals. Speculator's distributed file system application assumes the existence of a central file server that always knows ground truth. No such entity exists in a replicated state machine. For instance, non-faulty replicas may disagree about the ordering of read-only requests as discussed in Section 3.2.2. Prior to this paper, Speculator was only used to speculate on zero replies. The possibility of also speculating on a single reply opens up several potential protocol optimizations that we have explored, including the possibility of generating early replies and optimizing agreement protocols for throughput.

Speculative execution is a general computer science concept that has been successfully applied in hardware architecture [15, 17, 35], distributed simulations [19], file I/O [10, 16], configuration management [36], deadlock detection [26], parallelizing security checks [29], transaction processing [20] and surviving software failures [12, 31]. This work contributes by applying speculation to another domain, replicated state machines.

There has also been extensive prior work in the development of replicated state machines, both in the fail-stop [24, 30, 34] and Byzantine [1, 8, 11, 21, 22, 32, 37] failure models. While Byzantine fault tolerance in particular has been an area of active research, it has seen relatively limited deployment due to its perceived complexity and performance limitations.

Our client-side speculation techniques apply equally well to reducing latency in both fail-stop and Byzantine fault tolerance protocols. However, they are particularly useful for protocols that tolerate Byzantine faults due to the higher latencies of such protocols.

PBFT [8] provides a canonical example of a Byzantine fault-tolerant replicated state machine, using multiple phases of replica-to-replica agreement to order each operation. Several systems since PBFT have aimed to reduce the latency in ordering client operations, typically by optimizing for the no-failure case [22] or for workloads with few concurrent writes [1, 11].

Byzantine quorum state machine replication protocols such as Q/U [1] build upon earlier work in Byzantine quorum agreement [3, 4, 13, 27], and provide lower latency in the optimal case. Q/U is able to respond to write requests in a single phase, provided that there are no write operations by other clients that modify the service state; inconsistent state caused by other clients requires a costly repair protocol. HQ [11] aimed to reduce the cost of repair, and reduces the number of replicas required in a Byzantine Quorum system from $5f+1$ to $3f+1$, but it introduces an additional phase to the optimized protocol.

Agreement protocols that use a primary replica are able to batch multiple requests into a single agreement operation, greatly reducing the overhead of the protocol and increasing throughput. While our protocol applies to both quorum and agreement protocols, the higher throughput offered by batched agreement, along with resilience during concurrent write workloads, makes them a better match for our techniques.

Our work on client speculation complements the server-side use of speculation in Zyzzyva [22]. In Zyzzyva, *replicas* execute operations speculatively based on an ordering provided by the primary, while in our system *clients* speculate based on an early response from the primary (or on 0 replies), with replicas executing only committed operations. These two approaches are complementary. Client speculation allows a client to issue a subsequent operation after only a single phase of com-

munication with the primary, which is especially helpful for geographically dispersed deployments where some replicas are far from the client. Server speculation speeds up how fast replicas can supply a consensus response to the client, which would allow clients in our system to commit speculations faster. While we have evaluated client speculation on the PBFT protocol, it would apply equally well to Zyzzyva, where the client can receive early speculative *and* consensus responses, in the absence of failures.

8 Conclusions and future work

Replicated state machines are an important and widely-studied methodology for tolerating a wide range of faults. Unfortunately, while replicas should be distributed geographically for maximum fault tolerance, current replicated state machine protocols tend to magnify the effects of the long network latencies associated with geographic distribution. In this paper, we have shown how to use speculative execution at clients of a replicated service to reduce the impact of network and protocol latency. We outlined a general approach to using client speculation with replicated services, then implemented a detailed case study that applies our approach to a standard fault tolerant protocol (PBFT).

In the future, we hope to apply client speculation to a wider range of protocols and services. For example, adding client speculation to a protocol that uses server speculation [22] should allow clients to commit speculations faster. It may also be possible to apply client speculation to protocols that use more complex replication schemes, such as erasure encoding [18], although clients of such protocols may require more than one reply to predict the final response with high probability.

Acknowledgments

We would like to thank Miguel Castro, our shepherd Steven D. Gribble, and all anonymous reviewers for their valuable comments and suggestions. We would also like to thank Ramakrishna Kotla and Mike Dahlin for providing us with their implementation of Zyzzyva. The work has been supported by the National Science Foundation under awards CNS-0346686, CNS-0428107, CNS-0509093, and CNS-0614985. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 59–74.

[2] Avizienis, A. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985), 1491–1501.

[3] Ben-Or, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 27–30.

[4] Bracha, G., and Toueg, S. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 12–26.

[5] Bracha, G., and Toueg, S. Asynchronous consensus and broadcast protocols. *J. ACM 32*, 4 (1985), 824–840.

[6] Carson, M., and Santay, D. NIST Net – A Linux-based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review 33*, 3 (June 2003), 111–126.

[7] Castro, M. Practical Byzantine Fault Tolerance. Tech. Rep. MIT-LCS-TR-817, MIT, Jan 2001.

[8] Castro, M., and Liskov, B. Practical Byzantine Fault Tolerance. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation* (February 1999), pp. 173–186.

[9] Castro, M., and Liskov, B. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation* (October 2000), pp. 19–33.

[10] Chang, F., and Gibson, G. A. Automatic I/O hint generation through speculative execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation* (February 1999), pp. 1–14.

[11] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation* (November 2006), pp. 177–190.

[12] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., and Hutchinson, N. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)* (2008).

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

[14] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[15] Franklin, M., and Sohi, G. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (May 1996), 552–571.

[16] FRASER, K., AND CHANG, F. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference* (June 2003), pp. 325–338.

[17] HAMMOND, L., WILLEY, M., AND OLUKOTUN, K. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 1998 International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1998), pp. 58–69.

[18] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 73–86.

[19] JEFFERSON, D., BECKMAN, B., WIELAND, F., BLUME, L., DILORETO, M., HONTALAS, P., LAROCHE, P., STURDEVANT, K., TUPMAN, J., WARREN, V., WEDEL, J., YOUNGER, H., AND BELLENOT, S. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the 1987 Symposium on Operating Systems Principles* (November 1987), pp. 77–93.

[20] KEMME, B., PEDONE, F., ALONSO, G., AND SCHIPER, A. E. Processing transactions over optimistic atomic broadcast protocols. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 424.

[21] KIHLSTROM, K. P., MOSER, L. E., AND MELLIAR-SMITH, P. M. The SecureRing protocols for securing group communication. In *Proceedings of the 1998 Hawaii International Conference on System Sciences* (1998), vol. 3, pp. 317–326.

[22] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 45–58.

[23] KOTLA, R., AND DAHLIN, M. High throughput byzantine fault tolerance. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, p. 575.

[24] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[25] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[26] LI, T., ELLIS, C. S., LEBECK, A. R., AND SORIN, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005), pp. 31–44.

[27] MALKHI, D., AND REITER, M. Byzantine Quorum Systems. *Distributed Computing 11*, 4 (1998), 203–213.

[28] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 191–205.

[29] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., AND FLINN, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 2008 International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2008), pp. 308–318.

[30] OKI, B., AND LISKOV, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[31] QIN, F., TUCEK, J., SUNDARESAN, J., AND ZHOU, Y. Rx: Treating bugs as allergies–a safe method to survive software failure. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 235–248.

[32] REITER, M. K. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, vol. 938. Springer-Verlag, Berlin Germany, 1995, pp. 99–110.

[33] RODRIGUES, R., CASTRO, M., AND LISKOV, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 2001 Symposium on Operating Systems Principles* (October 2001), pp. 236–269.

[34] SCHNEIDER, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (December 1990), 299–319.

[35] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., AND MOWRY, T. C. A scalable approach to thread-level speculation. In *Proceedings of the 2000 International Symposium on Computer Architecture* (June 2000), pp. 1–24.

[36] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 237–250.

[37] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 2003 Symposium on Operating Systems Principles* (October 2003), pp. 253–267.