High Throughput Byzantine Fault Tolerance

Ramakrishna Kotla and Mike Dahlin
The University of Texas at Austin

Abstract

This paper argues for a simple change to Byzantine Fault Tolerant state machine replication libraries. Traditional state machine replication based Byzantine fault tolerant (BFT) techniques provide high availability and security but fail to provide high throughput. This limitation stems from the fundamental assumption of generalized state machine replication techniques that all replicas execute requests sequentially in the same total order to ensure consistency across replicas. We propose a high throughput Byzantine fault tolerant architecture that uses application-specific information to identify and concurrently execute independent requests. Our architecture thus provides a general way to exploit application parallelism in order to provide high throughput without compromising correctness. Although this approach is extremely simple, it yields dramatic practical benefits. When sufficient application concurrency and hardware resources exist, CBASE, our system prototype, provides orders of magnitude improvements in throughput over BASE, a traditional BFT architecture. CBASE-FS, a Byzantine fault tolerant file system that uses CBASE, achieves twice the throughput of BASE-FS for the IOZone micro-benchmarks even in a configuration with modest available hardware parallelism.

1 Introduction

With the growing prevalence of large-scale distributed services and access-anywhere Internet services, there is increasing need to build systems that provide high availability to ensure uninterrupted service, high reliability to ensure correctness, high confidentiality against malicious attacks [1] to steal data, and high throughput [23] to keep pace with high system load.

Recent work on Byzantine fault tolerant (BFT) state machine systems has demonstrated that generalized state machine replication can be used to improve availability and reliability [20, 8, 19] as well confidentiality [24]. Furthermore, this work suggests that the approach has important practical properties in that it adds low overhead [8, 20, 24], can recover proactively from faults [9], can make use of existing off-the-shelf implementations to improve availability and to reduce replication cost [20], and can minimize replication of the application-specific parts of the system [24].

However, current BFT state machine systems can fail to provide high throughput. They use generalized state machine replication techniques that require all non-faulty replicas to execute all requests sequentially in the same order, completing execution of each request before beginning execution of the next one. This sequential execution of requests can severely limit the throughput of systems designed to achieve high throughput via concurrency [23]. Unfortunately, this concurrency-dependent approach lies at the core of many (if not most) large-scale network services such as file systems, web servers, mail servers, and databases. Furthermore, technology trends generally make it easier for hardware architectures to scale throughput by increasing the number of hardware resources (e.g., processors, hardware threads, or disks) rather than increasing the speed of individual hardware elements.

Although current BFT systems like PBFT [8] and BASE [20] implement optimizations such as request batching in order to amortize their replication overheads due to agreement protocol messages and cryptographic computations, existing systems still impose this fundamental limitation on application-level concurrency.

In this paper, we argue for a simple addition to the existing BFT state machine replication architectures that allows throughput of the system to scale with application parallelism and available hardware resources. Our architecture separates agreement from execution [24] and inserts a general parallelizer module between them. The parallelizer uses application-supplied rules to identify and issue concurrent requests that can be executed in parallel without compromising the correctness of the replicated service. Hence, the throughput of the replicated system scales with the parallelism exposed by the application and with available hardware resources. More broadly, in our architecture replicas execute requests according to a partial order that allows for concurrency as opposed to the total order enforced by traditional BFT architectures.

We demonstrate the benefits of our architecture by building and evaluating a prototype library for constructing Byzantine fault-tolerant replicated services called CBASE (Concurrent BASE). CBASE extends the BASE system [20] which uses the traditional BFT state machine replication architecture. We use a set of micro-benchmarks to stress test our system and find that when sufficient application concurrency and hardware resources exist, CBASE provides orders of magnitude improvements in throughput over the traditional BFT architecture. We also find that for applications or hardware configurations that can not take advantage of concurrency, CBASE adds little overhead compared to the optimized BASE system. As a case study, we implement CBASE-FS, a replicated Byzantine fault tolerant file system, to quantify the benefits for a real application. CBASE-FS achieves twice the throughput of BASE-FS for the IOZone micro-benchmarks even in a configuration with modest available hardware parallelism. When we artificially simulate more hardware resources, CBASE's maximum write throughput scales by over an order of magnitude compared to the traditional BFT architecture.

The main contribution of this study is a case for changing the standard architecture for Byzantine fault tolerant state machine replication to include a parallelizer module that can expose potentially concurrent requests to enable parallel execution. Based on this study, we conclude that this idea is appealing for two reasons. First, it is simple. It requires only a small change to the existing standard BFT replication architecture. Second, it can provide large practical benefits. In particular, this simple change can improve the throughput of some services by orders of magnitude, making it practical to use BFT state machine replication for modern commercial services that rely on concurrency for high throughput.

The main limitation of this approach is that safely executing multiple requests in parallel fundamentally requires application-specific knowledge of inter-request dependencies. But, we do not believe this limitation undermines our argument. In particular, our prototype parallelizer implements a set of default rules that assume that all requests depend on all other requests. Applications that require or are satisfied with sequential execution can simply leave these default rules in place. But, applications that desire increased throughput can override these rules to expose their concurrency to the replication library. Furthermore, designers of such applications can take an iterative approach, first developing simple rules that expose some application concurrency and later developing more sophisticated rules that expose more concurrency if required for performance.

The rest of this paper proceeds as follows. Sections 2 and 3 outline our system model and review the standard architecture for existing Byzantine fault tolerant state machine replication systems. Then Section 4 describes our proposed architecture and Section 5 describes our prototype replication library, CBASE. Section 6 discusses our experimental evaluation, Section 7 discusses related work, and Section 8 summarizes our conclusions.

2 System Model

Our system model comprises a set of standard assumptions for Byzantine fault tolerant state machine replication. For brevity, we list them here. A more complete discussion of these assumptions is available elsewhere [24]. We assume an asynchronous distributed system where nodes may operate at arbitrarily different speeds and where the network may fail to deliver messages, delay them, corrupt them, duplicate them, or deliver them out of order. The system is safe under this asynchronous model, and it is live under a *bounded fair links* [24] system model that does include a weak synchrony assumption that bounds worst-case delivery time of a message that is sent infinitely often.

We assume a Byzantine fault model where faulty nodes can behave arbitrarily. They can crash, lose data, alter data, and send incorrect protocol messages. We assume a strong adversary who can coordinate faulty nodes in arbitrarily bad ways to disrupt the service. We assume the adversary to be computationally limited and that it cannot subvert cryptographic techniques. We assume that at most f nodes can fail out of n replicas.

3 Background: BFT systems

BFT state machine replication based systems provide high availability and reliability [8, 20] and high security [24] but fail to provide high throughput. There is a large body of research [15, 17, 19, 8, 20, 24] on replication techniques to implement highly-available systems that tolerate failures. Instead of using single server to implement a service, these techniques replicate the server and use a distributed algorithm to coordinate the replicas. The replicated system provides the abstraction of a single service to the clients and continues to provide correct service even when some of the replicas fail.

Figure 1 illustrates a typical BFT state machine replication architecture. Clients issue requests to the replicated service. Conceptually, replicas consist of two stages, an agreement stage and an execution stage. In reality, these two stages may be tightly integrated in a single machine [8, 20] or implemented on different machines [24]. The

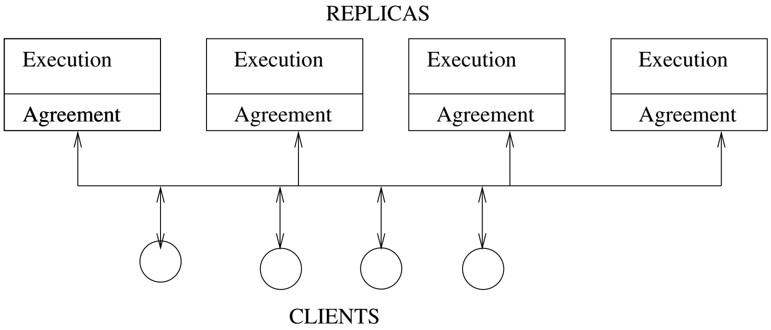


Fig. 1: Traditional BFT Architecture

agreement stage runs a distributed agreement protocol to agree on the order of client requests and the execution stage executes all of the requests in the same order.

Each execution node maintains a state machine that implements the desired service. A state machine consists of a set of state variables that encode the machine's state and a set of commands that transform its state. A state machine takes one or more of the following actions to execute a command:

1. Read a subset of the state variables, called the read-set R.
2. Modify a subset of the state variables, called the write-set W.
3. Produce some output O to the environment.

A command is non-deterministic if its write-set values or output are not uniquely determined by its input and read-set values; otherwise it is a deterministic command. A state machine is called a deterministic state machine if all commands are deterministic. The safety property (correctness condition) of a system that uses state machine replication technique requires that all the non-faulty replicas starting from the same state should reach the same final state after executing the same set of requests from clients. The following requirements [21] ensure safety of a replicated system:

1. **Deterministic state machine:** Every non-faulty replica state machine is deterministic.
2. **Agreement:** Every non-faulty state machine replica receives every request.
3. **Order:** Every non-faulty state machine replica processes the requests it receives in the same relative order.

By ensuring the above conditions, all non-faulty replicas start from the same state, execute all the requests in the same order, end up in the same final state, and produce the same output sequence.

Although the state machine replication technique can provide high availability and reliability, it can fail to provide high throughput. The fundamental drawback of the generalized state machine approach is that the Order requirement does not allow replicas to execute requests concurrently. In particular, unless strong assumptions

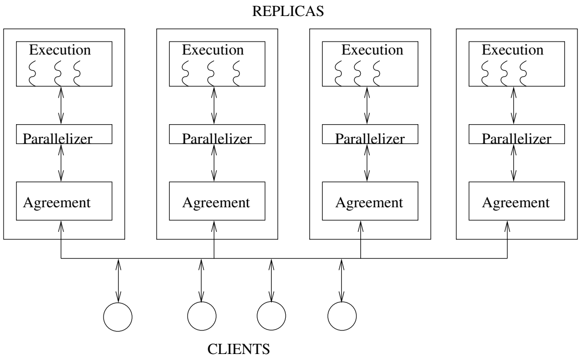


Fig. 2: High throughput BFT state machine replication architecture

are made about state machine's internal implementation, execution node must finish executing request i before executing request $i+1$. Otherwise, concurrency within a state machine could introduce non-determinism into the system, which can cause different replicas' state to diverge and cause safety to be violated. For example, if replicas implementing NFS were to execute multiple write requests to the same file concurrently, different replicas could execute the write requests in different orders and end up having different versions of the same file. Unfortunately, executing requests sequentially can severely limit the throughput of a system. Moreover, the throughput of such systems does not scale with resources as replicas execute one request at a time and hence fail to exploit the additional resources.

4 High Throughput BFT State Machine Replication

Figure 2 illustrates our high throughput state machine replication architecture, where we maintain the separation between the agreement and execution stages and introduce a *parallelizer* between them. The parallelizer takes a totally ordered set of requests from the agreement stage and uses application-supplied rules to first identify independent requests and then issue them concurrently to the execution stage. A thread pool in the execution stage can then execute the requests in parallel to improve system throughput.

4.1 Relaxed Order and Parallelizer

The key idea of high throughput state machine replication is to relax Schneider's *Order* requirement on state machine replication (defined above) to allow concurrent execution of independent requests without compromising safety.

We say that two requests are *dependent* if the write-set of one has at least one state variable in common with the read-set or write-set of the other. More formally, we define dependence as follows: Request r_i, with read-set R_i and write-set W_i and request r_j, with read-set is R_j and write-set W_j, are *dependent requests* if any of the following conditions is true (1) $\mathrm{W}_i \cap \mathrm{W}_j \neq \phi$, (2) $\mathrm{W}_i \cap \mathrm{R}_j \neq \phi$, or (3) $\mathrm{R}_i \cap \mathrm{W}_j \neq \phi$. We also define dependence to be transitive: if r_i and r_j are dependent and r_j and r_k are dependent, then r_i and r_k are dependent. Two requests r_i and r_j are said to be *concurrent* if they are not dependent.

Given this notion of dependence, we refine Schneider's *Order* requirement for replicated state machine safety into a *Relaxed Order* requirement:

3a **Relaxed Order:** Every non-faulty state machine replica processes any pair of dependent requests it receives sequentially and in the same relative order.

Notice that under the Relaxed Order requirement, concurrent requests can be processed in parallel. Thus, with the Relaxed Order requirement, all non-faulty replicas execute requests in the same *partial order* as opposed to the traditional architecture where all correct replicas execute requests in the same *total order*.

In the new architecture, the parallelizer uses application-specific information to take advantage of the Relaxed Order requirement. The parallelizer transforms a totally ordered schedule of requests provided by the agreement protocol into a partially ordered schedule based on application semantics.

A *sound parallelizer* ensures the following *partial order property*: for any two requests r_i and r_j such that r_i and r_j are dependent and r_i precedes r_j in the total order established by the agreement stage, then r_i completes execution before r_j begins execution. For fault tolerance, we also assume that the parallelizer has a *local decision* property: each replica's parallelizer does dependence analysis locally and does not exchange messages with other replicas. Hence, given a correct agreement protocol, faulty replicas cannot affect the partial order enforced at the correct replicas.

Notice that there are two properties that are *not* required of a parallelizer. First, we do not require *precision*: a sound parallelizer may enforce additional ordering constraints on requests beyond those required by the partial order property. This non-requirement is important because it allows us to simplify the design of parallelizers for complex applications by building *conservative* parallelizers that can introduce false dependencies between requests. For example, in Section 5.3 we describe a simple NFS implementation that uses a conservative analysis to identify some, but not all, concurrent requests. Second, we do not require *equality*: different correct parallelizers may enforce different partial orders as long as all correct parallelizers' partial orders are consistent with the order required by the partial order property. One could, for example, implement multiple versions of the parallelizer for an application to prevent any one implementation from being a single point of failure [22].

The properties of existing BFT state machine replication systems and of a sound parallelizer with local decisions ensure the correctness of our architecture.

Safety: The final state reached by our architecture for a given set of requests is equivalent to the final state reached by executing those requests in a linearizable [11] order. More precisely, for a given initial application state (or abstract state [20]) S_0, if a system comprising a correct agreement stage [8, 9, 20, 24], sound parallelizer, and correct execution stage [24] reaches final state S_{final} after the agreement stage receives a set of inputs I and establishes a total order I across these inputs, and after all requests in I have completed execution, then this final state S_{final} is equivalent to the final state reached by starting the state machine in the equivalent initial state and serially executing requests I in order I.

Proof sketch: The agreement protocol (e.g., atomic multicast) [8] used by the system guarantees total ordering of all requests. From the properties of the parallelizer, all non-faulty replicas execute the dependent requests in the same order and satisfy the Relaxed Order property. Relaxed Order ensures safety because (1) *Dependent requests execute in the same order at all replicas and follow the order provided by the agreement stage*. This constraint is by definition enforced by a sound parallelizer. And (2) *Independent requests can be commuted safely*. Because independent requests modify disjoint sets of state variables, the result of executing independent requests in any order places the system in the same final state.

Liveness If a system comprised of the agreement and execution stages is live, then the system comprised of the agreement, execution, and parallelizer is also live if the parallelizer is sound and also implements the following fairness property: eventually the *oldest pending request* that has been issued by the agreement protocol to the parallelizer but that has not yet completed execution must be the next request given to the execution thread pool. This property can be discharged by a sound parallelizer because the oldest pending request need not have any predecessors under the partial order property. And this property is sufficient when combined with our assumption that the agreement and execution stages are live.

4.2 Advantages and Limitations

This state machine replication architecture has two potential advantages. First, it can support high-throughput applications. If the workload contains independent requests and the system has enough hardware resources, then independent requests can be executed concurrently by the execution stage to improve the throughput of a system. Second, it is simple and flexible. In particular, to achieve high throughput, we did not change any of the other components in the system like client behavior, the agreement protocol, or the application. These components can therefore be changed to suit the requirements of the replicated system. For example, one can change the agreement protocol and client side behavior to build a system that either tolerates Byzantine failures or fail-stop failures while achieving high throughput without modifying the parallelizer.

The main limitation of a system using this architecture is that the rules used by the parallelizer to identify dependent requests require knowledge of the inner workings of each application. In many ways, this knowledge is similar to that required to build the abstraction layer used in BASE to mask differences in different implementations of the same underlying application [20]. However, it may in general be difficult to know what internal state a given request affects or to determine with certainty whether any given pair of requests are dependent.

Fortunately, it is not necessary to completely understand the inner workings of an application in order to define a parallelizer for it. In particular, it is always permissible to define *conservative* rules that include all true dependencies but also include some false dependencies. System designers may choose to follow an incremental approach by first defining a set of simple but conservative rules to identify "obvious" concurrent requests and then progressively refine the rules if more parallelism is needed to meet performance goals.

5 CBASE Prototype

The goal of our prototype is to demonstrate a general way to extend state machine replication systems in order to allow concurrent execution of requests for applications that can identify dependencies among requests.

Our prototype, CBASE (Concurrent BASE) system extends the BASE [20] system to use the high throughput state machine replication architecture described in the previous section. BASE (BFT with Abstract Specification Encapsulation) uses the PBFT (Practical Byzantine Fault Tolerance) algorithm [8] to provide Byzantine fault tolerant services. The PBFT agreement protocol is well suited for practical systems as it does not assume synchrony for safety and also in that it introduces modest delay while providing high reliability and availability. BASE extends PBFT by using abstraction to improve robustness and reduce cost.

CBASE modifies BASE to cleanly separate the agreement and execution stages[1] and introduces a parallelizer between these stages as shown in Figure 2. CBASE's single threaded agreement module uses BASE's 3-phase atomic multicast protocol to establish a total order on requests. The parallelizer thus receives a series of requests from the agreement module, and it uses an application-specific set of rules to identify dependencies among requests and thereby establish a partial order across them. A pool of worker threads each draws a request out of the parallelizer, executes it on the application state machine, and informs the parallelizer of request completion.

Internally, the parallelizer uses a dependency graph to maintain a partial order across all pending requests; vertices represent requests and directed edges represent dependencies. The dependency graph forms a DAG as there can be not be circular dependencies because dependent requests are ordered in the order they are inserted and the independent requests are not ordered. The parallelizer has an application-independent scheduler that uses the DAG to schedule the requests according to the partial order. The worker threads in the execution stage receive independent requests (vertices with no incoming edges) from the parallelizer, execute them concurrently, and remove a request from the DAG when its execution completes.

The default behavior of the parallelizer is to treat all the requests as dependent, in which case it behaves like the existing BASE system where the requests are executed sequentially. This default behavior can be used when the

[1] Note, however, that our implementation does not allow the agreement and execution modules to run on different sets of machines [24].

finite state machine is treated as a black box or where dependencies across requests cannot easily be inferred. The rules in the parallelizer can be incrementally refined by taking a conservative approach where the requests known to touch different states can be treated as independent and all the other requests can be treated as dependent. Similarly, for backwards compatibility with existing state machines, if a state machine is not thread safe we can just have single worker thread or implement a mutual exclusion lock around the state machine.

5.1 Parallelizer interface

The parallelizer appears to the agreement and execution threads as a variation of a producer/consumer queue. When a consumer thread asks for a request, the parallelizer searches for a request that is independent of all incomplete preceding requests and returns one if found; otherwise it blocks the consumer thread until a request becomes independent. The interface used by the agreement and execution stages is as follows:

- Parallelizer.insert(): Called by the agreement stage to enqueue a request when the request is committed in the agreement stage. It inserts a node in the dependency graph, does dependence analysis of this request with preceding outstanding requests, and creates incoming edges to this node from the requests that this request is dependent on. The new request is put in a blocked state if it has incoming edges.

- Parallelizer.next_request(): Called by the execution stage to return a ready request. A request is ready if it does not have any incoming edges in the dependency graph. If no request is ready, this call blocks until it can return a ready request.

- Parallelizer.remove_request(): Called by the execution stage after the execution of a request is completed and a reply is sent to the client. The parallelizer removes the node corresponding to this request and all the outgoing edges from this node. As a result, some of the blocked requests may transition to the ready state.

- Parallelizer.sync(): This interface supports replica checkpoints required by the BASE system [20]. This function is called by the agreement stage with the sequence number of the next checkpoint. The scheduler in the parallelizer blocks all requests that are enqueued beyond this sequence number even if they are not dependent on other requests until the agreement stage completes a checkpoint. The agreement stage updates the next checkpoint sequence number by calling this function as soon as the checkpoint is complete.

5.2 Dependence Analysis

The parallelizer's goal is to determine if a new request is dependent on any pending request using application-specific rules. The parallelizer design must balance three conflicting goals: (1) Generality – the parallelizer should provide an interface that allows a broad range of applications to encode rules for detecting dependencies among their requests; (2) Simplicity – the interface for specifying these rules should be simple to reduce the effort and likelihood of error in dependency-rule specification; and (3) Flexibility – the interface should allow specification of simple conservative dependency rules and progressive refinement to more precise dependency rules that expose more concurrency. Notice that our design is a compromise among these design goals and that other algorithms for identifying dependencies among requests could be explored in future work.

In the CBASE prototype, conflict detection between a pair of requests depends on the *function* each invokes and on the *arguments* each request passes to its function. An application that has F distinct function entry points provides the parallelizer with four things:

1. A *request parser* that takes an application request and produces a function ID and an argument object.

2. An *operator concurrency matrix OCM* that identifies pairs of functions that are considered to be in conflict *independent* of the arguments to the functions. *OCM* is an FxF matrix, where $OCM[i,j]$ is true if a request invoking function i and a request invoking function j are always considered to be dependent. This dependency may be because these functions always access common state with one of them updating that state, or this dependency may be because these functions sometimes access common state and a conservative design assumes they always do for simplicity or because more careful analysis of arguments is impractical for the application.

3. An *argument analysis function AAF* that takes two argument objects and returns true if an analysis of the arguments indicates that functions that are not flagged by the *OCM* may access common state when supplied with these arguments. More precisely, $AAF(a_1,a_2)$ must return true if there exists any pair of functions f_1, f_2 such that $OCM[f_1,f_2] = \mathit{false}$ but $f_1(a_1)$ and $f_2(a_2)$ access common state and either modifies that common state.

4. An *operator+argument concurrency matrix OACM* that identifies pairs of functions that are considered to be in conflict only when an analysis of the arguments indicates that they may access common state.

When a new request r_j calling function f_j with arguments a_j arrives, the parallelizer compares it to each pending request r_i calling function f_i with arguments a_i as follows. First, it checks for argument-independent dependencies: if $OCM[f_i,f_j]$ is true, the requests are dependent. If not, then it checks to see if the arguments indicate that there may be additional risk of dependencies: if $AAF(a_i,a_j)$ is true, then it also checks for argument-dependent dependencies and identifies a dependency between r_i and r_j if $OACM[f_i,f_j]$ is true. Finally, if $OCM[f_i,f_j]$ is false and either $AAF(a_i,a_j)$ is false or $OACM[f_i,f_j]$ is false, then no dependency between r_i and r_j exists.

This structure facilitates a 2-level analysis in which the operator concurrency matrix *OCM* defines broad rules where no argument analysis is attempted or needed and in which the operator+argument concurrency matrix *OACM* defines more precise rules that are invoked after an analysis of the arguments indicates that two calls that sometimes are independent may be in conflict due to their arguments. The next subsection describes our NFS file system prototype where we use the *OACM* to encode rules for functions if the state they affect is easily identified from file handles in their arguments and where we use the *OCM* to handle other functions.

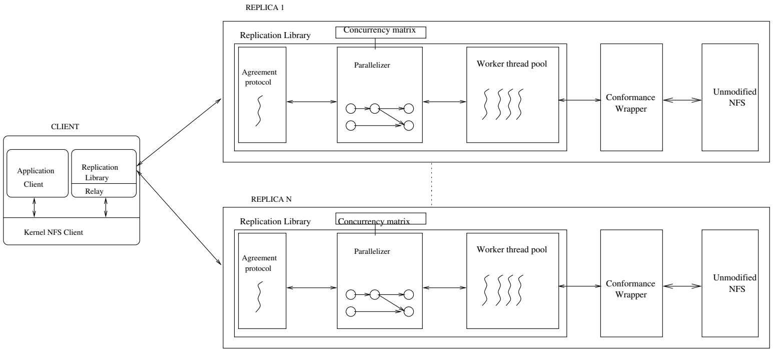


Fig. 3: CBASE-FS: High throughput Byzantine fault tolerant NFS

5.3 Example Service: NFS

We have implemented CBASE-FS, a Byzantine fault tolerant NFS [4] using CBASE as shown in Figure 3. Our implementation builds on BASE-FS [20], which uses existing implementations of NFS to implement each instance of the replicated state machine. In particular, a client in CBASE-FS mounts the replicated file system exported by the replicas as a local NFS file system [18]. Unmodified applications access the file system using standard file system calls. The local kernel sends NFS calls to the local user-level NFS server, which acts as a wrapper for CBASE-FS by calling the *invoke* procedure of the BASE replication library to relay the request to the replicas. This procedure returns when the wrapper receives $f+1$ matching replies from different replicas.

The agreement stage in CBASE takes part in the atomic multicast protocol to establish a total order on requests and then sends each ordered request to the parallelizer. The parallelizer updates the dependency graph using NFS's concurrency matrix as defined in section 5.3.1 whenever a request is enqueued. The worker threads in the execution stage dequeue independent requests and execute the requests using the conformance wrapper interface as defined by the BASE system for NFS.

CBASE-FS uses BASE's [20] abstraction layer (conformance wrapper) to resolve non-determinism in NFS such as file handle assignment or timestamp generation. Additionally, CBASE introduces a new source of non-determinism due to concurrent execution of NFS *create* operations to different files. The existing BASE conformance wrapper at different replicas could return different file handles based on the order of execution of these requests. We fix this problem by having a rule in the concurrency matrix to treat the requests with create/delete operations as always dependent.[2]

Rodrigues et al. [20] provide additional details of the BASE library and BASE-FS Byzantine fault tolerant NFS server on which we build.

[2] We speculate that additional concurrency could be exposed by including constraints based on a request's total-order sequence number to the conformance wrapper's file handle generation logic and the parallelizer's dependency logic.

5.3.1 Concurrency Matrix for NFS

For NFS, we keep the classification simple by just looking at the file handles, and thus have conservative rules for some of the operations. Our argument analysis function (AAF) defines two arguments as related if they include a common file handle. We present the key rules that are used in defining NFS's argument-independent operator concurrency matrix (OCM) and argument-dependent operator+argument concurrency matrix (OACM) below. Refer to [14] for the complete definitions of the concurrency matrices.

- getattr and null requests are read only requests and hence are independent for both related and unrelated arguments.
- Reads to different files are independent whereas reads to the same files are dependent. Reads modify the last-accessed-time attribute of a file, so we do not concurrently execute read requests to the same file.
- Writes to different files are independent and writes to the same file are dependent.
- All create and remove operations to the same file or different files are dependent as they introduce non-determinism if executed concurrently as discussed above.
- Create/Rename/Remove operations are always treated as dependent on Read/Write operations. Read/Write operations carry the file handle of the file whereas create/rename/remove requests carry the file handle of the directory in which file is present and the filename of the file to be deleted. As we just look at the file handle to decide if two arguments are related or not, we cannot execute the requests with create/rename/remove concurrently with read/write requests.

We give up some potential concurrency across requests with these conservative rules. Looking at other fields in the request apart from file handle and keeping additional state about file handles could allow for more sophisticated and accurate classification. There is a tradeoff between on one hand the simplicity of the design and the time spent to classify requests versus on the other hand the amount of concurrency realized by the parallelizer. This trade-off should be explored in more detail in the future.

5.4 Additional Optimizations

The BASE and the PBFT systems have implemented optimizations [8, 9, 20] to improve throughput of the system. CBASE also implements the following throughput optimizations from BASE:

- **Reduced Communication:** This optimization avoids having all the replicas send replies to the client. Instead, the client designates a replica to send the result and all the other replicas just send the digest of the result. The digests allow the client to check the correctness of the result while significantly reducing network bandwidth consumption for large replies. If the client does not receive a correct result from the designated replica, it retransmits the request as usual, requesting that all replicas send the full reply.

- **Request batching:** The primary reduces agreement protocol overhead by assigning a single sequence number to an ordered batch of requests and by starting single instance of the agreement protocol for the batch. When the system is heavily loaded, this batching reduces protocol overhead at little cost to latency.
- **Read-only optimization:** A client multicasts a read-only request directly to all replicas, and the replicas execute it immediately after checking that it is properly authenticated, that the client has access, and that the request is in fact read-only. The last check is done by service specific upcall that is similar in spirit to our more general dependency check in the parallelizer.

However, CBASE does not execute requests tentatively as in BASE and PBFT systems because it is shown in [7] that this optimization has little impact on throughput when used along with request batching and that it adds complexity to the code to keep uncommitted state in the system.

6 Evaluation

A high throughput BFT system should achieve two goals: (1) it should provide high throughput by exploiting application parallelism and additional resources, (2) it must have low overhead when there is no parallelism in the application or when there are no additional resources to execute requests concurrently. CBASE should outperform BASE when there is scope for concurrent execution of requests and should perform as well as BASE when there is no scope for concurrent execution. This section evaluates CBASE and compares its performance with BASE. We also evaluate the performance of replicated the NFS system (CBASE-FS) that uses CBASE and compare its performance with BASE-FS and NFS that is not replicated.

All experiments run with 4 replicas and the system tolerates one Byzantine fault. Replicas run on single processor machines with 933 MHZ PIII processor and connected by a 100 Mbit ethernet hub. All the machines have 256MB of memory except for one that has 512MB of memory. The experiments run on an isolated network. We use 5 client machines to load the system and the client processes were evenly distributed across the machines. Client machines are connected to the network through the same ethernet hub as the replicas. Two of the client machines have 933 MHZ PIII processor with 512MB of memory and the other three machines have 450 MHZ PIII processor with 128KB of memory. All machines run Redhat Linux 7.2.

6.1 Micro-Benchmark

The micro-benchmark compares the performance of BASE and CBASE executing a simple, stateless service - clients sends 0KB requests to which the servers repies 0KB results. We show that for our microbenchmark CBASE imposes little additional latency or overhead compared to BASE and that CBASE's throughput scales linearly with application parallelism and available hardware resources.

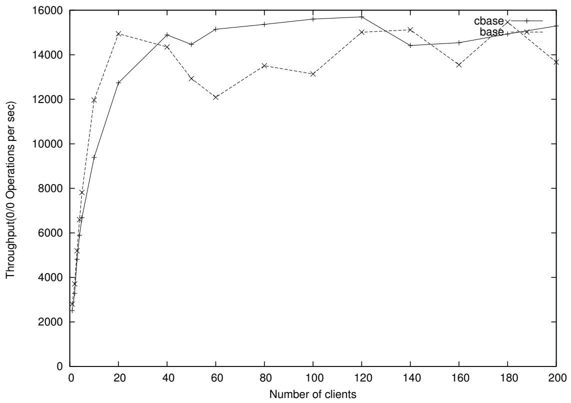
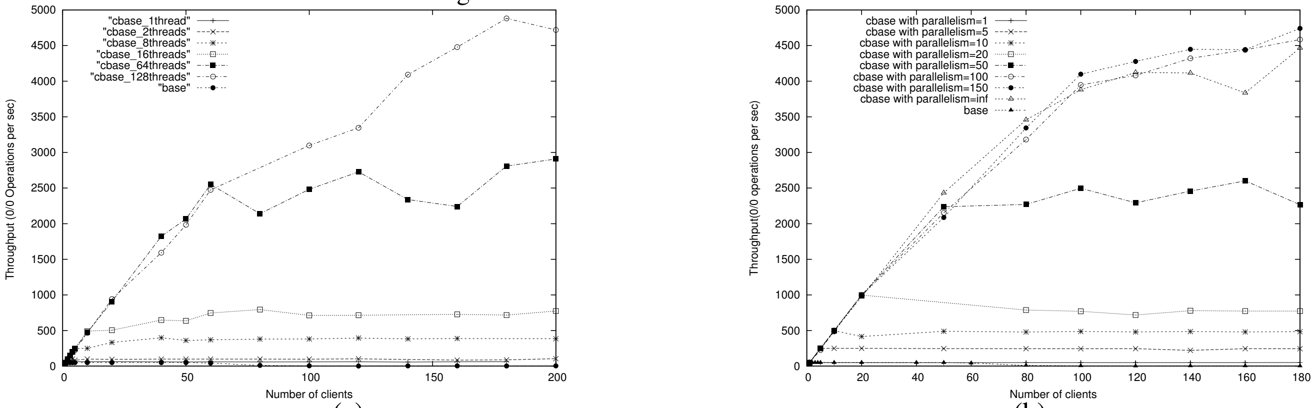


Fig. 4: Overhead of CBASE versus BASE



(a) (b)

Fig. 5: Overhead of CBASE versus BASE

6.1.1 Overhead

Figure 4 compares the overhead of BASE and CBASE by running the baseline benchmark configured with infinite application concurrency (no shared state across requests) and minimal hardware demand per request (each application request at the server simply returns immediately). BASE is CPU-limited—a small number of clients saturate the CPU, but BASE allows throughput to reach a peak of about 15,000 requests per second by employing agreement-stage batching [9], yielding a CPU overhead of less than 100 μs per request. CBASE runs with 16 execution threads and BASE runs with 1 thread. All points in the graph are averages of 3 runs with variance of less than 15%. The CBASE parallelizer treats all requests as independent, but limited hardware resources limit the benefits gained by concurrency —requests run on a uniprocessor and return immediately. Figure 4 shows that the lines representing CBASE and BASE closely follow each other illustrating that CBASE introduces little overhead when there is no scope for concurrent execution of requests.

6.1.2 Scalability of throughput with application parallelism and resources

The throughput of a service depends both on the parallelism present in the application and on the hardware resources (e.g., processors, disks, bandwidth) available to the system. In this set of experiments, we evaluate the scalability of throughput with varying application parallelism and hardware resources.

First, we evaluate the ability throughput to scale with resources. We simulate accesses to a varying array of parallel disks by running the benchmark with the modification that the code to process each request sleeps for 20ms before returning a reply. The CBASE parallelizer assumes infinite parallelism in the application and considers all requests to be independent. We simulate varying "disk" resources by configuring CBASE to run with varying numbers of execution threads. We note that BASE still runs with a single thread since it never attempts to issue more than one request to the execution stage at a time. Figure 5(a) shows that the throughput of BASE saturates at 50 ops/sec (as expected with 20ms service time for each operation) which matches the throughput of CBASE running with 1 thread. The throughput of CBASE increases with the number of clients but eventually saturates because increasing the number of clients improves concurrency only if throughput is limited by the available hardware resources. As the number of "disks" (threads) increases, the throughput of CBASE increases nearly linearly—128 "disks" reach a throughput of 4700 requests/second.

Next, we evaluate the scalability of throughput with parallelism in the application. We run the same experiment as above except that we fix the number of resources in this experiment and vary parallelism in the application. We emulate 100 resources by fixing the number of CBASE execution threads to 100. We define the *parallelism factor* as the number of requests that we allow to be executed concurrently, and simulate varying application parallelism by varying this parameter. Thus, the parallelizer assigns each incoming requests to one of *parallelism factor* buckets and creates dependencies among all requests to the same bucket, allowing only a fixed number of requests to be independent at any point of time. Figure 5(b) shows that the throughput of BASE saturates at 50 ops/sec and that CBASE matches this performance when the application *parallelism factor* is 1. CBASE's maximum throughput increases almost linearly with increasing *parallelism factor* up to 100. The throughput of CBASE does not improve beyond a *parallelism factor* of 100 because it is limited by the 100 simulated hardware resources.

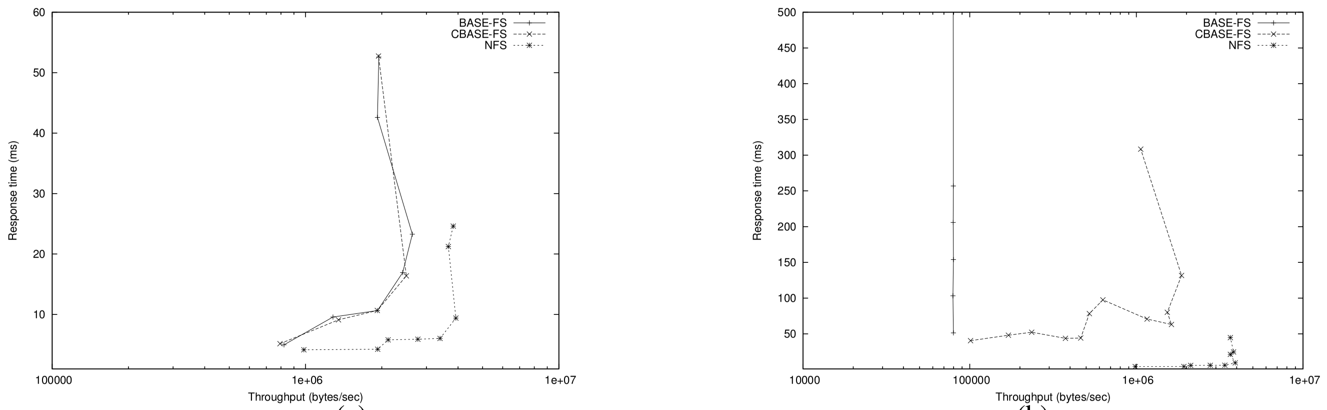
Notice that in both of these experiments, when application parallelism and hardware resources are available, CBASE's throughput can exceed BASE's by orders of magnitude.

6.2 NFS Micro-Benchmarks

In this subsection, we evaluate the performance of CBASE-FS, a replicated NFS that uses CBASE. We also compare the performance of CBASE-FS with BASE-FS and unreplicated NFS.

6.2.1 Local disk

In this benchmark, each client writes 4KB of data to a different file in a directory exported by the file system. We vary the number of concurrent clients and measure the response times throughput of the system. As described in Section 4, requests to different files are treated as independent requests by the CBASE parallelizer. CBASE-FS runs with 16 threads and unreplicated NFS runs with 16 daemon processes. In all file system instances, NFS servers write asynchronously to the disk.



(a) (b)

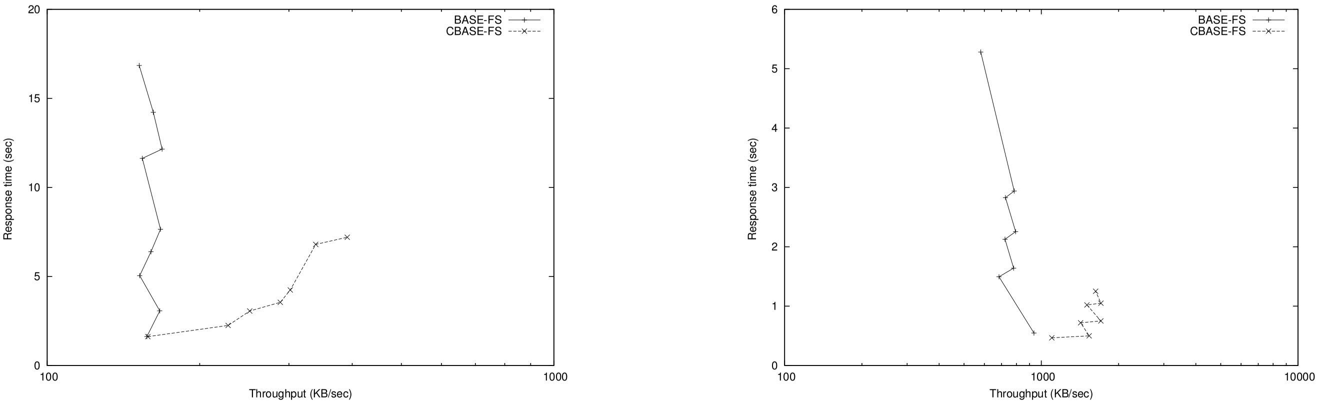
Fig. 6: Throughput versus response time with 4KB writes

Overhead Figure 6(a) plots the response time versus the throughput of CBASE-FS, BASE-FS, and unreplicated NFS. CBASE-FS and BASE-FS closely follow each other and their throughput saturates around 2.5MB/sec, whereas the throughput of NFS saturates around 4MB/sec. In this experiment, because servers run on a uniprocessor system and write asynchronously to the local disk there is little or no scope for the concurrent execution of requests because the threads rarely block as they write in the file buffer cache in memory. Hence we show that when there is no scope for concurrency CBASE-FS performs as well as BASE-FS and adds little or no overhead. The maximum throughput of BASE and CBASE is within a factor of 2 compared to unreplicated NFS; the difference stems from the extra overhead of processing protocol messages, additional cryptographic computations, and extra kernel crossings. For similar reasons, NFS also yields less latency than BASE and CBASE.

Benefits of Pipelining In this experiment we evaluate the performance when there is scope for concurrent execution of requests. We simulate this scenario by making BASE and CBASE servers sleeping for 20 ms after writing to a file and before sending a reply to the client. Figure 6(b) shows the response time plotted against the throughput of BASE, CBASE and NFS. The throughput of BASE saturates at about 90 KB/sec since it cannot execute more than 1 request at a time. However, CBASE achieves its maximum throughput of about 2MB/sec when there is sufficient load on the system to run enough concurrent requests to achieve this throughput, which is almost 20 times more than that of the throughput of BASE. We did not modify the NFS implementation to sleep for 20 ms so its performance remains the same. This experiment shows that CBASE-FS does orders of magnitude better than BASE-FS when there is scope for concurrent execution of requests.

6.2.2 Iozone micro-benchmark

Iozone [2] provides various microbenchmarks to test the performance of commercial file systems. We run the *write* and *random mix* micro-benchmarks to test CBASE-FS and compare its performance with BASE-FS. Rather than introduce artificial delays as above, we introduce the opportunity for hardware parallelism by configuring our system so that each file server accesses data on a *remote disk* that it mounts via NFS from a separate machine. Each IO request may thus access the local CPU, network, remote CPU, and remote disk, which affords the system an opportunity to benefit from pipelining.[3] We use the remote disk setup to evaluate the performance in these experiments . We run the Iozone micro-benchmarks in cluster-mode, where clients are equally divided among 5 client machines and each client accesses a different file.



(a) (b)

Fig. 7: Throughput versus response time for IOZONE micro-benchmarks

The write microbenchmark measures the performance of writing 256KB of data to a new file. We configure the test to have each client write to a different file to provide parallelism across the requests to the file systems. We vary the number of clients to vary the load on the system. As shown in Figure 7(a), the throughput of BASE saturates at about 160 KB/sec where as CBASE saturates at about 320 KB/sec resulting in 100% improvement in performance. CBASE-FS could not achieve more than a 2x improvement in performance despite having more available application-level parallelism because the system is limited by the remote disk bandwidth. Unreplicated NFS achieves a maximum throughput of 500KB/sec when the NFS server is running on the remote disk machine. All points in the graph are the average of three runs with a maximum variance of about 20%.

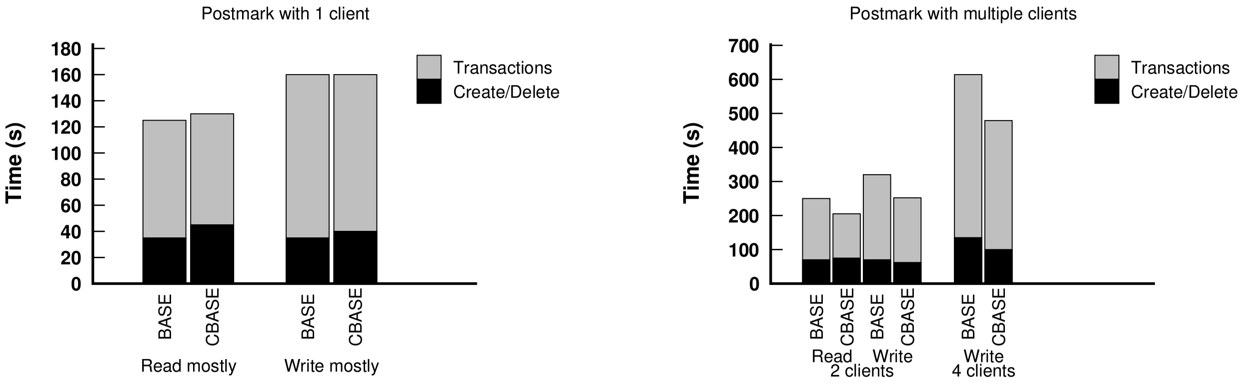
The random mix microbenchmark measures the performance of writing and reading files of size 256KB with accesses being made to random locations within each file. We configured the test to have clients write to different files to provide parallelism across requests, and we vary the number of clients to vary the load on the system. BASE's throughput saturates at about 1MB/sec and CBASE's at about 2MB/sec. File caching at clients improves the throughput of both systems compared to the previous experiment. All the points in the graph are averages of three runs with a maximum variance of 16%. Overall, CBASE-FS's maximum throughput is 100% better than that of BASE-FS.

[3] Our lab currently does not have a sufficient number of multi-disk machines to examine parallelism in that case. We hope to measure our system in such an environment for the final version of this paper.

6.3 Macro-benchmarks

We evaluate the performance of CBASE-FS and BASE-FS with two file system macro-benchmarks: Andrew [12] and Postmark [3].

For the Andrew-100 benchmark—which sequentially runs 100 copies of the Andrew benchmark, provides little concurrency, and is largely client-CPU-limited—CBASE-FS and BASE-FS have essentially identical performance with BASE outperforming CBASE by 4%. We omit this graph for brevity.



(a) (b)

Fig. 8: Postmark benchmark

PostMark [3] is a benchmark to measure performance of the Internet software class of applications such as email, net news, e-commerce, etc. It initially creates a pool of files and then performs a specified number of transactions consisting of creating or deleting a file and reading or appending a file. We set file sizes to be between 1KB and 100KB. We run the benchmark with 100 files for 500 transactions. In our *read-mostly* experiment, we set the read bias at 9 so that transactions are dominated by reads over appends. In our *write-mostly* experiment, we set the read bias at 1 so that transactions are dominated by writes compared to reads. CBASE-FS and BASE-FS replicas write to the remote disk to evaluate the benefits of concurrent execution when run with multiple postmark clients.

Figure 8(a) shows the performance of BASE-FS and CBASE-FS when the experiment is run with one postmark client. The performance of CBASE-FS and BASE-FS are nearly identical. Figure 8(b) shows the performance of BASE-FS and CBASE-FS when the experiment is run with 2 and 4 postmark clients and where each client operates on a different set of files. CBASE-FS is 20-25% faster than BASE-FS when run with multiple clients. CBASE-FS could not realise as much improvement in performance as in microbenchmarks because it is limited by the single available hardware disk.

7 Related Work

There is a large body of research on replication techniques to implement highly-available systems that tolerate failures. To the best of our knowledge, this is the first study that tries to improve throughput of a Byzantine fault tolerant system by using application semantics to execute requests concurrently, without compromising on the safety of the system.

Systems that tolerate *benign faults* [15, 17, 5, 13] assume that components fail by stopping or omitting some steps. Unfortunately, such assumptions are less tenable in large-scale distributed system because malicious attacks, software errors, and operator mistakes can cause faulty nodes to behave arbitrarily.

Byzantine fault tolerant systems provide highly-available and reliable services by tolerating such arbitrary *Byzantine* failures [16]. Byzantine fault tolerant state machine replication has been extensively studied in theoretical settings [6, 19, 10]. Recent work by Castro and Liskov [8, 9] has shown that BFT systems can be implemented in real systems with low overhead and latency. Rodrigues et al. have shown how to abstract away implementation differences across different replicas to allow BFT systems to be implemented with different off-the-shelf implementations of services like NFS and databases [20]. Yin et al. [24] cleanly separate agreement from execution in BFT state machine replication to reduce replication costs and improve the confidentiality properties of the architecture. We build on all of this work by exploiting the basic protocol [8], the read-only optimization [8], the use of MACs rather than signatures for authentication [8], batching for throughput [9], checkpoints and recovery [9], abstraction of application nondeterminism such as NFS's file handle generation [20], and separation of agreement from execution [24]. However, whereas existing approaches provide useful optimizations for improving system throughput such as request batching, read-only optimisations, and tentative execution, the throughput of all of these systems is fundamentally limited by their strategy of serializing all requests to the replicated application. We provide a general strategy for exploiting application-level and hardware-level parallelism that can be applied to any of these systems.

Schneider [21] introduces the idea of using application semantics to reorder commutative requests in the state machine replication technique. Reordering requests can improve average response time of system but will not improve throughput of a system. We generalize this idea to use application semantics to identify independent requests and concurrently execute these requests to improve throughput of a system.

8 Conclusion

This paper proposes a simple change to existing BFT state machine replication architectures to improve the throughput of a replicated system by separating agreement from execution and by introducing an application-specific parallelizer between these two stages. We have build a system prototype called CBASE uses this technique and demonstrate that it provides orders of magnitude improvement in performance over existing systems provided there is enough parallelism in the application and there are sufficient hardware resources.

References

[1] http://www.cert.org.

[2] http://www.iozone.org.

[3] http://www.netapp.com/tech_library/postmark.html.

[4] Nfs : Network file system protocol specification. Request for Comments 1094, Network Working Group, ISI, Mar. 1987.

[5] T. Bressoud and F. Schneider. Hypervisor-based Fault Tolerance. In *15th ACMSymposium on Operating Systems Principles*, pages 1–11, Dec. 1995.

[6] R. Canetti and T. Rabin. Optimal Asynchronous Byzantine Agreement. Technical Report 92-15, Dept. of Computer Science, Hebrew University, 1992.

[7] M. Castro. *Practical Byzantine Fault Tolerance*. PhD thesis, MIT, Jan. 2001.

[8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *3rd Symp. on Operating Systems Design and Impl.*, Feb. 1999.

[9] M. Castro and B. Liskov. Proactive recovery in a Byzantine-Fault-Tolerant system. In *4th Symp. on Operating Systems Design and Impl.*, pages 273–288, 2000.

[10] J. Garay and Y. Moses. Fully Polynomial Byzantine Agreement for $n>3t$ Processors in $t+1$ Rounds. *SIAM Journal of Compouting*, 27(1), 1998.

[11] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[12] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Trans. on Computer Systems*, 6(1):51–81, Feb. 1988.

[13] H. V. J. Napper, L. Alvisi. A fault-tolerant java virtual machine. In *Proc. of the Twenty Third International Conference on Distributed Computing Systems*, Jun 2003.

[14] R. Kotla. High throughput byzantine fault tolerant architecture. Master's thesis, UT, Austin, Dec. 2003.

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

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

[17] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp File System. In *13th ACM Symposium on Operating Systems Principles*, Oct. 1991.

[18] D. Mazires. A toolkit for user-level file systems. In *USENIX Annual Technical Conference*, pages 261–274, June 2001.

[19] M. Reiter. The Rampart toolkit for building high-integrity services. In *Dagstuhl Seminar on Dist. Sys.*, pages 99–110, 1994.

[20] R. Rodrigues, M. Castro, and B. Liskov. Base: Using abstraction to improve fault tolerance. In *18th ACM Symposium on Operating Systems Principles*, Oct. 2001.

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

[22] U. Voges and L. Gmeiner. Software diversity in reacter protection systems: An experiment. In *IFAC Workshop SAFECOMP79*, May 1979.

[23] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *18th ACM Symposium on Operating Systems Principles*, pages 230–243, 2001.

[24] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *19th ACM Symposium on Operating Systems Principles*, Oct 2003.