On the Practicality of Practical Byzantine Fault Tolerance

Nikos Chondros, Konstantinos Kokordelis, Mema Roussopoulos

On the Practicality of Practical Byzantine Fault Tolerance

Nikos Chondros, Konstantinos Kokordelis, and Mema Roussopoulos

University of Athens

Abstract. Byzantine Fault Tolerant (BFT) systems are considered to be state of the art with regards to providing reliability in distributed systems. Despite over a decade of research, however, BFT systems are rarely used in practice. In this paper, we describe our experience, from an application developer's perspective, trying to leverage the publicly available, highly- studied and extended "PBFT" middleware (by Castro and Liskov), to provide provable reliability guarantees for an electronic voting application with high security and robustness needs.
We describe several obstacles we encountered and drawbacks we identified in the PBFT approach. These include some that we tackled, such as lack of support for dynamic client management and leaving state management completely up to the application. Others still remaining include the lack of robust handling of non-determinism, lack of support for web-based applications, lack of support for stronger cryptographic primitives, and more. We find that, while many of the obstacles could be overcome, they require significant engineering effort and time and their performance implications for the end-application are unclear. An application developer is thus unlikely to be willing to invest the time and effort to do so to leverage the BFT approach.

Keywords: Byzantine Fault Tolerance, Reliability, Distributed Systems

1 Introduction

Byzantine Fault Tolerant (BFT) systems are considered by the systems research community to be state of the art with regards to providing reliability in distributed systems. A BFT system implements a replicated state machine [1] typically consisting of $n = 3f + 1$ replica servers that each provide a finite state machine and execute operations from clients in the same order. BFT systems assume a pessimistic failure model, based on the classic Byzantine generals' problem [2] which provides agreement amongst a set of nodes where at most f nodes display arbitrarily incorrect behaviors, known as Byzantine faults.

BFT systems are attractive because they provide guaranteed safety and liveness properties when the assumption of up to f faulty nodes hold. Early work on BFT systems was widely considered to be impractical for use by real systems because they were either too slow to be used in practice or assumed synchronous environments that rely on known message delay bounds. However, the seminal

work of Castro and Liskov [3], published in 1999, changed this view. This work proposed and implemented *Practical Byzantine Fault Tolerance* achieving an impressive peak throughput of several tens of thousands (null) operations per second, previously thought unattainable. As has been noted by others [4], over the last thirteen years, the research community has seen a flurry of excitement with several efforts to improve the performance and/or cost of BFT replication systems. These efforts include studies aimed at increasing throughput or reducing latency of client requests [5–12, 4, 13], efforts to reduce the number of replica servers needed to withstand f faults to achieve lower replication cost [5, 14, 13], and efforts to boost the robustness of the protocol under both faulty servers and faulty clients [15, 4]. A large majority of these systems [5, 6, 11, 15, 4, 10, 13] are direct descendents of the Castro and Liskov system, hereonin referred to as the PBFT approach (for Practical Byzantine Fault Tolerance). Both the implementations and evaluations of these systems depend on the initial PBFT code base.

Despite BFT's attractive correctness guarantees, BFTs are still rarely used in practice. This is unfortunate, given the ever-increasing need for reliability in real-world distributed systems. More and more applications require high security and reliability to be both trustworthy to users and successful in use (e.g, electronic voting and digital preservation). The lack of wide deployment of state-of-the-art BFT technologies is puzzling. The open-source PBFT code initially provided by Castro and later modified by others has been publicly available for several years, and while readily sized up by the academic community for research purposes, it has not been used in practice in real-world systems.

In this paper, we examine, from the perspective of an application developer, the *practicality*, i.e., feasibility, of using the PBFT protocol and accompanying implementation to provide provable reliability guarantees for a real-world application. Our motivating application is a state-of-the art electronic voting system, offered as a public Internet service. The current version is centralized [16]. Given the critical nature of the application, our aim is to build a system that has *no centralized component*. Every aspect of the system's design should be distributed to avoid single points of attack and failure. Our aim is to leverage the correctness guarantees provided by PBFT systems to improve the security and reliability properties of the system. In such a system, clients (on behalf of users/voters) connect to the voting service, view the election procedures in which they have a right to participate, send the user's vote, and potentially reconnect at a later point to view the progress and/or results of the election. Our aim has been to gauge, from the perspective of a developer in need of providing reliability beyond simple crash-fault recovery, how easily the PBFT approach and accompanying system could be molded to fit the application developer's needs.

We have focused on the original PBFT implementation for several reasons. First, over the past thirteen years, the large majority of research efforts on improving BFT systems have relied on the PBFT approach and implementation. The PBFT codebase is the most stable and complete (in terms of implemented features), and it has been widely studied and extended over several years by sev-

eral research groups. Second, this is the only publicly-available implementation for which a proof of correctness under a formal model has been completed [17]. Third, even as the debate over improving BFT systems continues, the interface to application developers provided by the PBFT middleware remains the same. This means that any later developments in the PBFT system suite can be easily leveraged by applications. Fourth, our particular electronic voting application is written in C; the PBFT code base is written in C++. A recent effort, called UpRight [18] aimed at easing the application developer's effort to make use of BFT technology is written in Java, still has several key features missing (e.g., view changes are unimplemented), and seems to be a work-in-progress that has not seen much development in the last two years. Thus, for a developer wanting to leverage the attractive reliability guarantees of BFT *now*, the original PBFT system offers more promise. Finally, we studied quorum-based BFT systems, using Q/U (e.g., [7]) as well as the study in [19], but found that these approaches are not suitable for our application, as we anticipate a high level of concurrency in our replicated service.

We describe our experience trying to leverage the PBFT approach and code base to enhance the reliability of our e-voting application. We describe several obstacles we encountered and drawbacks we identified in the PBFT approach. One key drawback we identified is that PBFT-based systems assume static membership – ie., clients and replica servers know each other apriori before system initialization. Most Internet services require support for dynamic client management, particularly when the number of envisioned clients is large. The PBFT literature (original as well as all subsequent descendants of PBFT) does not address this issue. Another key drawback is that PBFT leaves state management completely to the application developer, who is required to manually manage a raw memory region, while also issuing notifications to the library before changing memory contents. This may be fine when developing system services, but is not a convenient base for an application. Additionally, PBFT treats a replica server's memory as stable storage, by assuming the use of uninterruptable power supplies [3]. Many Internet application services, particularly an electronic voting system, cannot afford to rely on this assumption and instead require traditional ACID semantics to ensure data stored is consistent and persists despite crashes and faults. The PBFT system suite leaves state management to the application developer. This means that an application developer wishing to make use of an available legacy database to provide ACID semantics is faced with the decision of implementing these semantics into the application from scratch or retrofitting the BFT middleware to interface with and support the legacy database.

In addition to the above, we describe a number of other drawbacks including: the mechanism used by PBFT to handle nondeterminism in applications, the lack of support for stronger cryptography, the lack of support for web-based applications, and others. The description of our experience may seem pedantic, with many minute low-level details, but we provide these here to give the reader a clear understanding, from a holistic systems perspective, of the obstacles faced by a developer trying to put the PBFT system to real, practical use. These are

details that are often considered "not important enough" to warrant attention in many research papers (and prototype implementations, for that matter), usually due to time and space constraints. Nonetheless they can trip up a third-party developer hoping to make use of the novel research prototype. In practice, it is the details that make or break the widespread deployment and use of a system.

We find that while many of the obstacles we describe could be overcome with a "better" or "revised" BFT middleware implementation that is tuned specifically for the needs of the particular application, they require significant engineering effort and time. More importantly, the performance implications of the changes required to meet the application's needs are unclear. For example, we describe how we overcome the first two drawbacks above. While adding support for dynamic client management does not significantly affect system performance, measured in *null* operations per second, retrofitting the PBFT middleware to support a legacy database reveals a throughput performance of *real* operations that is two orders of magnitude smaller than the *null* ones, advertised by prior BFT studies.

To date, only two publications on BFT that we are aware of have noted that reporting null operations per second as throughput is not representative of real applications and thus not helpful to the end-developer [20, 13]. This is understandable, as the focus of most BFT research efforts has not been on end-application use but on improving the BFT middleware itself and null operations provide a basis for comparison. Nonetheless, a developer faced with having to make a slew of modifications to the BFT middleware to get an end-system that has unknown performance properties is hesitant to invest the effort to do so.

This paper makes the following contributions:

- We identify a number of drawbacks in the PBFT protocol suite, from the perspective of an end-application developer trying to leverage PBFT reliability guarantees and we describe solutions to address these. The sheer number of drawbacks severely affects the ease with which a developer can leverage the PBFT approach.
- We present changes we made to the PBFT protocol and implementation to enable dynamic client management, a must for many Internet service applications in use today. We show that these changes can be made with minimal additions to the PBFT protocol, thus not affecting its provable reliability guarantees. We demonstrate, via empirical experiments, that support for dynamic client management can be achieved with minimal performance impact.
- We evaluate the performance impact of retrofiting the PBFT middleware to support ACID semantics via a widely-used legacy database to ease the state management burden of many applications requiring these semantics. We evaluate the impact on performance of this change, and show that for *non-null* operations, the throughput can be many times smaller than the tens of thousands of *null* operations per second presented in prior studies.

The source code for our modifications to the PBFT protocol and implementation is available online at *http://sourceforge.net/p/p2bft*.

2 Background

2.1 Original algorithm

The Castro-Liskov algorithm for Practical Byzantine Fault Tolerance [3] (abbreviated as *PBFT*) is a replication algorithm that can tolerate arbitrary faults. It is based on State Machine Replication [21, 1] where transitions are applied to an instance of the application's state and result in a new, deterministic instance of the state. The general idea is that a group of replicas form a static group that provides a service. At each instance in time, one of them is the primary and is responsible for sequencing the requests, providing total order. This in turn guarantees linearizability [22], which is a correctness condition for concurrent objects where a concurrent computation is equivalent to a legal sequential computation. A view is the epoch where the primary is stable. The remaining replicas monitor client requests and the primary's behavior and, if the latter is found misbehaving, begin a view change procedure to elect a new primary.

The algorithm is asynchronous and provides liveness and safety guarantees when less than a third of the replicas are faulty. More specifically, to tolerate f Byzantine faults, the group needs at least $3f + 1$ members. Safety, formally proved by using the I/O Automaton model [23], guarantees that replies will be correct according to linearizability. Liveness assures that clients will eventually receive replies to their requests. The algorithm does not rely on synchrony to provide safety but does rely on a weak synchrony assumption to provide liveness: that *delay(t)* does not grow faster than t indefinitely. Here, *delay(t)* represents the time interval between initial message transmission *(t)* and message delivery to the replica process. For the protocol to be live, the client is expected to keep retransmitting its request until it finally obtains the reply. Further assumptions include independence of node failures and inability of an attacker to subvert cryptographic protocols.

In normal operation, the client sends a request to the primary. The primary assigns a monotonically increasing sequence number to the request and begins a 3-phase agreement protocol with the other replicas, at the end of which each node executes the request and directly transmits the reply to the client. The latter will accept the reply as correct only when $f + 1$ replies match. The 3-phase protocol consists of the exchange of the following messages, where the target of a multicast is the set of replicas:

1. *Pre-prepare*, multicast from the primary, which assigns a sequence number to a request and forwards its contents
2. *Prepare*, multicast by each replica, agreeing to the sequence number assignment
3. *Commit*, multicast by each replica, which helps guarantee total ordering across views

After the commit, each replica will execute the request and transmit the reply directly to the client. In all of the above message exchanges, the sender is expected to sign the contents with his private key.

Certain optimizations were applied by Castro and Liskov to this basic mode of operation to improve the latency and throughput of the system. First of all, the use of asymmetric cryptography was reduced, by introducing Message Authentication Codes. The client assigns a different key to each replica and sends the key to it, signed with the node's public key. From then on, all requests are accompanied by an 'authenticator', which is a structure that contains one MAC for each replica. This considerably boosts performance, as we confirm in Section 4. Another optimization is the tentative execution of requests before the commit phase. The client cooperates in this mode of operation as it expects $2f + 1$ tentative replies (marked as such by each replica) instead of the normal $f + 1$. If such a quorum is not assembled, the client simply retransmits the request message. As the replicas will in turn retransmit the last reply for this client (which by now should be marked as stable, since the Commit phase should be over), a smaller quorum of f+1 stable (non-tentative) replies may be enough.

Yet another optimization is the special treatment of *read-only* and *big* requests. A request is considered *big* if its size exceeds a configurable threshold, while the *read-only* status is explicitly set by the client. These differentiated requests are multicast from the client to all replicas, to relieve the primary of this burden. The default configuration of the original PBFT implementation sets the threshold to 0, resulting in all requests being treated as *big*. The *read-only* requests are specially treated and are executed as soon as they are received, sequencing permitting, of course. Finally, *request batching* is employed to minimize network usage and agreement latency. A *congestion window* is defined as the number of requests that have been received but not yet executed by the primary; its size is an adjustable parameter of the system. When the primary receives a request message, it calculates the difference between the last locally executed sequence number and the sequence number assigned to the new request. If this difference exceeds the defined *congestion window*, it postpones issuing the *pre-prepare* message, giving itself time to catch up on request execution. Once it does, it includes in a single *pre-prepare* message, as many outstanding request messages as possible, thus minimizing latency due to individual agreement. Note that batched requests capture parallelism from different clients, as each client is allowed a single outstanding request only.

The original PBFT implementation was developed by Castro, and published as open-source along with his dissertation. The environment chosen was: Linux, C++, UDP as the network protocol, the Rabin assymetric cryptosystem, UMAC32 for MACs and MD5 for digests. This implementation defines application "state" as a single continuous virtual memory region. In fact, it splits this region in two, the first part for the internal library needs and the second part for the application. The library has a subsystem that manages the synchronization and checkpointing of this state using copy-on-write techniques and Merkle (hash) trees [24]. The general idea is that the state is divided in pages of equal length. A hash tree is formed where the leaves are the actual data pages while the inner nodes are the hashes of their children (either of the data pages at level *height-1*, or of the hash text at smaller depths). At the root, a single digest uniquely identi-

fies the complete memory region. A checkpoint message communicates this root hash to the rest of the replicas to agree that the state is properly synchronized. If a peer finds itself out of sync, an efficient tree walking algorithm is started from the root, to identify the (hopefully few) data pages that are different and have them retransmitted by the rest of the group.

The server part of an application wishing to use PBFT services is expected to initialize the library and then wait for up-calls from it, to service requests and produce replies. While executing, it has free read access to arbitrary memory regions inside the "state" managed by PBFT, but is expected to notify the library **before** making any changes.

2.2 Reasoning about the default implementation

It is very hard to reason about the behavior of a distributed system when it is run on multiple hosts, without a common clock. To address this hardle, we modified the library to be able to run multiple times on the same host, using different port numbers. We also created a log of all messages exchanged between replicas that, given the common clock, allowed us to reason about the behavior of the system, by creating UML sequence diagrams depicting the operation of the system as a whole. All further observations are based on this groundwork.

2.3 Authenticators and Erratic Recovery Behavior

To better understand the recovery process, we stopped and restarted a replica, using the default optimal configuration. We immediately witnessed erratic behavior in the recovery process, which started and re-synchronized the state to the latest checkpoint, but was unable to execute the few requests remaining in the log after that point because they failed the authentication test. Upon investigation, we found that the use of authenticators, introduced for efficiency, impeded the recovery process, because the transient state of the restarted replica had no recording of the authenticators to use for validating client requests. The solution the existing system implements, is the periodic retransmission of the authenticators from each client to all replicas, based on a timer. This way, once the recovering replica receives the authenticators of the clients, it will be able to resume the recovery process from the next checkpoint. The only way to lower the timeframe for this service interruption, is to reduce the authenticator retransmission timeout, which results in increased load for the network. We investigated other solutions including on-demand retransmission of the authenticators; we did not pursue this however, because retransmissions can introduce denial-of-service vulnerabilities, as a faulty replica could simply bombard the clients with authenticator retransmission requests.

2.4 PBFT Behavior on UDP Packet Loss

The definition of a Byzantine fault is any possible fault, including an error as trivial as a UDP packet loss. This creates interesting behaviors. We observed

that UDP packets were indeed lost in our experiments, even in the loop-back interface, due to congestion caused by stress-testing the system. The impact of this is profound, as such an error will leave a replica lagging behind in transaction execution and will cause the recovery process to commence on the next checkpoint. Although elegant in theory, this approach will not work in a production environment where it is unacceptable to lose replicas from such trivial errors. One obvious solution would be the use of TCP for its reliability; that however is not trivial to implement as it requires switching from a message-oriented network layer to a connection-oriented one. The use of SCTP, which allows reliable message-oriented communication looks more suitable, as long as all replica hosting platforms implement it of course.

The *big* request optimization described above combined with a trivial UDP packet loss can adversely affect the robustness of the system. In this case, *big* requests are multicast to all replicas only once, from the client. The primary will then use only the digest of the request body for further communication with the rest of the replicas. Consider what happens if one of the packets traveling from the client to one of the replicas is dropped on the way. All replicas will begin the three-phase protocol to commit and execute the request, but when execution time comes, the replica that missed the request body will be unable to execute, and will be stuck at this point until the next checkpoint arrives and the recovery process kicks in. For a request not marked as *big* though, the process is different and more stable. Here, if the request from the client to the primary is dropped, the client will timeout and retransmit the request, resulting in a request execution workflow where either all or no replica at all participates.

2.5 PBFT Handling of Non-determinism

In the original PBFT implementation, a feature was introduced to resolve the non-deterministic characteristics of most applications. The primary makes an application-specific up-call, which returns a set of values that are attached by the primary to the Pre-Prepare message. This data becomes common to all replicas executing the request, thus providing deterministic behavior on request execution. Subsequent work on the PBFT protocol [25] added an extra mechanism to validate this data on each replica. A new application-specific up-call was established that, when passed the non-deterministic data, is expected to validate it and return success or failure. For example, the primary attaches the system clock to the Pre-Prepare message and each replica validates the passed value against its own clock to make sure it is appropriate.

However, the handling of non-determinism described above introduces a subtle issue. It is not always clear how the application can validate the non-deterministic data passed to it via the new upcall. The hurdle for such a validation is the instance in time it is supposed to happen. In the normal, fault-free lifetime of a request, the validation happens as soon as the Pre-Prepare message is received, which is almost immediately after it is transmitted. Thus validating against a time delta is viable. However, when a request is replayed from the log during recovery, the time drift can be quite large and validating using a time

delta will fail and impede the recovery process. A solution to this issue would be to differentiate message processing for the recovery process and completely skip non-deterministic data validation during recovery. This however is again a non-trivial exercise, as message execution in the original PBFT implementation is completely agnostic to message origin.

3 PBFT Deployment Drawbacks, Obstacles, and Solutions

3.1 Dynamic client membership

The existing PBFT protocol and implementation assumes completely static membership where each node in the system, client or replica, needs a priori knowledge of the address, port, and public key for every other node. Although this approach was sufficient for the proof-of-concept prototype, it is too limiting for real world use, particularly Internet service applications with a large number of clients. Our goal is to remedy this to enable clients to join and leave the replicated service dynamically, while letting the replicas remain statically bound to one another. The end result is that clients only need information regarding replicas, but no information regarding other clients, allowing for a more scalable deployment.

To achieve support for dynamic client membership, replicas need to identify each client in an identical (deterministic) manner. This leads us to store the client identifiers in the shared state of the service (i.e., in the continuous memory region). When a client requests to join or leave the group, each replica needs to process the request using the same version of the shared state. Thus, all such client requests need to be totally ordered, at least with respect to one another.

We define two special system requests, namely a *Join* and a *Leave*, which follow the same life-cycle as all other application-level (client) requests. This results in a single total order across all requests, application or system, fulfilling our requirement. The *Join* and *Leave* system requests are processed by the middleware library and are invisible to the application.

We introduce a level of indirection between what the PBFT library already uses as a node identifier and what the client reception module assigns to new clients, for efficiency of message evaluation. Instead of using a single address range of $[0..max\_clients]$, an arbitrary identifier is assigned to each new client and a table maps this number to the index in the array of client and server node entries. This way, when a client request arrives, the system first checks to see if the identifier exists in the redirection table before going into the more lengthy process of verifying its signature or authenticator.

Originally, our idea was for the client to multicast a simple Join system request to all replicas, carrying its address, its public key and a random nonce, signed with its private key. Each replica would assign the same new identifier and transmit it back in the reply. However, a malicious client could initiate an infinite number of connections, using phony addresses, thus exhausting the

bounded maximum number of node entries in each replica. To address this vulnerability, we improve the connection process by splitting the Join operation into two phases. In the first phase, the client submits its data as previously described and awaits a challenge. Upon receiving the challenge, the client calculates a response and transmits it back to the replicated service in the second phase of the Join. Only then will the replicas add the client to the system as a full member. This approach ensures the client indeed owns the address he claims, since receiving the challenge is necessary to compute the response.

We also add an application-level identification buffer to the Join message. This buffer is passed to the application for authorization. It might include, for example, an encrypted user id and password. The application then returns an identifier to be associated with this client (e.g., the user id). The middleware guarantees that only a single session can be active at a time for this specific identifier, by terminating all previous sessions when a new one is established. This way, even in a distributed denial of service attack, the attacker can only establish as many sessions as the number of credentials he has managed to obtain.

The Leave system request is much simpler. It simply instructs each replica to remove the client from its internal tables. All further communication with the service is prohibited for this client.

We add timeouts to enforce cleanup of stale sessions once the node structures are full. To achieve some common ground regarding time across all replicas, all requests are timestamped with the time of the primary. When each request is executed, its timestamp is recorded for each client. When a join request arrives that cannot be serviced because the client/server node table is full, a cleanup process is started that will locate all clients with a last executed request older than the current join request minus a configurable threshold. All such sessions are cleared to make room for the new connection. If no such stale sessions are found, the new Join request is denied.

Note we have enhanced the PBFT protocol with support for dynamic client membership without changing the inherent properties and message exchanges of the protocol. Thus, our changes do not affect the safety and liveness guarantees offered by PBFT.

3.2 A higher level state abstraction

In a replicated state machine, the term 'state' is an abstract definition of the persistent workspace of the application. PBFT defines state to be a continuous virtual memory region where both the application and the middleware library store their non-transient state in contiguous non-overlapping partitions. The middleware library has full access to this memory region while the application code is not executing, since it is responsible for managing replication and synchronization of this state across replicas. The application, on the other hand, has free read access to it, but is required to notify the library before making changes to any region, thus permitting copy-on-write optimizations of state synchronization.

While this approach relieves the application considerably from having to deal with state synchronization, it creates a number of questions which the application developer must face: What can a modern application do with just a pointer to a memory region? How is this state persistently stored on disk when the service stops? And how does the developer avoid the havoc caused by a misbehaving application which fails to notify the library before modifying memory? To address these questions in a satisfactory manner, we decided to adapt an embedded relational database engine, to intervene between the PBFT middleware library and the application. This way, the application will have SQL-level access to its state and the embedded engine will take care of interfacing with the PBFT library to satisfy its requirements.

In our search for an embedded relational database engine, the major feature we were after was storage of data in a single file, which we could map to virtual memory. We selected SQLite [26] because it exhibits this feature and because it is mature and widely deployed. SQLite is an embedded, in-process library that implements a self-contained relational database engine using SQL as its command language and a C call level interface for the application. It stores all data objects in a single database file that is binary compatible across machine architectures (endianness) and word sizes.

In SQLite's quest to be a multi-platform product, the authors have defined an abstraction layer called VFS (Virtual File System), that sits between the relational engine and the operating system. By hooking into this subsystem, we can manage memory mapping and perform PBFT-required memory modification notifications, as well as re-implement non-deterministic functions, such as system time and random values, by using the upcalls described in Section 2. Interaction with VFS is illustrated in Figure 1.

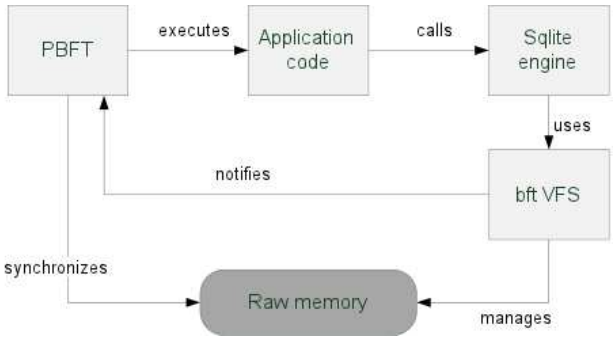

Fig. 1. SQLite with its VFS inside a PBFT application

SQLite uses two disk files to manage the database, for reliability reasons. The first file is the actual database, which we map to virtual memory. The second file is the rollback journal (or write-ahead-log, in a different mode of operation), which is used to rollback failed transactions. We leave this second file to be stored

on disk, since it allows the engine to recover in the case of system failure and it is not actually part of the application state. The database file is synchronized with its disk image on transaction commit.

We gain many advantages with this approach. First, a committed transaction will be durable, even in the case of a system crash. That is, when the replica node restarts operation, its state will include the last committed transaction, and PBFT recovery will commence from this point. Second, even if the node is to be removed from the replicated service, its data will be usable on its own, being just another database file. Moreover, an uncommitted transaction will be rolled back on the next attempt to access the database file, from the replicated service or on its own. These advantages are simply the by-product of the ACID semantics that SQLite provides and excellent reasons why developers will likely want to take advantage of it.

One obstacle we faced was that while SQLite can freely manage the growth and shrinkage of its database file, PBFT is not so permitting, because it requires knowledge of the size of the memory region that represents the state, during its initialization. To alleviate this, we use a sparse file that is defined to be a large enough size on initialization, without actually occupying that space on disk, a solution that is reasonable in modern 64-bit operating systems with large virtual memory address ranges.

The application code now simply passes the name of the database file to the PBFT initialization function responsible for starting up the replica server and setting up any data structures needed by the middleware. The function returns to the application code a standard SQLite database handle. Using this handle, the application can call standard SQLite library functions (e.g. sqlite3_exec, sqlite3_prepare_v2, sqlite3_step) to access the database while executing during the appropriate PBFT upcall. This way, an application already using SQLite is immediately portable to the PBFT middleware with only minor changes to the initialization code. Our approach thus guarantees that, whenever the application is called to execute a request, it will have a database consistent with all other replicas. This is achieved by the PBFT middleware library, which manages the raw memory content where the database file is mapped, as it was designed to.

3.3 Remaining issues

Cryptography Applications requiring strong cryptography, such as private key generation and storage on the server side of the application, are not well supported by the current PBFT implementation. For key generation, strong random values are required. Unfortunately, even if the primary obtains such strong randomness from its local OS services, for example via /dev/random, there is no way such values can be verified from the remaining replicas, by their very definition of being random. Because of this, an adversary can obtain access to one of the execution replicas, wait until it becomes the primary and use predetermined values instead of random values. In this manner, the adversary can trigger the generation of well-known private and public keys and thus violate confidentiality. To alleviate such attacks, one solution would be to enforce a

threshold signature scheme [27] for such authentication requirements, provided for by the middleware library. In such a scheme, private key information for each replica would never be transmitted over the network, as it would not be stored in shared state. In a $(f + 1, n)$ (where $n = 3f + 1$) threshold signature scheme, the set of n replicas would collectively generate a digital signature despite up to f byzantine faults. The PBFT protocol would have to be modified to provide for such cryptographic operations.

Another confidentiality issue is the matter of protecting storage of sensitive information. This has been studied by Yin et al [5], who propose separating the agreement part of the PBFT protocol from the execution part, while also adding an intermediate cluster of 'privacy firewall' nodes. In this layout, $3f+1$ *agreement* nodes receive the client requests and forward them to $2f + 1$ *execution* nodes for execution. To ensure that a faulty execution node cannot disclose sensitive information, an $h + 1$ rows by $h + 1$ columns *privacy firewall* set of nodes is positioned between the agreement and execution cluster, which allows tolerating up to h faulty firewall nodes. This approach however, increases both deployment complexity and request execution latency.

Stateless applications only The original PBFT implementation purposely ignores the notion of client-specific state. This, however, severely limits the target applications to those that are either stateless in nature, or manage session state on their own using their global state abstraction; the latter will need to pass session identifiers inside the request and reply bodies, without any assistance from the middleware library. This is not an inherent limitation of the State Machine Replication approach. It is simply a consequence of the lack of appropriate mechanisms in the PBFT library. With our addition of application level sign-on messages to the protocol, resulting in identification of specific sessions, a library-level subsystem can now be developed that will map parts of the state to a specific session. This enables easier porting of stateful applications to the BFT world.

Web applications Our end goal is to provide a web application to end users, which provides them hassle-free access to the server counterpart of the e-voting service. We aim to achieve this while providing end-to-end BFT semantics. To this end, the browser-hosted part of the application, typically written in JavaScript, will have to directly access each and every replica. This communication however cannot be carried over UDP because this protocol is not allowed in the JavaScript runtime environment. Moreover, binary messages are highly inconvenient in this context. Higher level protocols, such as WebSocket, and structures like JSON or XML need to be used. Support for these technologies needs to be incorporated in the middleware library, a task not so trivial because of the need to switch from a point-to-point message-based communication to a connected channel-oriented communication. Additionally, cryptographic functions will need to be available in the browser-hosted client part, which requires transitioning from Rabin to more widely available cryptosystems, such as RSA.

Additionally, we aim to have the replicas placed in different physical locations, to obtain real independance of faults caused by network partitions. This requirement dictates operation in a Wide Area Network environment, where the quadratic message complexity of PBFT will most probably prove costly regarding request latency. Although we tried to simulate a WAN deployment scenario using BFTsim [19], the simulator could not scale to a large enough number of nodes (> 100) to obtain meaningful results. This issue has been studied in [28], but no open-source implementation is readily available.

Summary: The above issues can be overcome, but require a significant amount of engineering effort. An application developer wanting to leverage and deploy PBFT *now* is likely to be unwilling to invest the time and effort required to retrofit the PBFT approach to match the needs of his/her application.

4 Evaluation

In this section we present empirical measurements of the PBFT library, both with and without our modifications supporting dynamic client and seamless state management for applications requiring ACID semantics provided by a legacy database. We test the PBFT library and our modifications to it on a cluster of 8 machines connected with a 1Gbit Ethernet switch. The first four machines are Intel Xeon E5620 at 2.40 GHz under CentOS 5.5 with Linux kernel 2.6.18-194. The remaining four are Intel Core 2 Duo E6600 at 2.40 GHz under Debian 5.0 with Linux kernel 2.6.26. All eight machines run 64 bit versions of their corresponding operating systems. Ping roundtrip time is measured at 134-183 microseconds between all hosts. Bandwidth is measured, using iperf, at 938 Mbits/sec. For all tests, we generate a server and client executable using a particular library configuration set so as to measure the effect of turning on or off a particular optimization and/or modification. We design the client to connect to the library and wait for a signal. On signal reception, it records the current time, starts its operation and then measures and reports elapsed time. To coordinate all processes running on different hosts while at the same time collecting and aggregating measurements, we implement a test framework using Python and netcat, where the latter runs on each host and allows a single controller to submit scripts (i.e., experiments) and collect the results.

4.1 Non-SQL Experiments

We first conduct an experiment without the SQL state abstraction modifications we made to benchmark the original PBFT implementation. Our goal is to measure the impact on system throughput of turning on/off the optimizations described in Section 2. Recall that the use of certain optimizations (such as the use of MACs and special handling of *big* requests) increases performance at the cost of decreased robustness (e.g., slow recovery) of the system.

We generate and test a series of PBFT library configurations, shown in Table 1. The first configuration is the default configuration preferred and recommended by Castro, with all optimizations enabled, including the use of MACs,

Name	TPS	StDev
sta_mac_allbig_batch	17.014	66
sta_mac_allbig_nobatch	1.051	56
sta_mac_noallbig_batch	3.030	57
sta_mac_noallbig_nobatch	1.109	103
sta_nomac_allbig_batch	1.291	4
sta_nomac_allbig_nobatch	1.199	12
sta_nomac_noallbig_batch	992	2
sta_nomac_noallbig_nobatch	1.186	7
nosta_nomac_noallbig_batch	988	1
nosta_nomac_noallbig_nobatch	1.205	1

Table 1. PBFT library configurations we test. sta=static clients, transactions are null requests/responses of 1024 bytes.

special treatment of all requests as *big* requests, and request batching. Since batching is the only optimization for which we did not observe faulty behavior, we isolate it and test all other combinations of configurations with batching enabled and disabled, to show its impact. The last four rows of Table 1 depict the most robust configurations (use of MACs and big request handling turned off). Since our particular application has stringent security and reliability requirements, we choose to measure the impact of adding support for dynamic client management using these configurations. We believe other Internet service applications with similar high security and robustness needs would need to run the PBFT library using these configurations. The client and server programs built to measure throughput transmit null requests and responses of varying sizes, of 256, 1024, 2048, and 4096 bytes. We test the system using 12 clients spread evenly across 4 machines while being serviced by 4 replicas, each running alone on a single host. In all cases, IP-level multicasting was turned off, as the networks we are targeting (WANs) do not support it. The results for each of the request and response sizes are similar, so for brevity we show a representative plot, for size of 1024 bytes in Figure 2.

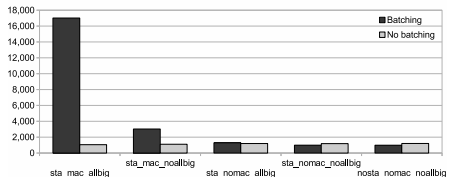

Fig. 2. PBFT tests

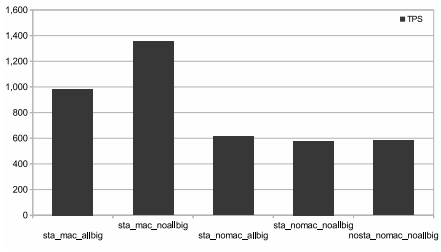

Fig. 3. PBFT + SQL benchmark

From Table 1 and Figure 2, it is clear that the first configuration, which is the default configuration of the PBFT library with all optimizations turned on achieves the best throughput performance. In our experiments, this configuration achieves approximately 17000 null operations per second, while for the most robust configurations the throughput drops to about 1000 null operations per second.

We observe that disabling the batching optimization seriously affects performance when using MACs. When switching to signing with private keys, the delay introduced is so large that batching can no longer assist in any way. Moreover, when disabling big request handling, performance drops to 18% of the optimal, while disabling the use of MACs causes performance to drop to 7.5% of the optimal respectively. Disabling both big request handling and MAC use causes performance to drop to 6% of the optimal. While we observe a difference in performance amongst these configurations where some subset of optimizations is turned off, the bottom line is that performance takes a big hit when turning off any of the optimizations. However, for an application with high security requirements, we conjecture robustness is favored over performance.

We evaluate the impact on performance of adding support for dynamic client management using the most robust configurations. The performance decrease is 0,5% (988 vs 992), which is negligible. This negligible decrease in performance is attributable to the cost of accessing the redirection table that converts assigned customer ids to indexes in the tables tracking participating nodes (clients and servers). We emphasize that the above tests are *artificial* because they are testing "null" operations. The software on the replica spends no time executing application code; it simply manages the network protocol. The large majority of prior BFT studies present throughput in terms of null operations per second. This is understandable as the focus is on providing a baseline benchmark against which varying BFT protocols can be compared, but is not helpful to the application developer who needs to understand how the system would behave using real application requests.

4.2 SQL state abstraction experiments

In this subsection, we evaluate the performance of adding seamless state management for applications requiring ACID semantics provided by a legacy database. Null operations are thus not realistic to use in this setting. For our client application request we choose the insertion of a single row into a database table. This is the operation our e-voting service must perform to record a user's vote in an ongoing election. The tuple inserted into the database includes a simple key and value text (representing voter identity and accompanying vote), in addition to a timestamp and a random value. We purposefully added the timestamp and random value to test that replies are indeed identical across all replicas. For this experiment, we enable request batching and vary turning on and off the remaining options (use of MACs, big request handling, and support for dynamic clients). ACID semantics are provided using the rollback journal mode of

SQLite. Throughput performance, measured as database insertion transactions per second, is illustrated in Figure 3.

In this experiment, the big request handling optimization pays no dividends because the system now spends time executing a real, non-null request which requires accessing the hard disk. This dominates the overall request execution lifetime. At any rate, the most robust configuration with dynamic clients enabled is now at 43% of the best (sta_mac_noallbig). Since disk access is a big factor in this experiment, we perform two more experiments to isolate its impact. In these experiments, we measure the most robust configuration (where the use of MACs and big request handling are disabled) with dynamic clients and ACID semantics (as above) and we measure another configuration without ACID semantics (no rollback journal and no flushing to disk on each operation). The ACID version achieves 534 TPS while the No-ACID version achieves 1155 TPS, an approximately $2x$ performance boost.

Summary: The optimizations turned on by default in the PBFT library, lead to the high throughput numbers reported in prior studies, but as we have shown in Section 2, using some simple fault scenarios (such as UDP packet loss), the high performance numbers come at the cost of decreased robustness of the system. Moreover, the performance numbers reported by a large majority of prior BFT studies are based on a metric of null operations per second. This is not a helpful metric for the end-application developer, particular for a developer whose application makes use of a legacy database for ACID semantics.

5 Related Work

Since the 1999 publication on PBFT by Castro and Liskov [3], there has been a flurry of research activity focused on improving the BFT middleware performance [5–8, 10–12, 4, 9, 13], replication cost [5, 14, 13], and robustness under both faulty servers and faulty clients [15, 4]. A large majority of these systems [5, 6, 11, 15, 4, 10, 13] are direct descendents of the Castro and Liskov PBFT system and reuse and build upon the Castro codebase. Thus, the obstacles we encountered as application developers in using the PBFT system apply to its descendents as well.

Wood et al. [13] write "no commercial data center uses BFT techniques despite the wealth of research in this area" and posit that this is due to the high cost of replication required by BFT protocols. They aptly point out that, for applications such as web servers and database servers, it is the execution of client requests and not the agreement of request ordering that dominates the performance of a BFT protocol. They propose lowering the number of active execution replicas to $f + 1$ by using virtual machines as execution nodes and ZFS snapshots for quick state checkpointing. When the $f + 1$ replicas produce inconsistent replies, a paused execution node is revived and starts executing requests immediately. The middleware library fetches the state needed by these requests on demand, to amortize the cost of state transfer. The paper claims that for applications running over a WAN environment, the time to perform

state transfer is minimal compared to WAN latencies. The focus of the paper is on reducing replication cost while maintaining good performance. While this is welcome for an application to be deployed in a data center, the paper does not address how the application developer can easily make use of the system, stating simply that applications must be rewritten to take advantage of the system.

Clement et al. [18] introduce *UpRight*, with the goal of making it easy for application developers to convert a crash-fault tolerant application into a BFT application. It includes a number of state-of-the-art BFT techniques, including separation of agreement from execution, insights from the Aardvark protocol [4] on dealing with faulty clients and alleviating denial-of-service attacks, as well as more flexible state management (but not at such a high level as a relational engine). It also allows individual tailoring of crash-fault (Up) and arbitrary-fault (Right) tolerance. Unfortunately, it is still a work in progress. with several key features missing (e.g., view changes are unimplemented) and does not seem to have seen much development since March 2010 [29], so it is not helpful to a developer wishing to make use of BFT techniques now.

Several attempts have been made to address the inability of replicated BFT services to mesh with the rest of the infrastructure in today's multi-tier world. Merideth et al. [30] introduced *Thema*, which aims to mask BFT complexity from the application developer of web services based applications. An agent, visible to the unaffected outside world, plays the role of the client of a BFT system. Additionally, a proxy collects the multiple out-call requests from the replicas of a BFT system, and issues the actual out-call on their behalf, returning the reply when available. Unfortunately, both the agent and the proxy are centralized components which are inappropriate for applications such as ours which require completely distributed design.

Pallemulle et al. [31] focus on interoperability between BFT systems, while enforcing fault isolation and introduce a new protocol, named *Perpetual* to achieve this. Sen et al. [20] in a system called Prophecy, designed to increase BFT performance, introduce a *Sketcher* component, that tries to trade space for performance, by storing a historical log of request/reply pairs and allowing the application to differentiate its requests, asking for possible log-based replies. In its distributed incarnation, *D-Prophecy* is simply an attempt to avoid re-execution of repetitive requests. In the centralized version, *Prophecy*, the *Sketcher* completely avoids BFT access but now becomes a single point of failure.

Amir et al. [28] introduce *Steward*, a hierarchical BFT architecture, that tries to scale BFT to a wide-area network, by introducing an abstraction layer above PBFT using a Paxos-based protocol. It uses a threshold signature scheme to ensure the recipient of a cross-domain message that enough replicas at the originating site agreed with the request. Both these features are welcome to security-conscious Internet application services. Unfortunately, no source code is available.

Vandiver et al. [12] and Garcia et al. [10] introduce middleware for BFT database replication. Incoroporating legacy databases into a BFT system is important for a wide range of Internet applications. Unfortunately, both sys-

tems assume closed systems with a finite number of clients. The developer of an Internet-facing application service must still deal with the issue of having end-user clients issue requests to the replicated database system. Either these systems need to provide support for dynamic client management or they must offload the Internet-facing application component accepting customer/user requests to a centralized component, something not appropriate for our particular application.

Finally, Guerraoui et al. [17] introduce a new abstraction allowing for the construction of new BFT protocols with a fraction of the code currently necessary, thus vastly simplifying the BFT researcher's task. Having waded through the 20,000 lines of PBFT code, we applaud this effort and emphasize here the need to simplify the end application developer's task as well.

6 Conclusion

This paper is a call to the research community to look more closely at BFT middleware from the perspective of a real-world application developer. Our experience in trying to apply the PBFT approach to a real-world application with stringent security and reliability needs reveals a slew of difficulties that the application developer must face if he wants to use even the highly- studied and several-times extended PBFT protocol and codebase upon which a large majority of subsequent BFT middleware is based. While the difficulties encountered by the developer can be overcome, they require significant engineering effort and have unclear performance ramifications. These two characteristics are likely to make the developer hesitant to invest the effort to leverage BFT techniques.

The systems community prides itself on building and measuring real systems. We believe that improving BFT middleware performance and robustness remain important. However, if BFT middleware are to see widespread deployment in real-world systems, then the research community needs to focus on the *usability* of BFT algorithms from the perspective of the application developer.

References

1. Schneider, F.: Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Computing Surveys **22**(4) (Dec 1990) 299–319
2. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM TPLS **4**(3) (July 1982) 382–401
3. Castro, M., Liskov, B.: Practical byzantine fault tolerance. In: OSDI. (February 1999)
4. Clement, A., Wong, E., Alvisi, L., Dahlin, M.: Making byzantine fault tolerant systems tolerate byzantine faults. In: NSDI. (April 2009)
5. Yin, J., Martin, J.P., Venkataramani, A., adn M. Dahlin, L.A.: Separating agreement from execution for byzantine fault tolerant services. In: SOSP. (Oct 2003)
6. Kotla, R., Dahlin, M.: High throughput byzantine fault tolerance. In: DSN. (Jun 2004)

7. Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M., Wylie, J.: Fault-scalable byzantine fault-tolerant services. In: SOSP. (October 2005)
8. Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: Hq relication: A hybrid quorum protocol for byzantine fault tolerance. In: OSDI. (Nov 2006)
9. Distler, T., Kapitza, R.: Increasing performance in byzantine fault-tolerant systems with on-demand replica consistency. In: EuroSys. (Apr 2011)
10. Garcia, R., Rodrigues, R., Preguica, N.: Efficient middleware for byzantine fault tolerant database replication. In: EuroSys. (Apr 2011)
11. Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: Speculative byzantine fault tolerance. In: SOSP. (Oct 2007)
12. Vandiver, B., Balakrishnan, H., Liskov, B., Madden, S.: Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In: SOSP. (Oct 2007)
13. Wood, T., Singh, R., Venkataramani, A., Shenoy, P., Cecchet, E.: Zz and the art of practical bft. In: EuroSys. (April 2011)
14. Distler, T., Kapitza, R., Popov, I., Reiser, H., Schroder-Preikschat, W.: Spare: Replicas on hold. In: NDSS. (Feb 2011)
15. Amir, Y., Coan, B., Kirsch, J., Lane, J.: Byzantine replication under attack. In: DSN. (Jun 2008)
16. Kiayias, A., Korman, M., Walluck, D.: An internet voting system supporting user privacy. In: ACSAC. (Dec 2006)
17. Guerraoui, R., Knezevic, N., Quema, V., Vukolic, M.: The next 700 bft protocols. In: EuroSys. (Apr 2010)
18. Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: SOSP. (Oct 2009)
19. Singh, A., Das, T., Maniatis, P., Druschel, P., Roscoe, T.: BFT protocols under fire. In: NSDI. (2008)
20. Sen, S., Lloyed, W., Freedman, M.: Prophecy: Using history for high-throughput fault tolerance. In: NSDI. (April 2010)
21. Lamport, L.: The implementation of reliable distributed multiprocess systems. Computer Networks **2** (1978)
22. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM TPLS **12**(3) (July 1990) 463–492
23. Lynch, N.: Distributed Algorithms. Morgan Kaufmann (1996)
24. Merkle, R.: A digital signature based on a conventional encryption function. In: CRYPTO. (1987)
25. Castro, M., Rodrigues, R., Liskov, B.: BASE: Using abstraction to improve fault tolerance. ACM TOCS **21**(3) (August 2003)
26. : Sqlite embedded database engine. http://www.sqlite.org
27. Desmedt, Frankel: Threshold cryptosystems. In: CRYPTO: Proceedings of Crypto. (1989)
28. Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-rotaru, C., Olsen, J., Zage, D.: Steward: Scaling byzantine fault-tolerant systems to wide area networks. In: DSN. (2006)
29. : Upright: Making distributed systems up (available) and right (correct). http://code.google.com/p/upright/w/list
30. Merideth, M., Iyengar, A., Mikalsen, T., Tai, S., Rouvellou, I., Narasimhan, P.: Thema: Byzantine-fault-tolerant middleware for web-service applications. In: SRDS. (October 2005)
31. Pallemulle, S.L., Thorvaldsson, H.D., Goldman, K.J.: Byzantine fault-tolerant web services for n-tier and service oriented architectures. In: ICDCS. (June 2008)