BYZANTINE FAULT TOLERANT EXECUTION OF LONG-RUNNING DISTRIBUTED APPLICATIONS

Sajeeva L. Pallemulle[1], Ian Wehrman, Kenneth J. Goldman[1]
Department of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO 63130 USA
{sajeeva, iwehrman, kjg}@cse.wustl.edu

ABSTRACT
Long-running distributed applications that automate critical decision processes require Byzantine fault tolerance to ensure progress in spite of arbitrary failures. Existing replication protocols for data servers guarantee that externally requested operations execute correctly even if a bounded number of replicas fail arbitrarily. However, since these protocols only support passive state machine replication, they are insufficient to support continued correct execution of autonomous long-running distributed applications.

Building on the Castro and Liskov Byzantine Fault Tolerance protocol for replicated state machines (CLBFT), we present a practical algorithm for Byzantine fault-tolerant execution of autonomous distributed applications. The algorithm supports replicated clients that actively execute application logic by issuing operation requests on replicated data servers as well as other replicated clients. Our work facilitates dynamic upgrades to replica groups, supports both synchronous and asynchronous operation requests, and provides fault isolation between replica groups with respect to both correctness and performance. The algorithm scales well to large replica groups with only twice the latency and message complexity as compared to CLBFT, which supports only unreplicated clients.

KEY WORDS
Fault tolerant systems, Byzantine agreement, autonomous distributed applications

1 Introduction

Long-running distributed applications that automate critical decision processes (e.g. automated stock trading, environmental controls, and missile defense) must continue to make progress in the presence of arbitrary (Byzantine) failures. Recent advances in distributed systems research have resulted in practical algorithms [1, 2, 3] for constructing replicated data servers that continue to operate correctly in the presence of a bounded number of Byzantine failures. These data servers are passive deterministic state machines that execute operations only in response to external requests from unreplicated clients, which are susceptible to arbitrary failures clearly not covered by the replication protocol. While systems based on these algorithms [4, 5] provide high availability and data integrity, such systems do not guarantee continued correct execution of applications that access those systems. While replicated state machines may be adequate for short-term applications (e.g. end user lookups), they are not sufficient for critical long-running autonomous applications where the active computation as well as data access must be fault tolerant.

This paper presents an algorithm for Byzantine fault-tolerant (BFT) execution of autonomous distributed applications. Our algorithm supports client replica groups that interact with both replicated data servers and other client replica groups. The algorithm provides fault isolation between the replica groups, ensuring that even *compromised* replica groups (with more faulty replicas than the tolerance level) cannot disrupt the operation of correct replica groups.

1.1 Problem Description

The problem of ensuring Byzantine fault tolerant execution of application logic on a client replica group is surprisingly difficult and not easily solved with simple extensions to CLBFT. To illustrate the complexity of the problem, we consider some simple solutions and their limitations.

The most simple approach would have each replica in the client group act as a separate client to the replicated server. Each client replica would request the operation independently, and the group would vote on the final result. However, this approach is inefficient and can only work for idempotent operations since each operation is executed more than once. In addition, a faulty client replica could issue arbitrary operation requests to the server.

One might consider using a primary client replica to act as a client of the server group on behalf of the client group. This would support mutating operations, but the client primary could be faulty and issue invalid requests. In addition, a client replica would not be able to distinguish between its primary being faulty and the server not responding. This would allow a compromised server group to force a potentially expensive configuration change at the client to change the client primary.

A further refinement would be to have each client replica directly send the request to the server. Each server replica would wait for a quorum of requests before accepting it as a valid request. After performing the request, the server replicas would respond to all the client replicas directly. Each client replica would wait for a quorum of server replies before accepting the result. However, a com-

[1]Supported in part by National Science Foundation grant 0305954.

promised server group could send quorums for different result values to different client replicas and cause the client replica states to diverge. Furthermore, utilizing a quadratic multicast for server replies would be highly inefficient.

One way to preserve the consistency of replica state in the face of a compromised server group is to have the client replica group vote on the result. However, if the server only responds to a subset of client replicas or if the server sends quorums for different result values to subsets of client replicas, the client replica group would deadlock on this vote. Without a mechanism to allow all non-faulty client replicas to abort the vote, this deadlock would not be resolved.

To avoid the inefficiency of quadratic multicast, assigning a designated responder at the server would be desirable, with the multicast approach used as a backup strategy only if the responder fails. However, a mechanism to change the designated responder would be needed, and it is important to ensure that faulty nodes in a correct client group cannot force unnecessary multicasts.

1.2 Background

Our algorithm builds upon CLBFT, a practical Byzantine agreement protocol for replicated deterministic state machines. The CLBFT algorithm uses $3f + 1$ replicas, where at most f can be faulty. Messages can be delayed, provided that the length of message delays does not increase faster than time (a weak assumption). Cryptographic techniques [6, 7] are used to verify authenticity of messages, and message digests [8] are used to reduce message size. The CLBFT algorithm for a mutating operation works roughly as follows (read operations require less communication). A client sends its request to a designated *primary* replica, which appends a sequence number and forwards it to the replicas in a *pre-prepare* message. Since the primary may be faulty, the replicas multicast a corresponding *prepare* message to each other, to ensure that all were given the same request and sequence number. Upon receiving $2f$ prepare messages matching the pre-prepare message it received from the primary, a replica multicasts a *commit* message to all the replicas. When it has matching commit messages from $2f + 1$ replicas (possibly including itself), a replica executes the requested operation and sends the result to the client. Upon receiving $f+1$ matching replies, the client accepts that return value. If a client times out waiting for a reply (perhaps due to a faulty primary), it multicasts its original request to all the replicas. The replica starts a progress timer if the operation has not yet executed. Also, if the replica has not yet received a preprepare, it forwards the request to the primary. When the operation completes, it replies to the client with the return value. If progress under the current primary is unsatisfactory, the replicas change the primary in a *view change* operation. Since view changes are expensive, progress timers adapt to prevent frequent view changes.

1.3 Contributions

We present a practical algorithm for Byzantine fault-tolerant execution of long-running autonomous distributed applications in which replicated clients invoke operations on replicated servers and other replicated clients. The algorithm has the following desirable properties:

- Client replica group c continues to execute application logic correctly provided that the number of client replicas is at least $3f_c + 1$, where f_c is the maximum number of (possibly Byzantine) faults to be tolerated by the client replica group.
- External operation requests can be executed on the client replica group. This can be used to facilitate interaction between client replica groups as well as to perform fault tolerant atomic software upgrades of long-running applications. [9]
- The algorithm uses a designated responder in the server replica group to reduce the number of reply messages sent from the server to the client.
- Faults are isolated between replica groups with respect to correctness. Servers continue to execute correctly even if a client group becomes compromised with more than f_c client replicas becoming faulty. Similarly, even if server groups become compromised, non-faulty clients continue to agree on their execution and make progress to the extent possible without interaction with compromised servers.
- Faults are isolated with respect to performance (with modifications to CLBFT stated in [10]). Even if the bounds on the number of faulty client nodes are exceeded, compromised client groups cannot force a view change at a server. Similarly, compromised server groups cannot force a view change at clients. Furthermore, the algorithm uses an inexpensive protocol at the client replica group to select the designated responder without incurring a view change operation at either replica group.
- The algorithm supports both synchronous and asynchronous requests, masking operation latency.

Section 2 discusses our approach in the context of related work. In Section 3, we present a high-level overview of our algorithm. Section 4 describes our algorithm in more detail. Section 5 provides analysis of the time and message complexity of the algorithm. We conclude in Section 6.

2 Related Work

Our work is concerned with interaction between Byzantine fault tolerant replica groups. Prior work addresses aspects of the problem but does not provide a complete practical solution to the problem of replicated clients that access replicated servers and other replicated clients.

Thema [11] provides a server wrapper for creating replicated BFT Web Services and an external service wrapper that allows a replicated Web Service to access an external unreplicated Web Service safely. However, the current implementation does not vote on the result value of external operations and does not have a mechanism to abort an external operation request as a group. These limitations could lead to inconsistent replica state. Furthermore, Thema does

not allow a replicated Web Service to access another replicated Web Service. Hence, Thema cannot be used in systems where replica groups communicate with each other.

In the Byzantine fault-tolerant Domain Name Service (BFT-DNS) [12], each level in the BFT-DNS lookup system is replicated, with replicated clients invoking read operations on replicated servers. The primary at one level makes a call to the next level on behalf of its replicas. The replicas in the next level send replies that are collected by the lower-level primary and forwarded to its peer replicas. A compromised replica group can force a view change on the lower level by not replying to calls from the lower-level, and if the lower-level primary is also faulty, it can collude with the server group to send quorums with different results to different replicas, leading to inconsistent replica state.

Fry and Reiter [13] present a quorum replication mechanism that allows replicated objects to invoke operations on other potentially replicated objects. Client objects invoke operations on a selected quorum of server objects. Clients refer to server objects using *handles*, which can be used in calls to other objects. Certificates in the handle ensure that a quorum of handles is required to invoke an operation. However, the exponential growth in the size of the certificates with each nesting step drives up the message complexity and the cost of certificate verification.

Immune [14] supports replicated clients that invoke operations on replicated servers. It uses a totally ordered multicast mechanism based on SecureRing [15] to ensure consistently ordered one-time message delivery across replicas. This mechanism does not scale well since the number of rounds of communication is proportional to the replica group size. Our algorithm uses a constant number of rounds of communication. Therefore, latency does not increase with the size of the replica group.

3 Overview

Our algorithm supports any number of replicated clients that access any number of replicated servers or other replicated clients. However, for ease of exposition, we describe the algorithm in terms of a single replicated server s, composed of $m = 3f_s + 1$ replicas $s_1, \ldots, s_m$, and a single replicated client c composed of $n = 3f_c + 1$ replicas $c_1, \ldots, c_n$, where f_s and f_c are specified upper bounds on the number of faulty replicas to be tolerated at the server and client, respectively. All assumptions made in the system model of CLBFT [1] apply here, including standard cryptographic assumptions and a weak synchrony assumption that message delays do not grow faster than time.

We use a modular architecture to manage the complexity of the algorithm and simplify reasoning about its correctness. On the server side, our algorithm "wraps" CLBFT replicas, making it appear to them as if clients are not replicated. Each client replica c_i is composed of an *active client* replica and a *client store* (or simply *store*) replica. The active client has an *oracle*, a black box that captures the logic of the application. The store acts as a passive data server that carries out agreement operations on behalf of the active client. Each active client replica and its corresponding store replica reside on the same host.

The oracle models a deterministic application that requests operations on external servers (including the client stores of other clients) and processes their replies. The requests may be synchronous or asynchronous. Since the oracle is deterministic, we are guaranteed that if two non-faulty oracles have experienced the same sequence of requests and replies, then the next request issued by both oracles will be the same. The reply to an operation request may arrive from the server at different times at different client replica, so a mechanism is needed to ensure that replies are consumed in the same order by all client replicas. Furthermore, since some replicas may be faulty and the server may be compromised, it is necessary for the client replicas to agree on the reply value. Both of these needs are met by the client store replica group, which agrees on the reply to each server operation and places the result in a FIFO queue (in the execution order at the store) until it is consumed by the oracle in a blocking operation. Since all oracles issue the same request sequence (including requests to dequeue results from the FIFO queue), all oracles of non-faulty replicas receive the same results at the same points in their executions.

The algorithm is designed to minimize the number of messages between clients and servers. Consequently, we wish to avoid a quadratic reply multicast from server replicas to client replicas. To this end, the client group names a server replica as the *designated responder* (or simply, *responder*) for each request. The responder forwards the reply from the server to all the client replicas. The responder need not be the server primary, and therefore a client group that deems the responder unsatisfactory can select a new responder without forcing a view change at the server.

As an overview of the algorithm, we trace the execution of a request issued by the application. We begin with an overview of normal operation. Section 4 has a detailed discussion, including mechanisms for fault handling.

3.1 Normal Operation

As seen in Figure 1, when the application executing on an active client replica requests an operation on a remote server, a request (with the identity of the responder for that server) is sent to the primary of the server group. The server primary waits for at least $f_c + 1$ matching requests and then starts the CLBFT protocol to execute the operation. Instead of multicasting replies back to the $3f_c + 1$ client replicas, the server replicas forward their replies to the responder. The responder waits for at least $f_s + 1$ matching replies and then sends to each of the active client replicas a single reply message and a bundle of $f_s + 1$ reply digest signatures as proof that the server group agreed on the reply.

When an active client replica receives a reply, it verifies the validity of the signature bundle and forwards the result to the client store, which uses CLBFT to agree upon the reply. In the execute step of the CLBFT algorithm at the store, each replica places its reply in a FIFO *result queue*

for use by its corresponding active client replica. When the application deterministically decides to check for a reply to a previous request, it reads the first item from the result queue, blocking if necessary until a result is available.

Since they reside on the same host, an active client replica and its corresponding store replica fail together. Therefore, the store does not multicast reply values back to the client replicas. Instead, each active client replica trusts the reply in the result queue that was provided by its corresponding store replica.

3.2 External Updates to Client State

Because we envision this algorithm being applied to critical long-running distributed applications, it is important to accommodate online upgrades to application logic on clients as well as servers. These are accomplished as external operations on the client store, whose results become available to all oracles at the same point in their execution. Thus, behavior remains consistent across all non-faulty replicas. The same mechanism can be used to inform applications of other events to which they may subscribe.

3.3 Interaction Between Client Replica Groups

Our algorithm reuses the server wrapper at the client store. Consequently, any given replica group can act as both a client and a server with some modifications.

External operation requests to a client store can be used as a means of making requests on the client group as if it were a server. Each store replica agrees on the operation and forwards it to the active replica using the FIFO result queue. Once the application executes the operation, the active replica forms the result value and sends it back to the store replica. The store replica constructs a reply to the original request using the result value and sends it to the "client" that issued the operation. This model can be used to support both work-flow packet migration (essentially remote method invocation without return values) and nested remote method invocations in which the intermediate active client makes further invocations in order to compute the result value to the original request.

3.4 Fault Handling

In the presence of faulty replicas, normal operation may not proceed to completion. Section 4.2 discusses the mechanism used to complete operation requests if normal operation fails. We also present strategies for changing the server primary and the responder if they are suspected of being faulty or slow. Finally, we also address the issue of fault isolation if either the client or server group becomes compromised.

4 Detailed Algorithm Description

This section describes the algorithm in detail starting in Section 4.1 with the operation of the protocol in the absence of faults. Section 4.2 discusses fault handling. A complete algorithm specification using the I/O automaton model [16], key lemmas that capture the algorithm, and a proof sketch are presented in [17].

The system consists of a client group c, composed of $n = 3f_c + 1$ replicas $c_1, \ldots, c_n$, and a server-group s composed of $m = 3f_s + 1$ replicas $s_1, \ldots, s_m$. Each server replica s_i is modeled as the composition of a *server back end* (SBE_i) component that encapsulates the CLBFT protocol and a *server front end* (SFE_i) component that implements our server protocol and wraps SBE_i. Each client replica c_j is composed of an active client replica and a store replica. The active client replica is composed of a component that models the application and encapsulates the oracle (APP_j) and a *client front end* (CFE_j) component that implements our client protocol. We assume that all non-faulty client replicas begin execution with the same initial application state. A store replica is simply a composition of a SFE_j and a SBE_j component. We refer to a replica and its identifier interchangeably, and similarly for groups and group identifiers. We assume a reliable asynchronous communication channel.

4.1 Normal Operation

We consider the execution of a request from a non-faulty client group c to a non-faulty server group s.

4.1.1 Client Send Request

After APP_i at client replica i issues a request with timestamp t for operation o to be executed on server group g, CFE_i creates a CLBFT request $r = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$ where c is the replica group of i. In addition, a *request bundle* $b = \langle d, \rho, i \rangle_{\sigma_i}$ is created, where d is the digest of the CLBFT request r and ρ is CFE_i's current *designated responder* from server group g. CFE_i combines the CLBFT request and the request bundle as an *extended request* $\langle r, b \rangle$ and sends it to primary k of g. CFE_i also starts an operation timer with the tuple $\langle g, r, b \rangle$ and adds the tuple to set *requests-current*$_i$, which is used later to verify the correctness of replies from the server.

4.1.2 Server Execution

When $f_c + 1$ matching requests (with matching CLBFT bundles from distinct replicas naming the same designated responder) have been received at SFE_k, their bundles are added to a set *requests-current*$_k$, and the CLBFT request is added to the set *sbe-buffer*$_k$ of messages to be delivered to the local back-end SBE_k. Once the CLBFT request is received by SBE_k, the CLBFT algorithm starts and eventually sends a pre-prepare message p to the other replicas in the server group. The server wrapper intercepts message p containing the CLBFT request and creates an *extended pre-prepare* $\langle p, S \rangle$ with S the set of $f_c + 1$ signed bundles saved in *requests-current*$_k$ for m. These bundles serve as proof to other server replicas that at least one non-faulty client replica sent the request. This prevents faulty client replicas from colluding with a faulty server primary to convince the server to execute incorrect requests.

When SFE_j at server replica j receives the extended pre-prepare, the signed bundles in S are verified, and the bundles are saved in *requests-current*$_j$ just as at the primary. SFE_j then forward the CLBFT pre-prepare message

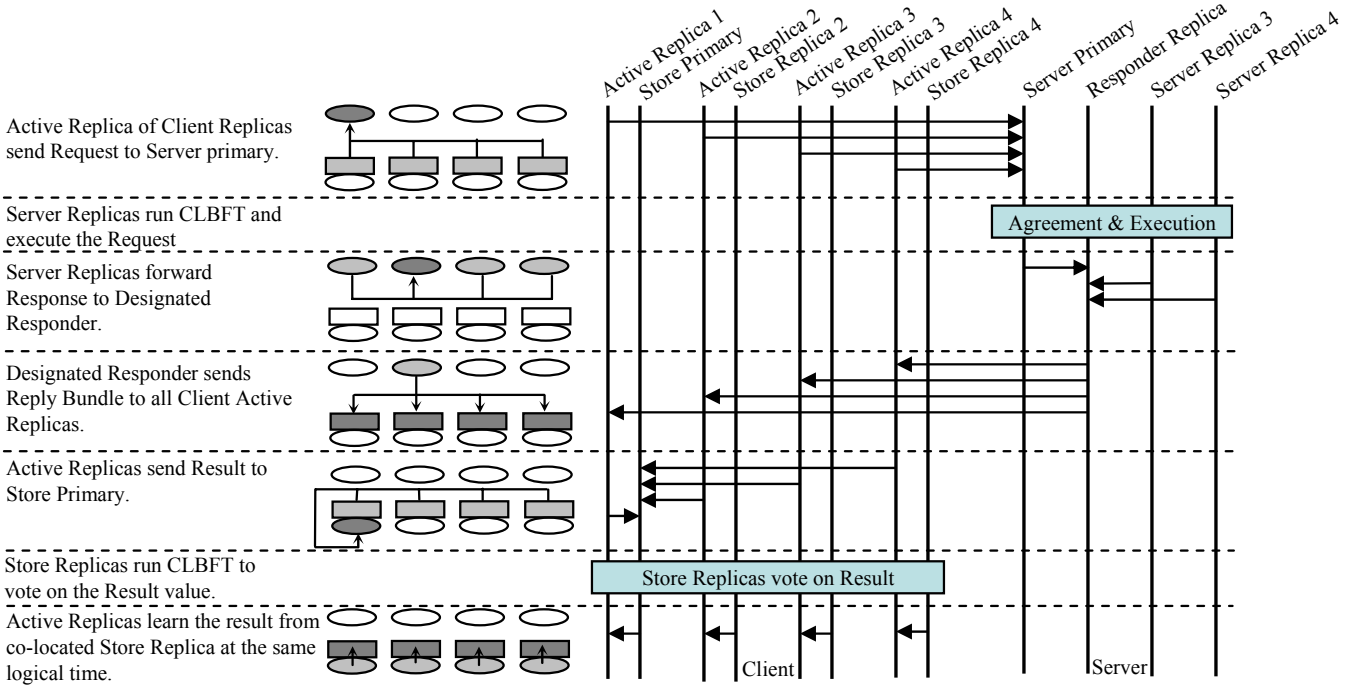

Figure 1. Normal (Non-Faulty) Operation

to SBE_j. In normal operation, the CLBFT protocol proceeds unmodified from this point through the prepare and commit stages. After SBE_j executes the operation, it generates a CLBFT reply message to be sent to the client.

SFE_j forwards the reply m from SBE_j to the designated responder (instead of the client, as in CLBFT) as an *extended reply* $\langle m, \langle j, \sigma_j \rangle \rangle$, consisting of the CLBFT reply along with a tuple consisting of the signature from the reply and the replica identifier j. Information saved in *requests-current*$_j$ is used to determine the responder.

The replies are received by SFE_ρ at responder ρ and added to the set *replies-incoming*$_\rho$. When this set contains $f_s + 1$ matching extended replies (two extended replies *match* when the CLBFT replies in each are identical, the signatures for each are correct, and the signers are distinct) for the request for client group c with timestamp t with result r, an *extended reply forward* consisting of the original CLBFT reply and the corresponding set of $f_s + 1$ signature–identifier tuples is created and sent to each of the client replicas. This set serves as proof of correct execution by the server group to each of the client replicas. The n extended reply forwards sent by the designated responder avoid a costly quadratic broadcast.

4.1.3 Client Receive Reply

After the responder sends an extended reply forward $m = \langle r', S \rangle$ to a client replica i, that replica eventually receives the reply, verifies the signatures, and adds the reply–signatures tuples to the set *replies-incoming*$_i$. When at least $f_s + 1$ matching tuples are present in set *replies-incoming*$_i$, which is immediate if the responder is non-faulty, and a tuple $m = \langle g, r, b \rangle$ that corresponds to the reply is in *requests-current*$_i$, the client replica removes m from the *requests-current*$_i$, and constructs an extended request r_μ (using the same timestamp as the reply) that contains an APPLY-RESULT operation for the client store to apply the result. This extended request is then sent to the client store primary k_μ as a regular server operation request.

This sequence of actions takes place at each non-faulty client replica, so eventually at least $f_c + 1$ matching extended requests for an APPLY-RESULT operation are sent to the store primary. Assuming that the client group, and hence client store (i.e., the composition of server front-end and back-end automata) is not compromised, we expect to receive a reply from the store that contains an agreed upon result for the request. This result could either be the one proposed by client replica i or, in the case of a faulty server, an ABORT message, as described in Section 4.2.3. CFE_i receives the store reply r'_μ directly from the co-located store replica instead of via the channel. We make a simple modification to CLBFT to send replies in execution order. As the replies are delivered to the client front-end, they are appended to the end of the queue *store-replies*$_i$, and the corresponding APPLY-RESULT request is removed from *requests-current*$_i$.

The component APP_i schedules requests until it requires the result of some request to make further requests. When it becomes blocked, it signals CFE_i. CFE_i dequeues the head of the reply queue and sends it to APP_i. At this point, the application updates its state using the new result and can schedule further requests.

4.2 Fault Handling

Each client replica starts a timer upon sending a request to a server primary. If the timer expires before a reply is

Algorithm stage	Latency (rounds)	Message count	Total message size
1. Send request to server	1	$O(n)$	$O(\ell n)$
2. Server agreement (CLBFT)	4 (3 for reads)	$O(m^2)$	$O(m^2 + mn + \ell m)$
3. Responder to client group	1	$O(n)$	$O(mn + \ell n)$
4. Forward reply to client store	1	$O(n)$	$O(\ell n)$
5. Client store agreement (CLBFT)	3 (local reply)	$O(n^2)$	$O(n^2 + \ell n)$
Total	10 (9 for reads)	$O(m^2 + n^2)$	$O(m^2 + n^2 + mn + \ell m + \ell n)$

Table 1. Message Complexity

received, there are three possible scenarios: (1) the server primary is faulty and has discarded the client requests, (2) the designated responder is faulty and did not send the response to some or all of the active client replicas, or (3) the timeout value is too low for current network conditions.

When a client replica times out waiting for a reply, it resends the request to all m server replicas. A server replica waits for at least $f_c + 1$ matching requests (to prevent a faulty client replica from forcing a multicast) and then determines whether agreement on the requested operation has been started by the primary. If not, the server replica forwards the request bundle (including the $f_c + 1$ matching requests) to the server primary. It also starts a view-change timer as defined in CLBFT. If the replica has executed (or eventually executes) the operation under the current primary, it multicasts the extended reply message to all n client replicas.

If the number of faulty server replicas does not exceed f_s, each active client replica eventually receives at least $f_s + 1$ correct extended reply messages with matching results. It then creates a new store request to apply the result, just as in Section 4.1.3.

4.2.1 Unresponsive Designated Responder

We wish to bound the additional traffic that could be caused by a slow or faulty responder. The designated responder for a server group need not be the server primary, and changing the responder does not require a view change at the server. At any time, a client replica may vote to change the designated responder for server group g by sending a request to the store with operation $\langle \text{RESPONDER-CHANGE}, g \rangle$. If it is ever the case that $f_c + 1$ requests to change the responder are received at the store from distinct client replicas, the store will process the request to change the responder and notify all client replicas at the same logical time.

4.2.2 Compromised Client Group

If the number of faulty client replicas exceeds f_c, we still want non-faulty server groups to function correctly. The SFE automata only trigger the CLBFT protocol when f_c+1 matching client requests arrive so the case of a faulty client group reduces to the case of a single faulty client in CLBFT, and safety is ensured. Information is piggybacked on existing CLBFT messages, but messages to or from non-faulty clients are not dropped, so view changes due to a faulty or slow primary are not hindered. One potential concern is a faulty client group sending two matching sets of extended requests that contain identical CLBFT requests but different designated responders. A faulty server primary could collude with the client group and distribute different sets of signed bundles to different server replicas. In this case, neither designated responder might receive enough extended reply forwards to achieve quorum to send the extended reply to the client group. This does not affect correctness because the client group is compromised to begin with.

4.2.3 Compromised Server Group

A compromised server group, in which the number of faulty replicas exceeds f_s, should not be able to cause diverging application state at client replicas or prevent the client application from making progress. Client replicas are individually satisfied with a reply from a server after receiving $f_s + 1$ correct signatures for a particular result, but a compromised server group could send different results to different replicas. As a result, the client replicas may send APPLY-RESULT requests for different result values to the store. The store will process the requests only if at least $f_c + 1$ client replicas request to apply the same result value. The timestamp used to send the original request to the server is reused for the APPLY-RESULT request. It is possible for the client store to form quorums for the same timestamp and two separate result values. In this case, the store primary will assign a sequence number to each request and process it, but as defined by CLBFT, only one apply result operation will execute at any of the store replicas, ensuring consistency.

In the CLBFT protocol, an unreplicated client is free to time out when waiting for a response, handle the exception, and ignore any future reply to that request. In our case, the client application is replicated. Since we do not assume even loosely synchronized clocks, replicas may time out at different times. We cannot allow client replicas to stop waiting for a reply solely on the basis of their local timers, because a delayed reply may arrive after only some replicas have timed out, causing the behavior of non-faulty active client replicas to diverge. We handle the problem of faulty server groups as follows.

After issuing a request r to a server, a client replica may time out (or otherwise suspect that the server is faulty) and wish to stop waiting for a reply to r. Then, the replica requests that a "suspect faulty" operation be performed on the client store. If $2f_c + 1$ active client replicas issue such requests, and if the client store has not yet processed a reply for r, then the client store will agree to suspect the server as faulty and inform the active replicas by placing a reply to the "suspect faulty" operation in the result queue. All

replicas will consume this result at the same point in their execution, handle the missing reply as dictated by the application, and continue to exhibit identical behavior. While all other store operations can be processed with $f_c + 1$ matching requests, "suspect faulty" operations require $2f_c + 1$ to ensure that at least $f_c + 1$ non-faulty replicas had sent r to the server giving the server a chance to accept the request and execute the operation.

5 Complexity Analysis

We analyze the latency, message count, and total message size during normal operation in terms of the client replica group size $n = 3f_c + 1$ and server replica group size $m = 3f_s + 1$. Let ℓ be the maximum length of operation requests and replies. We build on CLBFT, which supports only unreplicated clients and incurs 4 message delays (3 for reads), $O(m^2)$ messages and $O(m^2 + \ell m)$ total message size. Table 1 shows the analysis for each stage of our algorithm during normal operation. We assume message digests and digital signatures are of constant length.

Taking constants into account, our algorithm incurs approximately twice the number of messages and total message size as CLBFT. If we assume the replica groups are the same size and constant length requests and replies, we have a latency of $O(1)$, message count of $O(n^2)$, and total message size of $O(n^2)$.

The checkpoint and recovery mechanism of the CLBFT algorithm can be leveraged to preserve the additional state on both client and server sides with no additional overhead. By synchronizing on the consumption of the result queue to a particular store operation sequence number, we can ensure that each checkpoint performed by the store contains consistent state information for all replicas, including the state of the active client replica.

Several optimizations for reducing message complexity as well as a description of how to linearly bound the space required at each replica for non-faulty CFE and SFE replicas are presented elsewhere [17].

6 Conclusion

We have presented a practical algorithm for Byzantine fault-tolerant execution of long-running autonomous distributed applications. We support interaction between both passive and active replica groups while providing a high degree of fault isolation between replica groups. Our algorithm can be used to build tiered distributed systems where each tier is replicated. Such tiered systems support fault tolerant distributed applications that use either workflow packet migration or nested remote method invocation models. We also support online upgrades to replica groups by leveraging the basic operation invocation mechanism.

We are currently implementing a prototype of the algorithm and we plan to investigate the scalability of our algorithm both in terms of replica group size and the number of replica groups that interact with each other.

References

[1] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, pages 173–186, 1999.

[2] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proc. 5th Intl. Conf. on Dependable Systems and Networks*, pages 135–144, 2004.

[3] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Proc. 29th Symp. on Theory of Computing*, pages 569–578, 1997.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *ACM Operating Systems Review*, 36(SI):1–14, 2002.

[5] J. P. Martin, L. Alvisi, and M. Dahlin. Minimal Byzantine Storage. In *Proc. 16th Intl. Conf. on Distributed Computing*, pages 311–325, 2002.

[6] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[7] B. Prenel and P. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Proc. 15th Conf. on Advances in Cryptology*, pages 1–14, 1995.

[8] R. Rivest. RFC 1321: The MD5 Message-Digest Algorithm, 1992.

[9] H. D. Thorvaldsson and K. J. Goldman. Dynamic Evolution in a Survivable Application Infrastructure. In *Proc. 18th IASTED Intl. Conf. on Paralel and Distributed Computing and Systems*, 2006.

[10] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman. Preserving Performance of Byzantine Fault Tolerant Replica Groups in the Presence of Malicious Clients. Technical Report WUCSE-2006-52, Washington University, 2006.

[11] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware forWeb-Service Applications. In *Proc. 24th Symp. on Reliable Distributed Systems*, pages 131–140, 2005.

[12] S. Ahmed. A Scalable Byzantine Fault Tolerant Secure Domain Name System, 2001. Master's thesis, Massachusetts Institute of Technology.

[13] C. Fry and M. Reiter. Nested Objects in a Byzantine Quorum-Replicated System. In *Proc. 23rd Intl. Symp. on Reliable Distributed Systems*, pages 79–89, 2004.

[14] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing Support for Survivable CORBA Applications with the Immune System. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, pages 507–516, 1999.

[15] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, 2001.

[16] N. Lynch and M. Tuttle. Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[17] I. Wehrman, S. L. Pallemulle, and K. J. Goldman. Extending Byzantine Fault Tolerance to Replicated Clients. Technical Report WUCSE-2006-7, Washington University, 2006.