WUCSE-2006-7

Extending Byzantine Fault Tolerance to Replicated Clients

Authors: Wehrman, Ian; Pallemulle, Sajeeva L.; Goldman, Kenneth J.

February 7, 2006

Corresponding Author: kjg@cse.wustl.edu

Extending Byzantine Fault Tolerance to Replicated Clients

Ian Wehrman
iwehrman@cse.wustl.edu

Sajeeva L. Pallemulle
sajeeva@cse.wustl.edu

Kenneth J. Goldman
kjg@cse.wustl.edu

Computer Science and Engineering
Washington University in St. Louis

Abstract

Byzantine agreement protocols for replicated deterministic state machines guarantee that externally requested operations continue to execute correctly even if a bounded number of replicas fail in arbitrary ways. The state machines are passive, with clients responsible for any active ongoing application behavior. However, the clients are unreplicated and outside the fault-tolerance boundary. Consequently, agreement protocols for replicated state machines do not guarantee continued correct execution of long-running client applications.

Building on the Castro and Liskov Byzantine Fault Tolerance protocol for unreplicated clients (CLBFT), we present a practical algorithm for Byzantine fault-tolerant execution of long-running distributed applications in which replicated deterministic clients invoke operations on replicated deterministic servers. The algorithm scales well to large replica groups, with roughly double the latency and message count when compared to CLBFT, which supports only unreplicated clients. The algorithm supports both synchronous and asynchronous clients, provides fault isolation between client and server groups with respect to both correctness and performance, and uses a novel architecture that accommodates externally requested software upgrades for long-running evolvable client applications.

Keywords: distributed algorithms, Byzantine agreement, fault-tolerance, replication, replicated clients

1 Introduction

This paper addresses the problem of Byzantine fault-tolerant (BFT) execution of long-running distributed applications. Recent advances in distributed systems research have resulted in practical algorithms for constructing replicated *data servers* that continue to operate in the face of crashes, intermittent communication failures, and malicious (Byzantine) attacks in which processes may collude or lie [7, 9]. These algorithms support distributed applications in which client applications invoke operations on remote data servers with the guarantee that the servers continue to respond correctly even if a bounded number of their replicas are faulty or compromised. The fault-tolerant data servers are passive deterministic state machines that execute operations only in response to external clients. Since the active portions of applications run outside the system, on client hosts, the system cannot guarantee that applications continue to run.

This paper presents an algorithm for Byzantine fault-tolerant execution of replicated clients that access remote data servers. To our knowledge, this is the first practical algorithm that supports Byzantine fault tolerance for replicated clients that access replicated servers. The algorithm scales well, provides fault isolation between the client and the server, and has a modular design. We envision this algorithm being used in a shared infrastructure that supports the installation, evolution, and fault-tolerant execution of long-running distributed applications [17].

We begin, in Section 2, with a brief description of the Castro and Liskov Byzantine Fault Tolerance protocol for replicated deterministic state machines (CLBFT) [9], on which our protocol is based. Section 3 summarizes the contributions of the paper, including the key properties of our algorithm, and Section 4 discusses those properties in the context of related work. Section 5 presents a high-level overview of our algorithm. Section 6 provides a brief review of the I/O automaton model [10], which is used in Section 7 to describe and reason about the correctness of our algorithm. Section 8 provides analysis of the time and message complexity of the algorithm. Section 9 discusses optimizations, recovery, and garbage collection. We conclude in Section 10 with a discussion of future work.

2 The CLBFT Protocol

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 [5, 14] are used to verify authenticity of messages, and message digests [15] are used to reduce message size. The CLBFT algorithm for a mutating operation works roughly as follows. (Accessors require less communication.) A client sends its request to a designated replica, called the *primary*, 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 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 proceeded to the commit stage. 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 switch to another primary in a *view change* operation. Since view changes are expensive, progress timers adapt to prevent view changes from happening too often.

3 Contributions

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

- Arbitrary long-running deterministic client applications continue to execute correctly provided that the number of client replicas n is at least $3f_c + 1$, where f_c is the maximum number of (possibly Byzantine) faults to be tolerated in the client replica group.

- The algorithm uses CLBFT to guarantee that servers continue to execute operations correctly provided that the number of server replicas m is at least $3f_s + 1$, where f_s is the maximum number of (possibly Byzantine) faults to be tolerated by the server replica group.

- The algorithm uses a designated responder in the server replica group to reduce the number of reply messages sent from the server group to the client group.

- Faults are isolated between the client and server with respect to correctness. Servers continue to execute correctly even if a client as a whole becomes faulty because the number of faulty client replicas has exceeded f_c. Similarly, non-faulty replicated clients continue to agree on their execution and make progress even if more than f_s server replicas are faulty.

- Faults are isolated with respect to performance. Even if the bounds on the number of faulty client nodes are exceeded, faulty client replicas cannot force a view change at the server. Similarly, faulty server nodes cannot force a view change at the client. Furthermore, the algorithm uses an inexpensive protocol at the client replica group to select the designated responder, which is not necessarily the server primary. This allows clients to select a new designated responder without incurring a view change operation at either replica group.

- External operation requests can be executed on the client replica group, for example to perform logically simultaneous software upgrades of long-running client applications.
- To support replicated clients, our algorithm incurs a modest amount of overhead when compared to CLBFT for unreplicated clients: approximately twice the operation latency and number of messages during normal operation. Clients groups of size one do not incur this overhead.
- The algorithm supports both synchronous and asynchronous clients, masking operation latency.

This is a difficult problem that does not lend itself to a simple solution. We use a modular architecture which helps manage the complexity of the algorithm and simplifies reasoning about its correctness. A key feature of this architecture is the use of CLBFT as a subroutine at both client and server. Our algorithm "wraps" the CLBFT replicas, making it appear to them as if their clients are not replicated. Each client replica is a twin, participating in an active group that encapsulates the behavior of the running application, and a passive store that both accepts external requests for application upgrades and manages decisions that are triggered asynchronously by the active portion. We reuse the server wrapper at the client store.

We use I/O automata [10] to describe and reason about our algorithm. We leverage the existing I/O automaton description and proof for CLBFT [2] and use a proof technique [18] for modular reasoning about distributed algorithms in terms of distributed subroutines. For performance, we piggyback information on existing CLBFT messages, so we cannot treat CLBFT entirely as a black box. Instead, modularity is achieved by reasoning about CLBFT in terms of well-formed sequences that characterize its behavior.

4 Related Work

Our work is concerned with Byzantine fault tolerance for replicated clients. Prior algorithms address aspects of the problem, but none provide a complete practical solution to the problem of active replicated clients that access passive replicated data servers.

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, Thema does not provide a mechanism for a replicated Web Service to access another replicated Web Service.

In the Byzantine fault-tolerant Domain Name Service (BFT-DNS) [1], each level in the BFT-DNS lookup system is replicated, with replicated BFT clients invoking read operations on replicated BFT 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 which are collected by the lower-level primary and forwarded to its peer replicas. A faulty replica group can force a view change on the lower level by not replying to calls from the lower-level primary and if the lower-level primary is also faulty it can collude with the server group to send quorums with different result values to different replicas thereby producing replica inconsistency.

Fry and Reiter [6] presents a mechanism that allows replicated objects to invoke other potentially replicated objects. They use quorum-based techniques instead of state machine replication. Client objects invoke operations on a selected quorum of server objects. Clients refer to server objects using handles, which can be passed in a call to another object. Certificates in the handle ensure that a quorum of handles is required to invoke an operation. However, the dependance on client invocations to exchange state information on the server side, as well as the exponential growth in the size of the certificates with each nesting step, drive up the message complexity and the cost of certificate verification.

Immune [12] supports replicated clients that invoke operations on replicated servers. It uses a secure, reliable, totally ordered multicast mechanism based on SecureRing [8] to ensure consistently ordered one-time message delivery across replicas. This mechanism does not scale well to large replica groups since the number of rounds of communication is proportional to the replica group size. Our algorithm uses only a constant number of rounds of communication, so latency does not increase with the size of the replica group.

5 Overview

Our algorithm supports any number of client replica groups that access any number of shared replicated servers. However, for ease of exposition, we describe the algorithm in terms of a single replicated client c that invokes operations on a single replicated server s. We assume there are exactly $m = 3f_s + 1$ replicas for s, named $s_1 \ldots s_m$, and $n = 3f_c + 1$ replicas for c, named $c_1 \ldots c_n$, 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 [3] apply here, including the standard cryptographic assumptions and a weak synchrony assumption that message delays do not grow faster than real time.

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 ongoing 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 and processes their replies. The application may be synchronous (wait for the result of each request before issuing the next request) or asynchronous (issue multiple operation requests before using their results). Since the oracle is deterministic, we are guaranteed that if two non-faulty oracle replicas have experienced the same sequence of requests and replies, then the next requests issued at both oracles will be the same. The reply to an operation request may arrive from the server at different times at different client replicas, so a mechanism is required to order the replies identically at each replica. Furthermore, since some nodes may be faulty and the server may misbehave, 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), all non-faulty oracles receive the same results at the same points in their executions.

The algorithm is designed to minimize the number of messages between the client and server. Consequently, we wish to avoid having all of the server replicas multicast replies to all of the client replicas. To this end, the client group names a particular server replica as the *designated responder* (or simply, *responder*) for each request. The responder is responsible for forwarding the reply from the server to all the client replicas. The responder need not be the server primary, so 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 normal operation, and then describe fault handling. Details are discussed in Section 7.

5.1 Normal Operation

When an application requests an operation on a remote server, its active client replica forwards the request (with the identity of the current designated responder) 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 at the server by providing it with the (single) client request. The CLBFT protocol runs as a subroutine to execute the operation. Then, rather than multicasting replies back to the $3f_c + 1$ client replicas, the server replicas instead forward their replies to the designated 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 that includes 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 reply to the client store, which uses CLBFT to agree upon the reply. Agreement on the reply is necessary for fault isolation because a faulty server group could, for a given request, send different client replicas “valid” replies containing different return values. Accepting those replies without agreement could cause the behavior of non-faulty client applications to diverge. 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 application 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, we do 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.

5.2 Fault Handling

Each client replica starts a timer upon sending a request to a server primary. If the timer expires before a reply is received, then there are three possible scenarios. (1) The server primary is faulty and 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. If a server replica receives at least f_c+1 matching requests, the replica determines if 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 reply 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 $f_s + 1$ matching replies, and the reply is then processed normally as described above.

5.3 Changing the Designated Responder

We wish to bound the additional traffic that could be caused by a slow or faulty designated responder. A client replica can decide, at any time, that it is receiving poor service from the designated responder replica s_i. When it makes this decision, it invokes an operation on the client store replica group to request a designated responder change. When $f_c + 1$ client replicas request a designated responder change, the client store processes the request like any other, and places the reply in the result queue. When the reply comes to the front of the queue, the active client replica consumes it and invokes future requests with the next designated responder.

5.4 Faulty Server Groups

The number of faulty server replicas may exceed f_s, in which case the server as a whole may exhibit faulty behavior. A faulty server group may not provide a coherent response to a request, preventing the client store from agreeing on the result. However, we cannot allow this to block the client application.

In the CLBFT protocol, an unreplicated client application is free to time out 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 application 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. When this happens, the replica requests that a "suspect faulty" operation be performed on the client store. If $2f_c + 1$ application 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 application 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. Note that while all other store operations can be processed with $f_c + 1$ matching requests, "suspect faulty" operations require $2f_c + 1$. This ensures that at least $f_c + 1$ non-faulty replicas had sent r to the server, enough for the server to accept the request and have a chance to execute the operation.

5.5 External Updates to Client State

Because we envision this algorithm being applied to critical long-running distributed applications, it is important to accommodate upgrades to client programs without quiescing the system. These are accomplished as external operations on the client store, whose results become available to all application replicas 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.

6 I/O Automata

We specify and reason about our algorithm using the I/O automaton model [10]. An I/O automaton a is an (infinite) state machine, whose state transitions are called *actions*, denoted $acts(a)$. These are partitioned into *input* actions $in(a)$, *output* actions $out(a)$ and *internal* actions $int(a)$. Input and output actions are called *external* actions. Output and internal actions are called *local* actions and have *preconditions* defining the sets of states in which they are *enabled.* Input actions are always enabled. An *execution* α is an alternating sequence of states s and actions π such that each consecutive triple (s, π, s') is in the transition relation. The occurrence of an action is called an *event.* The *trace* of α, denoted $trace(\alpha)$, is the subsequence consisting of all events for external actions. In this paper, we say that an execution α is *fair* if local actions are given chances to execute infinitely often: if a local action remains enabled, eventually it will occur.

Automata $a_1 \ldots a_k$ may be *composed* to form an automaton a, provided each action is a local action of at most one automaton a_i. The states of a are the Cartesian product of the states of the component automata. The actions of a are $out(a) = \bigcup_{i \in I} out(a_i)$, $int(a) = \bigcup_{i \in I} int(a_i)$ and $in(a) = \bigcup_{i \in I} in(a_i) - \bigcup_{i \in I} out(a_i)$. Component automata may *share* an action (e.g., as an output of one and an input of another). In the composition, the component automata change state simultaneously, according to their individual transition relations, when a shared action occurs. For Σ a set of actions, $hide_{\Sigma}(a)$ is an automaton identical to a except that all actions in Σ are internal.

Given a trace α and a set of actions Σ, the *projection* of α on Σ, denoted $\alpha|\Sigma$, is the subsequence of α consisting of all events for actions in Σ. For automaton a, we use $\alpha|a$ as a shorthand for $\alpha|acts(a)$. Let a be a composition of automata including a_i. If s is a state of a, the projection $s|a_i$ is the state of a_i in s. The *projection* $\alpha|a_i$ of execution α of a is the subsequence of α containing events in $acts(a_i)$ and their adjacent states s projected on a_i. It can be shown that traces (executions) of a composition yield traces (executions) of the component automata when projected on those components.

7 The Algorithm

This section describes the algorithm in sufficient detail to reason more formally about its correctness. Section 7.2 describes the actions that occur during normal execution of the protocol, and Section 7.3 discusses actions for fault handling. In Section 7.4 reasons about the correctness of the algorithm as a whole. *A complete specification of the algorithm using the I/O automaton model is included the appendix.*

The system consists of a non-faulty client group c, composed of $n = 3f_c + 1$ replicas $c_1, \ldots, c_n$, and a non-faulty server-group s composed of $m = 3f_s + 1$ replicas $s_1, \ldots, s_m$. We model each server replica s_i as the composition of a *server back end* SBE_i automaton that encapsulates the CLBFT protocol and a *server front end* SFE_i automaton that implements our server protocol and wraps SBE_i. CLBFT actions are renamed so that to SBE_i, it appears as if SFE_i is the network. Each client replica c_j is composed of an active client replica and a store replica. The active client replica is composed of automaton APP_j, which models the application and encapsulates the oracle, and a *client front end* CFE_j automaton that implements our client protocol. The store replica, like any other server, is composed of an SFE_j and a SBE_j automaton. The CFE and SFE automata communicate over a *message channel* MC, which models the network. The system S is the composition of these automata with MC. We take the liberty of using a replica and its identifier interchangeably, and similarly for groups and group identifiers.

7.1 Communication Model

The automaton MC is an asynchronous multicast message channel. A message m is *sent* by replica i to replicas in set X with action $\text{FILTER-TO-CHANNEL}(m, X)_i$, and m is *delivered* to replica j (or is *received* by j) with action $\textsc{channel-to-filter}(m)_j$. The message channel defines the former as an input, and the latter as an output action; the SFE and CFE automata define the former as an output and latter as an input action. We define the message channel so that, in a fair execution of S, a message is eventually delivered to all non-faulty recipients intended by the sender (and possibly others). We say a message m is *in transit* to replica i if either, for some X with $i \in X$, if it is waiting to be delivered or has been received by replica i with action $\textsc{channel-to-filter}(m)_i$.

7.2 Normal Operation

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

7.2.1 Client Send Request

The application $\textsc{app}_i$ at each client replica i schedules requests to be executed with the output action $\textsc{app-to-cfe}(\langle \textsc{schedule}, t, g, o\rangle)_i$ for some timestamp t, server group g and operation o. The same input action at $\textsc{cfe}_i$ uses this information to create a CLBFT request $r = \langle \textsc{request}, o, t, c\rangle_{\sigma_c}$, where c is the replica group of i and σ_c the signature for the request using the client group's shared private key. In addition, a *request bundle* (or just *bundle* when context is clear) $b = \langle d, \rho, i\rangle_{\sigma_i}$ is created, where d is the digest of the CLBFT request r and ρ is the *designated responder* for the server group g. The designated responder is calculated deterministically using the set *responder-seq*$_i$ by the client replica as the remote replica that will send replies on behalf of all replicas in the server group. (The operation of the responder is discussed in more detail below in Section 7.2.2.) We define the pair $\langle r, b\rangle$ of CLBFT request and request bundle as an *extended request*. Finally, $\textsc{app-to-cfe}(\langle \textsc{schedule}, t, g, o\rangle)_i$ adds tuple $\langle g, r, b\rangle$ to *requests-new*$_i$, the set of messages to be sent out. Another internal action $\textsc{process-server-request}(g, r, b)_i$ adds the extended request to the set *channel-buffer*$_i$ of outgoing channel messages to be delivered to primary k of g. It also adds the tuple $\langle g, r, b\rangle$ to the the set *timer-buffer*$_i$ of outgoing timer messages, and the set *requests-current*$_i$, used later to verify the correctness of replies from the server. This causes the output action $\textsc{filter-to-channel}(\langle r, b\rangle, \{k\})_i$ to be enabled, which sends the message on the channel to k, and the action $\textsc{cfe-to-timer}(g, r, b)_i$ to be enabled, which starts the timer for this request when executed. The following lemma asserts the correctness of this process.

Lemma 1 (Application $\textsc{schedule}$ to Extended Request). *Let $\beta = \alpha\alpha'$ be a fair execution and $\textsc{app-to-cfe}(\langle \textsc{schedule}, t, o, g\rangle)_i$ be the final event of α, for some timestamp t, operation o, group g and non-faulty replica i in group c. Then α' contains the event $\textsc{filter-to-channel}(\langle\langle \textsc{request}, o, t, c\rangle_{\sigma_c}, \langle \mathsf{digest}(m), \rho, i\rangle_{\sigma_i}\rangle, \{k\})$ with k the primary of group g and ρ the responder for g.*

A set of extended requests *match* if it contains identical CLBFT requests with bundles correctly signed by distinct replicas and that all name the same designated responder. We assume that each client replica begins execution with identical application state, and since the transition is deterministic, the sequence of operations scheduled is the same at each replica. (In Section 7.2.3 we describe how consistent application states across replicas are maintained during execution.) Consequently, there is some state in which a set of $2f_c + 1$ matching extended requests for r is in transit to server primary k, one sent by each non-faulty replica in group c.

7.2.2 Server Execution

The individual requests are eventually delivered to the primary by the action $\textsc{channel-to-filter}(\langle m = \langle \textsc{request}, o, t, c\rangle_{\sigma_c}, \langle d, \rho, i\rangle_{\sigma_i}\rangle)_k$ where the request and bundle are added to the set *requests-incoming*$_k$. When $f_c + 1$ matching requests have been received, $\textsc{request-confirm}(\langle \textsc{request}, o, t, g\rangle_{\sigma_g})_k$ becomes enabled, which adds the matching set of bundles to the set *requests-current*$_k$ and adds the CLBFT request to the set *sbe-buffer*$_k$ of messages to be delivered to the local back-end $\textsc{sbe}_i$. Delivery by action $\textsc{sfe-to-sbe}(m)_k$ starts the CLBFT operation in execution, and we expect a non-faulty primary to follow with a pre-prepare message to the other replicas in the server group. The server wrapper intercepts this outgoing message with action $\textsc{sbe-to-sfe}(p = \langle \textsc{pre-prepare}, v, n, m\rangle, R)_k$ 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 piggybacked bundles serve as proof to other server replicas that at least one non-faulty client replica sent the request; without this, a single faulty client replica could collude with a faulty primary to convince the server to execute incorrect requests.

Lemma 2 (Extended Requests to Extended Pre-prepares). *Let $\beta = \alpha\alpha'$ be a fair execution and q be the final state of α. If q is the first state in which $f_c + 1$ matching extended requests are in transit to a non-faulty server primary i, then α' contains either the event $\textsc{filter-to-channel}(p, X)_i$ where p is the corresponding extended pre-prepare and X the set of server group identifiers, or the event $\textsc{channel-to-filter}(n)_i$ with n an extended new view message.*

When wrappers at each replica receive the extended pre-prepare with action $\textsc{channel-to-filter}(\langle p, S\rangle)_i$, the authenticity of the signed bundles S is verified and, if correct, they are saved in *requests-current*$_i$ just as at the primary. The replicas then forward the CLBFT pre-prepare message into their local back end. In normal operation, the CLBFT protocol proceeds unmodified from here through the prepare and commit stages. After the individual server replicas execute the operation, they send CLBFT reply messages to the client. For later convenience, we assume replies are sent out in the order in which they are executed, a minor change that does not affect the correctness of CLBFT.

Lemma 3 (Extended Pre-prepare to Reply). *Let $\beta = \alpha\alpha'$ be a fair execution and q be the final state of α. If q is the first state in which an extended pre-prepare p is in transit to a non-faulty server replica i, then α' contains the event* $\textsc{sbe-to-sfe}(r)_i$ *with r the corresponding CLBFT reply.*

These replies are intercepted by the wrappers in action $\textsc{sbe-to-sfe}(m = \langle \textsc{reply}, v, t, g, c, r\rangle)_i$ and forwarded to the designated responder (instead of the client, as intended by CLBFT) as *extended replies* $\langle m, \{\langle i, \sigma_s\rangle\}\rangle$, consisting of the CLBFT reply along with a tuple consisting of the signature on the reply and the replica's own identifier. Information in the bundles saved in $\textit{requests-current}_i$ is used to determine the agreed-upon responder. A set of extended replies *match* when the CLBFT replies are identical, the signatures for each are correct, and the signers are distinct.

Lemma 4 (Reply to Extended Reply). *Let $\beta = \alpha\alpha'$ be a fair execution and* $\textsc{sbe-to-sfe}(r)_i$ *be the final event of α, for some reply r. If the group g that formed the request is not $\textit{twin}_i$ (a co-located* CFE*) then α' contains the event* $\textsc{filter-to-channel}(m, X)_i$ *with $m = \langle r, S\rangle$ the corresponding extended reply and either X is either $\{\textit{responder}(m)\}$ or the set of replicas in g.*

The replies are received by the responder ρ with action $\textsc{channel-to-filter}(m, S)_\rho$ and added to the set $\textit{replies-incoming}_\rho$. The precondition for action $\textsc{reply-confirm}(t, c, r)_\rho$ becomes enabled when there exists a set $f_s + 1$ matching extended replies for the request for client group c with timestamp t with result r. Upon executing, an *extended reply forward* is created to send to each of the client replicas, consisting of the original CLBFT reply and the corresponding set of $f_s + 1$ signature–identifier pairs. 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 an $m \times n$ broadcast of the reply.

Lemma 5 (Extended Replies to Extended Reply Forwards). *Let $\beta = \alpha\alpha'$ be a fair execution and q be the final state of α. If q is the first state in which $f_c + 1$ matching extended replies are in transit to a non-faulty server replica i, then α' contains the event* $\textsc{filter-to-channel}(m, X)_i$*, where m is the corresponding extended reply forward and X is the set of client replicas in the group that formed the original request.*

7.2.3 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 with input action $\textsc{channel-to-filter}(m)_i$, which verifies the validity of the signatures and adds the reply–signatures pairs to the set $\textit{replies-incoming}_i$. When at least $f_s + 1$ pairs are present in set $\textit{replies-incoming}_i$ (which is immediate in case the responder is non-faulty) and when a waiting request $m = \langle g, r, b\rangle$ is in $\textit{requests-current}_i$ that corresponds to the reply, the internal action

PROCESS-REPLIES-INCOMING$(m)_i$ becomes enabled, whose execution removes the request from the waiting set, constructs an extended request r_μ (using the same timestamp as the reply) for the client store to apply the result and adds it to the set of outgoing requests, *requests-new*$_i$. The action PROCESS-SERVER-REQUEST (that also prepared the request for the remote server) then creates an APPLY-RESULT request for the store primary k_μ, and FILTER-TO-CHANNEL$(r_\mu, \{k_\mu\})_i$ sends it to k_μ via the channel.

Lemma 6 (Extended Reply Forward to APPLY-RESULT Extended Store Request). *Let $\beta = \alpha\alpha'$ be a fair execution and q be the final state of α. If q is the first state in which an extended reply forward r is in transit to a non-faulty client replica i, then α' contains the event* FILTER-TO-CHANNEL$(r_\mu, \{k_\mu\})_i$*, where r_μ is the* APPLY-RESULT *extended request for r, with k_μ the identifier of client store primary.*

This sequence of actions takes place at each non-faulty client replica, so eventually there are at least f_c+1 matching extended requests for an APPLY-RESULT operation in transit to the store primary. Assuming that the client group, and hence client store (i.e., the composition of server front-ends and back-end automata) is non-faulty, we expect to receive a reply from the store that contains an agreed upon result for the prior 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 7.3.3. However, since the client store replicas are co-located with the client group replicas, CFE$_i$ receives the store reply $r_\mu^{'}$ directly with action SFE-TO-CFE$(r_\mu^{'})_i$ instead of via the channel. As stated above, we assume a slightly modified CLBFT implementation that sends 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 request is removed from *requests-current*$_i$.

Lemma 7 (APPLY-RESULT Extended Store Requests to Reply Queue). *Let $\beta = \alpha\alpha'$ be a fair execution and q be the final state of α. If q is the first state in which a set of $f_c + 1$ matching extended requests with operation o and tag$(o,$* APPLY-RESULT*) are in transit to a non-faulty client store primary k_μ, then α' contains the event* SFE-TO-CFE$(r)_i$ *for all non-faulty replicas in the client group, where r is the corresponding extended reply containing either the result proposed in o or $\langle$*ABORT*$, t\rangle$, with t the timestamp of the request.*

Automaton APP$_i$ schedules requests until it becomes blocked because it requires the result of some request to make further progress. When it becomes blocked, the output action APP-TO-CFE$(\langle$LOOKUP$\rangle)_i$ can execute, which causes *lookup-pending*$_i$ to be set at CFE$_i$. The output action CFE-TO-APP$(m)_i$ can then execute, where m is the head of the reply queue. At this point, the application updates its state using the new result, and can schedule further requests. The following lemma states that the results accepted by the application are identical at each non-faulty replica. In the following, $\mathcal{M}$ is the universe of messages.

Lemma 8 (Determinacy of Application Input). *Let β be the trace of a fair execution, and $A_i = \{\textsc{cfe-to-app}(m)_i \mid m \in \mathcal{M}\}$ and $A_j = \{\textsc{cfe-to-app}(m)_j \mid m \in \mathcal{M}\}$ for non-faulty client replicas i and j. Then the projections $\beta|A_i$ and $\beta|A_j$ are equal up to the renaming $\{\textsc{cfe-to-app}(m)_i \mapsto \textsc{cfe-to-app}(m)_j \mid m \in \mathcal{M}\}$.*

7.3 Fault Handling

7.3.1 Unresponsive Designated Responder

If the designated responder is faulty or providing client replicas poor service, they may retransmit a request to all server replicas, e.g. after having timed out waiting for a reply. If a reply is in response to a multicast retransmission by the client replicas, then each server replica will respond directly to the client after receiving $f_c + 1$ matching requests. After a client replica i has received $f_s + 1$ correct extended reply messages with matching results, internal action PROCESS-INCOMING-REPLIES creates a new store request to apply the result, just as in Section 7.2.3.

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

Lemma 9 (Extended Replies to Extended Store Request). *Let $\beta = \alpha\alpha'$ be a fair execution and s be the final state of α. If s is the first state in which $f_s + 1$ matching extended replies $\langle r, S \rangle$ are in transit to a non-faulty client replica i, then α' contains the event $\textsc{filter-to-channel}(r_\mu, \{k_\mu\})_i$, where r_μ is the subsequent extended request of r and k_μ the identifier of client store primary.*

7.3.2 Faulty Client Group

If the number of faulty client replicas exceeds the f_c, we still want non-faulty server groups to to function correctly, with consistent state at all non-faulty nodes. The SFE automata only change the CLBFT protocol by requiring f_c+1 matching client requests before processing an operation, 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, the designated responder might not ever receive enough extended reply forwards to achieve quorum on the server side to send the extended reply to the client group. This does not affect correctness, though, because the client group is faulty to begin with.

7.3.3 Faulty Server Group

A faulty server group (in which the number of faulty replicas exceeds $f_s + 1$) 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 faulty server group could sign different results for different replicas. This is solved by using the client store. Each client replica submits its result to the store primary as a request to write to the store, and the request is executed only when the store primary has an $f_c + 1$ quorum of matching requests. In this way, the clients use the store for agreement on the result of the remote server request. However, it could be the case that two quorums can be formed by the clients' requests (e.g., if the faulty server gives one answer to half of the replicas and another to the other half). In this case, the store primary makes the decision about which value to commit (e.g., whichever quorum it receives first), and all replicas receive that result in their incoming queue at the same logical time. All store requests that correspond to a given server request use the same timestamp. If a second quorum of store requests arrives at the store primary after it has started the operation for the first one, the store primary will assign it a sequence number and process it, but as defined by CLBFT the second result will not execute at any of the store replicas, ensuring consistency.

7.4 Correctness

Our correctness condition is specified as the composition of an application APP (as described earlier) with a single unreplicated server automaton SERVER, whose external signature is as follows.

$$\begin{array}{ll} \text{Input:} & \text{APP-TO-CFE}(\langle \text{SCHEDULE}, t, g, o\rangle)_i \\ & \text{APP-TO-CFE}(\langle \text{SUSPECT-FAULTY}, t\rangle)_i \\ & \text{APP-TO-CFE}(\langle \text{LOOKUP}\rangle)_i \\ \text{Output:} & \text{CFE-TO-APP}(\langle t, r\rangle)_i \end{array}$$

We refer to the composition of APP and SERVER as A. In S, the APP automaton uses a CFE automaton as a proxy to the server. In A, we strip away CFE and compose APP directly with the SERVER automaton. Note that SERVER is essentially the same as in [2], with the exception that it handles asynchronous client requests. For our system S, however, the client falls within the fault-tolerance boundary, and so our correctness condition must specifically address the operation of the application. The following theorem states that our system S implements the system A.

Theorem 1 (Correctness). *Let β be a fair execution of S in which the number of faulty server replicas does not exceed f_s and the number of faulty client replicas does not exceed f_c. Let $\Sigma = \bigcup_{i \in c} (\mathit{acts}(\textsc{app}_i) - \{\textsc{app-to-cfe}(\langle\textsc{susp}$ Then there exists a fair execution γ of A such that, for any non-faulty client replica i, $(\beta|\textsc{app}_i)|\Sigma = (\gamma|\textsc{app})|\Sigma$ up to renaming of identifier suffixes.*

The execution γ can be constructed by induction on the length of β. Starting with A in its initial state, we insert steps into its execution in order as we scan *trace*(β). For the external actions, we add $\textsc{app-to-cfe}(\langle\textsc{schedule}, t, g, o\rangle)$ to γ after scanning the $f_c + 1$ occurences of that action with (with distinct identifier suffix), $\textsc{app-to-cfe}(\langle\textsc{suspect-faulty}, t\rangle)$ after scanning $2f_c + 1$ occurences of that action, and both $\textsc{app-to-cfe}(\langle\textsc{lookup}\rangle)$ and $\textsc{cfe-to-app}(\langle t, r\rangle)$ after scanning $f_c + 1$ occurences of the latter. For the internal actions, we add $\textsc{execute-result}(t, g, o)$ after scanning $f_s + 1$ occurences of the action $\textsc{sbe-to-sfe}(m = \langle\textsc{reply}, v, t, c, i, r\rangle_{\sigma_i})$, where m is the reply for request $\langle\textsc{request}, o, t, c\rangle_{\sigma_c}$, and we add $\textsc{execute-abort}(t, g, o)$ after scanning $f_c + 1$ occurences of $\textsc{cfe-to-app}(\langle t, \langle\textsc{abort}, t\rangle\rangle)$.

Moreover, a request is aborted by the client store only if at least $f_c + 1$ non-faulty clients suspected the server faulty. This implies that $\textsc{cfe-to-app}(t, \langle\textsc{abort}, t\rangle)$ only appears in γ after the event $\textsc{app-to-cfe}(\langle\textsc{suspect-faul}$ The following theorem states this formally.

Theorem 2 (ABORT Reply only from SUSPECT-FAULTY Requests). *Let $\beta = \alpha\alpha'$ be a fair execution of S and let $\textsc{cfe-to-app}(\langle t, \langle\textsc{abort}, t\rangle\rangle)_i$ be the first event of α' for some timestamp t and non-faulty replica i in client group c. Then α contains at least $2f_c + 1$ events $\textsc{app-to-cfe}(\langle\textsc{suspect-faulty}, t\rangle)_j$ for distinct replicas j in group c.*

8 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 the application's operation request and server's return value. 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. Analysis for each stage of our algorithm during normal operation is shown in the table below, in which we assume message digests and digital signatures are of constant length.

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)	$O(m^2 + n^2)$	$O(m^2 + n^2 + mn + \ell m + \ell n)$

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

9 Discussion

This section discusses optimizations, checkpointing, recovery, and garbage collection.

9.1 Optimizations

For uniformity of the CLBFT wrappers, each active client replica forwards to its store primary the reply received from the designated responder in order to carry out agreement on the reply value. However, since $f_s + 1$ server replicas sign the reply, the primary could safely begin the CLBFT agreement on the reply without waiting for f_c messages from its peers, so these messages need not be sent by a replica unless it times out waiting for a pre-prepare. This reduces latency by eliminating stage four, shown in Section 8.

If the number of faulty server replicas exceeds f_s, they could force the client group to run superfluous agreements by sending different valid reply bundles to different client replicas. To avoid this extra communication, the SFE for each store wrapper could check reply and pre-prepare messages for conflicting bundles, which would constitute proof that the server is faulty. In such cases, a replica could initiate a "suspect faulty" operation and participate only in a "suspect faulty" agreement for that timestamp value.

A faulty server primary can force its replicas to multicast their replies by delaying the preprepare. We believe these extra messages can be bounded without giving a faulty client group the opportunity to force a view change. Suspicious clients could give server replicas, in turn, the opportunity time the responsiveness of the primary by funnelling each initial request through a different server replica. Server replicas would gossip about primary responsiveness and, when sufficiently many agree, perform a view change.

Applications are deterministic, but replicas may run at different speeds. If fewer than $f_c + 1$ applications run far enough ahead, they may time out waiting for a reply from the server because too few of their peers have issued requests in time. The fast replicas would then multicast their requests to the server replicas, causing unnecessary message traffic. In practice, we expect client replicas to be kept sufficiently synchronized by occasionally consuming replies from their result queues in blocking operations. However, it is conceivable that if an application issues many asynchronous requests before blocking on a reply, some applications may run significantly ahead. Byzantine fault-tolerant clock synchronization [4] within the active client replica group could be used to overcome this problem by more closely aligning time-out periods for requests.

A short time-out period may cause a replica to multicast requests unneccessarily. To adapt to network conditions, our operation timer can double the timeout period whenever a timeout occurs. Since message delays do not grow faster than time, the value will eventually stabilize. Timeout values can be reset as part of the proactive recovery mechanism, as in CLBFT.

9.2 Recovery

The checkpoint and recovery mechanism of the CLBFT algorithm can be leveraged to preserve the state of the additional components on both client and server sides. By synchronizing on the consumption of the result queue to a particular store operation sequence number, we can ensure that each checkpoint done by the store contains consistent state information for all replicas, including the state of the CFE and the APP.

9.3 Garbage collection

Our formal algorithm description assumes unbounded space at each replica. This section describes how to bound the space required at each replica for non-faulty CFE and SFE replicas.

At the CFE, the only source of garbage is the reply buffer that accepts extended-replies from the channel. One could imagine having the CFE accept replies only to requests that it sent earlier. However, the replica may have fallen behind the other client replicas, in which case it could eventually make the request and then time out due to the lack of a reply (which it had earlier discarded) and multicast the request to all the server replicas. Therefore, we need a mechanism that bounds the space required by the reply buffer while minimizing the risk of discarding valid replies to requests the CFE may make in the future.

To this end, we use a *high watermark* and *low watermark* for timestamps. The low watermark of a replica is the lowest timestamp among the outstanding requests. The high watermark is some constant W greater than the low watermark. The APP will only send requests with timestamps in the range between the watermarks. The CFE will discard any replies that fall outside the range between the two watermarks. If the CFE buffers only one extended-reply for a given timestamp from a given replica, we can bound the size of the reply buffer as the product of W and the number of known replicas. By tuning W, one can trade off space usage against the risk of an unnecessary time-out and multicast.

At the SFE there are two potential sources of garbage. The first is the set $\textit{requests-incoming}_i$ that collects incoming requests. To bound this, the SFE can keep a separate incoming queue of a maximum size L_g for each known replica in group g. The constant L_g will be known by client replicas and non-faulty client replicas will ensure that no more than L_g incomplete requests to that server exist at any given time in their execution. This bounds the total number of elements in the incoming queues for each client group. The second source of garbage at the SFE is the set $\textit{replies-incoming}_i$ that holds extended reply messages from other server replicas. If a SFE replica only accepts replies to a known request that stipulated it as the designated responder, it runs the risk that it didn't know about the request since it has fallen behind the other replicas. (It can still act as the responder if it forms a quorum of replies.) However, if a replica keeps all replies, then faulty replicas could swamp the CFE with replies to requests that were never made, disguised as replies to future requests. Buffering only a fixed number of replies from each of the other server replicas places an upper bound on space. Various heuristics could be used to selectively purge old replies for best performance. Since the responder is only an optimization, discarding old replies does not affect correctness.

10 Future Work

We plan to implement our algorithm using CORBA [13] and the BASE [16] library. We will first implement a version without any of the stated optimizations and gradually incorporate each optimization in a configurable manner. This implementation will be used to conduct a comprehensive performance analysis, verify the practicality of our algorithm, and measure the impact of each optimization.

References

[1] S. Ahmed. A Scalable Byzantine Fault Tolerant Secure Domain Name System, 2001. Master's thesis, M.I.T.

[2] Miguel Castro and Barbara Liskov. A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm. Technical Report MIT-LCS-TM-597, M.I.T., Cambridge, MA, USA, 1999.

[3] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Third Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, Louisiana, February 1999. USENIX Association, Co-sponsored by IEEE TCOS and ACM SIGOPS.

[4] Ariel Daliot, Danny Dolev, and Hanna Parnas. Linear Time Byzantine Self-Stabilizing Clock Synchronization. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS-2003)*, La Martinique, France, December 2003.

[5] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, 1976.

[6] Charles Fry and Michael Reiter. Nested Objects in a Byzantine Quorum-Replicated System. In *23rd International Symposium on Reliable Distributed Systems (SRDS)*, pages 79–89. IEEE, 2004.

[7] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, pages 135–144. IEEE, 2004.

[8] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Trans. Inf. Syst. Secur.*, 4(4):371–406, 2001.

[9] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[10] Nancy Lynch and Mark Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[11] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware forWeb-Service Applications. In *Proceedings of the 15th Annual 24th IEEE Symposium on Reliable Distributed Systems*, 2005.

[12] Priya Narasimhan, Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Providing Support for Survivable CORBA Applications with the Immune System. In *International Conference on Distributed Computing Systems*, pages 507–516, 1999.

[13] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition, September 2001.

[14] Bart Preneel and Paul C. van Oorschot. MDx-MAC and building fast MACs from hash functions. *Lecture Notes in Computer Science*, 963:1–14, 1995.

[15] R. Rivest. The MD5 Message-Digest Algorithm, 1992.

[16] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 15–28, New York, NY, USA, 2001. ACM Press.

[17] Haraldur D. Thorvaldsson and Kenneth J. Goldman. Architecture and execution model for a survivable workflow transaction infrastructure. Technical Report WUCSE-2005-61, Department of Computer Science and Engineering, Washington University in St. Louis, December 2005.

[18] Jennifer L. Welch and Nancy A. Lynch. A modular drinking philosophers algorithm. *Distributed Computing*, 6(4):233–244, 1993.

A Background Theory and Notation

A background theory (i.e., uninterpreted function symbols and axioms constraining their values) is assumed in the automata definitions, defined below.

1. A choice operator ε is used to select an arbitrary element of a set. The result is defined only if the set is non-empty, and if the set contains only one element, then the result of the choice is just that element.

   (a) $\forall X.\ \exists x.\ x \in X \Rightarrow \exists y.\ y \in X \wedge y = \varepsilon X$

   (b) $\forall X.\ \exists x.\ X = \{x\} \Rightarrow x = \varepsilon X$

2. A collision-free message digest is assumed.

   (a) $\forall m, m'.\ \mathsf{digest}(m) = \mathsf{digest}(m') \Leftrightarrow m = m'$

3. The set of possible digital signatures of message m by signer c does not overlap with the signatures of any other message or responder.

   (a) $\forall m, m', c, c'.\ \neg(m = m' \wedge c = c') \Rightarrow \mathsf{signature}(m, c) \cap \mathsf{signature}(m', c') = \emptyset$

4. Construction and destruction of queues.

   (a) $\forall x.\ \mathsf{head}(\mathsf{append}(\emptyset, x)) = x$

   (b) $\forall \ell.\ \forall x, y.\ \mathsf{head}(\ell, x) = x \Rightarrow \mathsf{head}(\mathsf{append}(\ell, y)) = x$

   (c) $\forall x.\ \mathsf{tail}(\mathsf{append}(\emptyset, x)) = \emptyset$

   (d) $\forall \ell, m.\ \forall x, y.\ \mathsf{tail}(\ell) = m \Rightarrow \mathsf{tail}(\mathsf{append}(\ell, y)) = \mathsf{append}(m, y)$

5. A scaling function $\mathsf{scale}()$ which returns a replica identifier given a group identifier and natural number. This is used instead of mod to calculate the primary from a given view number and designated responder from a sequence number because we do not assume identifiers in $\mathbb{N}$.

6. Application behavior is encapsulated in two oracle functions: $\mathsf{oracle\text{-}update}()$ updates the application state based on the result of a request, and $\mathsf{oracle\text{-}requests}()$ produces a set of requests to be scheduled asynchronously and another.

7. The function $\mathsf{apply}()$ executes an operation at the server and updates a state variable. It is identical to the function g in [2].

Variable names in the automata definitions are selected to coincide with their domain: $o \in O$, the set of operations; $t \in T$, the set of timestamps; $g \in G$, the set of group identifiers; $c, i, j, \rho \in R$ and $X \subseteq R$, the set of replica identifiers; $v \in V$, the set of view numbers; $n \in \mathbb{N}$, the set of sequence numbers; $m, r \in \mathcal{M}$, the universe of messages; $\sigma \in \mathcal{S}$, the set of digital signatures; $d \in \mathcal{D}$, the set of message digests; $S \in \mathcal{D} \times R \times R \times \mathcal{S}$, the signed digests. For a message m and replica c, the notation m_{σ_c} is defined as $\langle m, \sigma \rangle$, with $\sigma \in \mathsf{signature}(m, c)$.

Free variables in set comprehensions are implicitly existentially quantified.

B CLBFT Characterization

We adopt a version of the CLBFT protocol that allows a given client to issue concurrent requests (with different timestamps) to the same server, where the server may commit the concurrent requests in any order. In other words, we allow the CLBFT protocol to commit requests in an order that differs from the timestamp order assigned by the client.

We characterize the behavior of the CLBFT protocol as the set of allowable sequences of input and output actions that may occur in fair executions.

Lemma 10 (CLBFT Operational Characterization). *Let $\beta = \alpha\alpha'$ be the projection of a fair execution of the CLBFT protocol onto non-faulty server replica i, and let π be the final event of α. Assume that in the final state of α, $\mathrm{view}_i = v$ and $\mathsf{primary}(v) = k$, and that R is the set replica identifiers in the replica group. Then the following propositions hold for β:*

1. *If $\pi = \text{SFE-TO-SBE}(\langle\text{REQUEST}, o, t, c\rangle_{\sigma_c})_i$, and $i \neq k$, then α' contains the event $\text{SBE-TO-SFE}(\langle\text{REQUEST}, o, t, c\rangle_{\sigma_i}, \{k\})_i$.*

2. *If $\pi = \text{SFE-TO-SBE}(m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c})_i$, and $i = k$, then α' contains either the event $\text{SBE-TO-SFE}(\langle\text{PRE-PREPARE}, v', n, m\rangle_{\sigma_i}, R - \{i\})_i$ for some $v' \geq v$ and n, or the event $\text{SFE-TO-SBE}(\langle\text{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_j})_i$ for some $\mathcal{V}$ and $\mathcal{O}$.*

3. *If $\pi = \text{SBE-TO-SFE}(\langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i}, R - \{i\})_i$ and $m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c}$, then α' contains contains the event $\text{SBE-TO-SFE}(\langle\text{REPLY}, v', t, c, i, r\rangle_{\sigma_i}, \{c\})_i$, for some $v' \geq v$ and r.*

4. *If $\pi = \text{SFE-TO-SBE}(\langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k})_i$ and $m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c}$, then α' contains the event $\text{SBE-TO-SFE}(\langle\text{REPLY}, v', t, c, i, r\rangle_{\sigma_i}, \{c\})_i$, for some $v' \geq v$ and r.*

5. *If $\pi = \text{SFE-TO-SBE}(\langle\text{NEW-VIEW}, v', \mathcal{V}, \mathcal{O}\rangle_{\sigma_j})_i$, then for each $m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c}$ and $p = \langle\text{PRE-PREPARE}, v', n, m\rangle_{\sigma_k} \in \mathcal{O}$, then α' contains the event $\text{SBE-TO-SFE}(\langle\text{REPLY}, v', t, c, i, r\rangle_{\sigma_i}, \{c\})_i$, for some $v' > v$ and r.*

6. *If $\pi = \text{SBE-TO-SFE}(\langle\text{VIEW-CHANGE}, v+1, \mathcal{P}, j\rangle_{\sigma_j}, R - \{i\})_i$, then for each $m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c}$ and $p = \langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k} \in \mathcal{P}$, then α contains either the event $\text{SFE-TO-SBE}(m)_i$, the event $\text{SFE-TO-SBE}(p)_i$, or the event $\text{SFE-TO-SBE}(\langle\text{VIEW-CHANGE}, v+1, \mathcal{P}, j\rangle_{\sigma_j})_i$ for some j and $\mathcal{P}$ with $p \in \mathcal{P}$.*

7. *If $\pi = \text{SBE-TO-SFE}(\langle\text{NEW-VIEW}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_j}, R - \{i\})_i$, then for each $m = \langle\text{REQUEST}, o, t, c\rangle_{\sigma_c}$ and $p = \langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_k} \in \mathcal{O}$, then α contains either the event $\text{SFE-TO-SBE}(m)_i$, the event $\text{SFE-TO-SBE}(p)_i$, or the event $\text{SFE-TO-SBE}(\langle\text{VIEW-CHANGE}, v+1, \mathcal{P}, j\rangle_{\sigma_j})_i$ for some j and $\mathcal{P}$ with $p \in \mathcal{P}$.*

C I/O Automata Definitions

This appendix contains complete I/O automaton specifications for the Unreplicated Correct Server (SERVER), Client Application (APP), Client Front End (CFE), Server Front End (SFE), and Message Channel (MC) automata.

C.1 Unreplicated Correct Server: SERVER

This automaton specification captures the behavior of a correct unreplicated server.

Signature

Input: $\text{APP-TO-CFE}(\langle \text{SCHEDULE}, t, o, g\rangle)_i$
$\text{APP-TO-CFE}(\langle \text{SUSPECT-FAULTY}, t\rangle)_i$
$\text{APP-TO-CFE}(\langle \text{LOOKUP}\rangle)_i$

Internal: $\text{EXECUTE-RESULT}(\langle t, g, o\rangle)_i$
$\text{EXECUTE-ABORT}(\langle t, g, o\rangle)_i$

Output: $\text{CFE-TO-APP}(\langle t, r\rangle)_i$

State

The state of the server.
$val \in \mathcal{V}$

Set when the app wants a reply.
$lookup\text{-}pending_i \in Bool$, initially $false$

Incoming request set.
$\text{in}_i \subset \mathcal{M}$

Outgoing reply set.
$out_i \subset \mathcal{M}$

Executed operation set.
$executed_i \subset \mathcal{M}$

Aborted operation set.
$aborted_i \subset \mathcal{M}$

Input Transitions

This input action executes when the APP *that represents a correct unreplicated client makes a request to the server by scheduling the operation o with the timestamp t. Group g must be the id of the current group.*

$\text{APP-TO-CFE}(\langle \text{SCHEDULE}, t, g, o\rangle)_i$
Eff:
$\text{in}_i \leftarrow \text{in}_i \cup \{\langle t, g, o\rangle\}$

A client application tells the server that it suspects that it is faulty. The unreplicated server ignores this.

$\text{APP-TO-CFE}(\langle \text{SUSPECT-FAULTY}, t\rangle)_i$
Eff:

A client application requests the result of some request it has previously made.

$\text{APP-TO-CFE}(\langle \text{LOOKUP}\rangle)_i$
Eff:
$lookup\text{-}pending_i \leftarrow true$

Internal Transitions

The server executes the operation found in a request and returns the value.

```
EXECUTE-RESULT$(t, g, o)_i$
Pre:
    $\langle t, g, o \rangle \in in_i$
Eff:
    $in_i \leftarrow in_i - \{\langle t, g, o \rangle\}$
    if $t \notin executed_i$
        $executed_i \leftarrow executed_i \cup \{t\}$
        $\langle r, val \rangle \leftarrow \mathsf{apply}(i, o, val)$
        $out_i \leftarrow out_i \cup \{\langle t, g, r \rangle\}$
```

The server chooses not to execute the operation in a request, and instead returns an ABORT *message.*

```
EXECUTE-ABORT$(t, g, o)_i$
Pre:
    $\langle t, g, o \rangle \in in_i \vee t \in aborted_i$
Eff:
    $in_i \leftarrow in_i - \{\langle t, g, o \rangle\}$
    if $t \notin aborted_i$
        $aborted_i \leftarrow aborted_i \cup \{t\}$
        $r \leftarrow \langle \text{ABORT}, t \rangle$
        $out_i \leftarrow out_i \cup \{\langle t, g, r \rangle\}$
```

Output Transitions

The server replies to a client application with the result of its execution.

```
CFE-TO-APP$(\langle t, r \rangle)_i$
Pre:
    $lookup\text{-}pending_i \wedge \langle t, r \rangle = \mathsf{head}(out_i)$
Eff:
    $lookup\text{-}pending_i \leftarrow false$
    $out_i \leftarrow \mathsf{tail}(out_i)$
```

C.2 Application Automaton: APP$_i$

This automaton specification captures the behavior of the client application.

Signature

Input: CFE-TO-APP$(\langle t, r \rangle)_i$

Output: APP-TO-CFE$(\langle \text{LOOKUP} \rangle)_i$
APP-TO-CFE$(\langle \text{SCHEDULE}, t, g, o \rangle)_i$
APP-TO-CFE$(\langle \text{SUSPECT-FAULTY}, t \rangle)_i$

State

The state of the oracle.

$state_i \in \mathcal{M}$

Whether the oracle is blocked waiting for a reply.

$blocked_i \in Bool$, initially $false$

Scheduled requests.

$scheduled_i \subseteq \mathcal{M}$, initially $\textsf{oracle-requests}(state_i)$

Requests that have been sent, but lack a result.

$pending_i \subseteq \mathcal{M}$, initially $\emptyset$

Input Transitions

When this action fires the app receives a reply to a previous request (or an upgrade). It uses this information to update its state and schedule the next set of operations. The action then clears theblocked$_i$ flag to enable the newly scheduled actions to be sent out.

```
CFE-TO-APP$(m = \langle t, r \rangle)_i$
Eff:
    $state_i \leftarrow \textsf{oracle-update}(state_i, m)$
    $scheduled_i \leftarrow \textsf{append}(scheduled_i, \textsf{oracle-requests}(state_i))$
    $pending_i \leftarrow pending_i - \{t\}$
    $blocked_i \leftarrow false$
```

Output Transitions

This output action fires when there is nothing scheduled to be sent to the CFE. *It sets the* blocked$_i$ *flag to true to signal that the* APP *is waiting to consume a reply/upgrade.*

```
APP-TO-CFE$(\langle \text{LOOKUP} \rangle)_i$
Pre:
    $scheduled_i = \emptyset \wedge \neg blocked_i$
Eff:
    $blocked_i \leftarrow true$
```

This output action fires when there is something in the scheduled$_i$ *queue to be sent to the* CFE. *It sends the operation o at the head of the queue to the* CFE *to be requested from group g with the timestamp t.*

```
APP-TO-CFE$(\langle \text{SCHEDULE}, t, g, o \rangle)_i$
Pre:
    $\langle t, g, o \rangle = \textsf{head}(scheduled_i)$
Eff:
    $scheduled_i \leftarrow \textsf{tail}(scheduled_i, \{\langle t, g, o \rangle\})$
    $pending_i \leftarrow pending_i \cup \{t\}$
```

This output action fires to signal the CFE *to send a* SUSPECT-FAULTY *request for timestamp t. This captures the mechanism for possibly aborting operations.*

```
APP-TO-CFE$(\langle \text{SUSPECT-FAULTY}, t \rangle)_i$
```

Pre:
$t \in pending_i$

C.3 Client Front End Automaton: CFE_i

This automaton specification captures the Client Front End behavior.

Signature

Input:
$\text{APP-TO-CFE}(\langle\text{SCHEDULE}, t, o, g\rangle)_i$
$\text{APP-TO-CFE}(\langle\text{SUSPECT-FAULTY}, t\rangle)_i$
$\text{APP-TO-CFE}(\langle\text{LOOKUP}\rangle)_i$
$\text{TIMER-TO-CFE}(g, r, b)_i$
$\text{CHANNEL-TO-FILTER}(\langle\langle\text{REPLY}, v, t, g', g, r\rangle_{\sigma_g}, S\rangle)_i$
$\text{SFE-TO-CFE}(\langle\text{REPLY}, v_{store}, t, g, g_{store}, result\rangle_{\sigma_{g_{store}}})_i$

Internal:
$\text{PROCESS-SERVER-REQUEST}(g, r, b)_i$
$\text{INITIATE-RESPONDER-CHANGE}(g)_i$
$\text{PROCESS-REPLIES-INCOMING}(\langle\langle\text{REQUEST}, o, t, g\rangle_{\sigma_g}, \langle d, \rho_{g'}, i\rangle, g'\rangle)_i$
$\text{PROCESS-RESPONDER-CHANGE}(\langle\text{RESPONDER-CHANGE}, t, \langle\tau, g'\rangle, v_{store}\rangle)_i$

Output:
$\text{FILTER-TO-CHANNEL}(m, X)_i$
$\text{CFE-TO-APP}(t, r)_i$
$\text{CFE-TO-TIMER}(g, r, b)_i$

State

A functional relation between replicas and replica groups. A group must have at least one member to be recognized as a valid replica group. The contents of the relation are left unspecified; it is assumed that it is non-empty and that contents do not change over time.

$group_i \subseteq R \times G$

A functional relation between replica groups and their fault tolerance requirement (FTR). A The contents of the relation are left unspecified with the stipulation that it contains an entry for each unique group id in $group_i$*. Contents do not change over time.*

$ftr_i \subseteq G \times \mathbb{N}$

A functional relation between replica groups and their primaries. The contents of the relation are left unspecified with the stipulation that it contains an entry for each unique group id in $group_i$*. Contents may change over time.*

$primary_i \subseteq G \times R$

A functional relation between replica groups and the number of times the responder has been changed for each group.The contents of the relation are left unspecified with the stipulation that it initially contains an entry for each unique group id g in $group_i$ *with a sequence value between zero and —*replicas(g)*—. Contents may change over time and the responder id for each group is the sequence value modulo group size.*

$responder\text{-}seq_i \subseteq G \times \mathbb{N}$

The group id of the client store. Must be initially set and does not change.

$\mathit{store}_i \in G$

The set of group ids for which CFE *hs requested a responder change that has not been completed.*

$\mathit{responder\text{-}change\text{-}pending}_i \subseteq G$ initially $\emptyset$

This set contains elements of the form $\langle g, r, b\rangle$, where g is the destination group id, r is the CLBFT request, and b is the request bundle. Each element represents a request that has not yet been sent out by the CFE*.*

$\mathit{requests\text{-}new}_i \subseteq G \times \mathcal{M} \times \mathcal{M}$ initially $\emptyset$

This set contains elements of the form $\langle g, r, b\rangle$, where g is the destination group id, r is the CLBFT request, and b is the request bundle. Each element represents an extended request that has been sent and is currently pending, i.e. a valid quorum of extended replies (or a single client store reply) has not been received for any r contained in an element of this set.

$\mathit{requests\text{-}current}_i \subseteq G \times \mathcal{M} \times \mathcal{M}$ initially $\emptyset$

This set contains elements of the form $\langle r, j, \sigma\rangle$ where r is the reply message, j is the sender, and σ is the valid signature of j for message r. Each element represents an extended reply received by the CFE*.*

$\mathit{replies\text{-}incoming}_i \subseteq \mathcal{M} \times R \times S$ initially $\emptyset$

This fifo queue contains replies received from the client store.

$\mathit{store\text{-}replies}_i \subseteq \mathcal{M}$ initially $\emptyset$

This set contains elements of the form $\langle m, X\rangle$ where m is the extended request and X contains the replica ids of all the recipients. Each element represents a message to be sent out on the message channel to other replicas.

$\mathit{channel\text{-}buffer}_i \subseteq \mathcal{M} \times \mathscr{P}R$, initially $\emptyset$

This set contains elements of the form $\langle g, r, b\rangle$, where g is the destination group id, r is the CLBFT request, and b is the request bundle. Each element represents a request for which a timer should be started by the TIMER.

$\mathit{timer\text{-}buffer}_i \subseteq G \times \mathcal{M} \times \mathcal{M}$, initially $\emptyset$

Auxiliary Functions

Returns a tuple that represents a request message for operation o with timestamp t. g is the id of the receiving group for this request.

$\mathsf{server\text{-}request}(t, o, g) =$

$\quad \langle g, r = \langle \text{REQUEST}, o, t, \mathit{group}_i(i)\rangle_{\sigma_{\mathit{group}_i(i)}}, \langle \mathsf{digest}(r), \mathsf{scale}(\mathit{responder\text{-}seq}_i(g), g), i\rangle_{\sigma_i}\rangle$

Returns a tuple that represents a request message for a RESPONDER-CHANGE *operation for group. The sender of the request for a responder change for group g is the logical group $\mathit{group}_i(i).g$. The client store (*SFE*) knows about the existence of the logical groups. Using this notation enables the use of the responder sequence n for a group as the timestamp for a responder change request. Since n is incremented in unison by the client group this enables each* CFE *to send responder change requests and expect the other replicas to send matching requests eventually.*

$\mathsf{responder\text{-}change\text{-}request}(g) =$

$\quad \langle \mathit{store}_i, r = \langle \text{REQUEST}, \langle \text{RESPONDER-CHANGE}, g\rangle, \mathit{responder\text{-}seq}_i(g), \mathit{group}_i(i).g\rangle_{\sigma_{\mathit{group}_i(i)}}, \langle \mathsf{digest}(r), \emptyset, i.g\rangle_{\sigma_i}\rangle$

Returns a tuple that represents a request message to be sent to the client store. Operation o could be a SUSPECT-FAULTY *or* APPLY-RESULT*.*

$\mathsf{store\text{-}request}(t, o) =$
$\quad \mathsf{server\text{-}request}(t, o, store_i)$

Returns the set of request tuples in requests-current$_i$ *that have a timestamp matching t.*

$\mathsf{requests\text{-}for}(t) =$
$\quad \{x \mid \exists b, o, g, g'.\ \langle g', r = \langle \text{REQUEST}, o, t, g\rangle_{\sigma_{\mathsf{group}_i(i)}}, b\rangle \in \mathit{requests\text{-}current}_i\}$

Returns a set of replies contained in the replies-incoming$_i$ *buffer containing the reply messages for the given request, where the result and server group view contained in each reply are equal to the result and view specified in the parameters.*

$\mathsf{replies\text{-}for\text{-}result}(\langle \text{REQUEST}, o, t, g\rangle_{\sigma_g}, g', \mathit{result}, v') =$
$\quad \{x \in \mathit{replies\text{-}incoming}_i \mid \exists j.\ x = \langle \langle \text{REPLY}, v', t, g, g', \mathit{result}\rangle_{\sigma_{g'}}, j, \sigma_j\rangle\}$

Returns ids of all the replicas that belong to the given group.

$\mathsf{replicas}(g) =$
$\quad \{x \mid \langle x, g\rangle \in \mathit{known\text{-}replica\text{-}groups}_i\}$

Auxiliary Predicates

Holds iff the first element of a message tuple m is the symbol τ.

$\mathit{tag}(m, \tau) \Leftrightarrow$
$\quad \exists m'.\ m = \langle \tau, m'\rangle$

Holds iff σ is a correct signature of m signed by the key of g, a group.

$\mathit{valid\text{-}group\text{-}sig}(\sigma, m, g) \Leftrightarrow$
$\quad \sigma \in \mathsf{signature}(m, g) \wedge \exists c.\ \langle c, g\rangle \in \mathit{group}_i$

Holds iff σ is a correct signature of m signed by the key of i, a replica, that is a member of group g.

$\mathit{valid\text{-}replica\text{-}sig}(\sigma, m, i, g) \Leftrightarrow$
$\quad \sigma \in \mathsf{signature}(m, i) \wedge \langle i, g\rangle \in \mathit{group}_i$

Holds iff a reply with a timestamp t exists in the incoming queue from the client store.

$\mathit{reply\text{-}received}(t) \Leftrightarrow$
$\quad \exists \tau, r, v, g.\ \langle \tau, t, r, v, g\rangle \in \mathit{store\text{-}replies}_i$

Holds iff the given reply was sent by a known replica j of a known group g' that received a request from the group g that must include the current replica i.

$\mathit{valid\text{-}server\text{-}reply}(\mathit{signed} = \langle \mathit{unsigned} = \langle \text{REPLY}, v', t, g, g', \mathit{result}\rangle, \sigma_{g'}\rangle, \sigma_j, j) \Leftrightarrow$
$\quad \mathit{valid\text{-}group\text{-}sig}(\sigma_{g'}, \mathit{unsigned}, g') \wedge \mathit{valid\text{-}replica\text{-}sig}(\sigma_j, \mathit{signed}, j, g') \wedge \mathsf{group}_i(i) = g$

Holds iff the given group id was the same as the group id of the current node or if the given group id was a concatenation of the group id of the current node with the id of a known group. The second case occurs during responder changes.

```
$same\text{-}group(g) \Leftrightarrow$
  $g = group_i(i) \vee (\exists g'.\ known\text{-}group(g') \wedge g = group_i(i).g')$
```

Holds iff the given group is a known group.

```
$known\text{-}group(g) \Leftrightarrow$
  $\exists c.\ \langle c, g\rangle \in known\text{-}replica\text{-}groups_i$
```

Input Transitions

This input action is used by the APP inform the CFE of scheduled operations. Each operation will have a unique timestamp t operation specification o, and the group id g of the server group that this operation should be invoked on. CFE takes this information, and if a reply to this request has not already been received from the client store (if the current replica had fallen behind) constructs an outgoing extended request, and puts it in it's requests-new$_i$ set to be sent out.

```
APP-TO-CFE($\langle$SCHEDULE$, t, o, g\rangle)_i$
Eff:
  if $\neg reply\text{-}received(t)$
    $requests\text{-}new_i \leftarrow requests\text{-}new_i \cup \{\mathsf{server\text{-}request}(t, o, g)\}$
```

This input action is used by the APP to "give up" on an operation it has already requested. The parameter includes the timestamp of that operation. CFE then takes the corresponding extended request out of the requests-current$_i$ set (to stop waiting for the operation result), and if a reply to this request has not already been received from the client store (if the current replica had fallen behind) constructs an extended request to be sent to the client store with the SUSPECT-FAULTY operation with the given timestamp. The same timestamp is used to ensure that either the operation result (obtained by the client store through other CFE replicas) or the SUSPECT-FAULTY ack is adopted by the client store, but not both. (CLBFT will execute only one operation for a given timestamp, even if several are committed.

```
APP-TO-CFE($\langle$SUSPECT-FAULTY$, t\rangle)_i$
Eff:
  if $\neg reply\text{-}received(t)$
    if $\exists g, o, b.\ \langle g, r = \langle \text{REQUEST}, o, t, group_i(i)\rangle_{\sigma_{group_i(i)}}, b\rangle \in requests\text{-}current_i$
      $requests\text{-}current_i \leftarrow requests\text{-}current_i - \{\langle g, r, b\rangle\}$
    $requests\text{-}new_i \leftarrow requests\text{-}new_i \cup \{\mathsf{store\text{-}request}(t, \langle \text{SUSPECT-FAULTY}, t\rangle)\}$
```

This action is used by the scheduler to ask the CFE for the next result in the queue.

```
APP-TO-CFE($\langle$LOOKUP$\rangle)_i$
Eff:
  $lookup\text{-}pending_i \leftarrow true$
```

This action is used by the operation timer to signal the CFE that the extended request, The extended request should be multicast to all the group members of the group g with the change that the designated responder slot should be set to ∅ to signal a retransmission to the server group and to request a direct response from server replicas.

```
TIMER-TO-CFE$(g, r, b = \langle d, \rho, id\rangle)_i$
```

```
Eff:
  if $\langle g, r, b\rangle \in \textit{requests-current}_i$
    $\textit{channel-buffer}_i \leftarrow \textit{channel-buffer}_i \cup \{\langle\langle r, \langle d, \emptyset, id\rangle\rangle, \mathsf{replicas}(g)\rangle\}$
```

This action fires when there is a incoming extended reply message or a extended reply forward to the CFE. *The set S may contain one (if the reply comes directly from a server group replica) or more (if the reply is a forwrded extended request from the responder) signatures. We validate each signature, construct a tuple with the reply and signature for each valid signature, and put that tuple in the* replies-incoming$_i$ *set for further processing.*

```
CHANNEL-TO-FILTER$(\langle m = \langle \text{REPLY}, v, t, g', g, r\rangle_{\sigma_g}, S\rangle)_i$
Eff:
  for all $\langle j, \sigma_j\rangle \in S$ do
    if $\textit{valid-server-reply}(m, j, \sigma_j)$
      $\textit{replies-incoming}_i \leftarrow \textit{replies-incoming}_i \cup \{\langle m, j, \sigma_j\rangle\}$
```

This action is used by the local store replica to inform CFE *of an agreed upon result of an operation request. The parameter is a regular CLBFT reply message that was generated at the local client store replica. Since the output action of the* SFE *at the client store replica fires in the execution order of the embedded CLBFT replica, this action fires in that same deterministic order. We remove the matching extended server request(s) (if any) from* requests-current$_i$*, categorize the result value contained in the reply to* RESPONDER-CHANGE, OPERATION-REPLY, *and*UPDATE-REPLY*, and include that classification in the* store-replies$_i$ *FIFO for further processing.*

```
SFE-TO-CFE$(\langle \text{REPLY}, v_{store}, t, g, g_{store}, result\rangle_{\sigma_{g_{store}}})_i$
Eff:
  if $\textit{same-group}(g, \text{group}_i(i))$
    $\textit{requests-current}_i \leftarrow \textit{requests-current}_i - \{\mathsf{requests\text{-}for}(t)\}$
    if $\textit{tag}(result, \text{RESPONDER-CHANGE})$
      $\textit{store-replies}_i \leftarrow \mathsf{append}(\textit{store-replies}_i, \langle \text{RESPONDER-CHANGE}, t, result, v_{store}\rangle)$
    else
      $\textit{store-replies}_i \leftarrow \mathsf{append}(\textit{store-replies}_i, \langle \text{OPERATION-REPLY}, t, result, v_{store}\rangle)$
  else
    $\textit{store-replies}_i \leftarrow \mathsf{append}(\textit{store-replies}_i, \langle \text{UPDATE-REPLY}, \emptyset, result, v_{store}\rangle)$
```

Internal Transitions

This internal action checks the requests-new$_i$ *bucket for new extended requests. If found, that extended request is removed from the* requests-new$_i$ *bucket and sent to both the operation timer as well to the primary of the destination group g.*

```
PROCESS-SERVER-REQUEST$(g, r, b)_i$
Pre:
  $sr = \langle g, r, b\rangle \in \textit{requests-new}_i$
Eff:
  $\textit{channel-buffer}_i \leftarrow \textit{channel-buffer}_i \cup \{\langle\langle r, b\rangle, \mathsf{primary}(g)\rangle\}$
  $\textit{timer-buffer}_i \leftarrow \textit{timer-buffer}_i \cup \{sr\}$
  $\textit{requests-current}_i \leftarrow \textit{requests-current}_i \cup \{sr\}$
  $\textit{requests-new}_i \leftarrow \textit{requests-new}_i - \{sr\}$;
```

This internal action can fire any time for a known replica group g provided that a responder change operation is not pending for that group. A new extended request to the client store is created with the special group id (composition of the current group id and g) and put in the requests-new$_i$ bucket to be sent out using the regular mechanism. The group id g is added to the pending set to disable further responder change operations until a response to the new extended request is received.

INITIATE-RESPONDER-CHANGE$(g)_i$

```
Pre:
    $\mathit{known\text{-}group}(g) \wedge g \notin \mathit{responder\text{-}change\text{-}pending}_i$
Eff:
    $\mathit{requests\text{-}new}_i \leftarrow \mathit{requests\text{-}new}_i \cup \{\mathsf{responder\text{-}change\text{-}request}(g)\}$
    $\mathit{responder\text{-}change\text{-}pending}_i \leftarrow \mathit{responder\text{-}change\text{-}pending}_i \cup \{g\}$
```

This internal action checks for the presence of matching reply sets of size $f_s + 1$ that corresponds to a pending current request. If such a reply set and request are found, a new extended request encapsulating the result values is constructed and put in requests-new$_i$ to be sent to the client store using the regular request mechanism. The primary$_i(g)$ functional relation is also updated using the server view seen in the reply set. Finally, the matching request is removed from the requests-current$_i$ set.

PROCESS-REPLIES-INCOMING$(sr = \langle m = \langle \text{REQUEST}, o, t, g\rangle_{\sigma_g}, \langle d, \rho_{g'}, i\rangle, g'\rangle)_i$

```
Pre:
    $sr \in \mathit{requests\text{-}current}_i \wedge \exists r, v'.\ |\mathsf{replies\text{-}for\text{-}result}(m, g', r, v')| \geq \mathsf{ftr}(g') + 1$
Eff:
    $\mathit{requests\text{-}new}_i \leftarrow \mathit{requests\text{-}new}_i \cup \{\mathsf{store\text{-}request}(t, \langle \text{APPLY-RESULT}, t, r\rangle)\}$
    $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i - \{sr\}$
    $\mathit{primary}_i(g') \leftarrow \mathsf{scale}(v', g')$
```

This internal action fires when the element at the head of the store-replies$_i$ queue is a RESPONDER-CHANGE *reply and there is a pending* APP *lookup as indicated by lookup-pending$_i$.* CFE *uses the provided information to change increment the responder sequence number. The primary id for the client store is also updated. Finally, we dequeue* RESPONDER-CHANGE *from store-replies$_i$ and leave the lookup-pending$_i$ set to true since we did not return anything to the* APP.

PROCESS-RESPONDER-CHANGE$(m = \langle \text{RESPONDER-CHANGE}, t, \langle \tau, g'\rangle, v_{store}\rangle)_i$

```
Pre:
    $\mathit{lookup\text{-}pending}_i \wedge m = \mathsf{head}(\mathit{store\text{-}replies}_i)$
Eff:
    $\mathit{primary}_i(\mathit{store}_i) \leftarrow \mathsf{scale}(v_{store}, \mathit{store}_i)$
    $\mathit{responder\text{-}seq}_i(g') \leftarrow \mathit{responder\text{-}seq}_i(g') + 1$
    $\mathit{responder\text{-}change\text{-}pending}_i \leftarrow \mathit{responder\text{-}change\text{-}pending}_i - \{g'\}$
    $\mathit{store\text{-}replies}_i \leftarrow \mathsf{tail}(\mathit{store\text{-}replies}_i)$
```

Output Transitions

The output action to put items into the channel.

FILTER-TO-CHANNEL$(m, X)_i$

```
Pre:
    $\langle m, X\rangle \in \mathit{channel\text{-}buffer}_i$
Eff:
    $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i - \{\langle m, X\rangle\}$
```

This output action fires when the element at the head of the store-replies$_i$ queue is not a RESPONDER-CHANGE *reply and there is a pending scheduler lookup as indicated by lookup-pending$_i$. We simply dequeue the head element from store-replies$_i$ and set the lookup-pending$_i$ set to false so that no items are dequeued from store-replies$_i$ until a new lookup is issued. The primary id in the for the client store is also updated.*

```
CFE-TO-APP$(t, r)_i$
Pre:
    $\mathit{lookup\text{-}pending}_i \wedge \exists \tau, v.\ \langle \tau, t, r, v\rangle = \mathsf{head}(\mathit{store\text{-}replies}_i) \wedge \tau \neq$ RESPONDER-CHANGE
Eff:
    $\mathit{primary}_i(\mathit{store}_i) \leftarrow \mathsf{scale}(v, \mathit{store}_i)$
    $\mathit{lookup\text{-}pending}_i \leftarrow \mathit{false}$
    $\mathit{store\text{-}replies}_i \leftarrow \mathsf{tail}(\mathit{store\text{-}replies}_i)$
```

The output action to pass extended request tuples to the timer.

```
CFE-TO-TIMER$(g, r, b)_i$
Pre:
    $\langle g, r, b\rangle = \mathsf{head}(\mathit{timer\text{-}buffer}_i)$
Eff:
    $\mathit{timer\text{-}buffer}_i \leftarrow \mathit{timer\text{-}buffer}_i - \{\langle g, r, b\rangle\}$
```

C.4 Server Front End Automaton: SFE$_i$

The SFE automaton coordinates communication between a CLBFT SBE automaton and a replicated client group.

Signature

Input:
$\textsc{sbe-to-sfe}(\langle\textsc{request}, o, t, g\rangle_{\sigma_g}, X)_i$
$\textsc{channel-to-filter}(\langle\langle\textsc{request}, o, t, g\rangle_{\sigma_g}, \langle d, \rho, c\rangle_{\sigma_c}\rangle)_i$
$\textsc{channel-to-filter}(\langle\textsc{forward}, \langle\textsc{request}, o, t, g\rangle_{\sigma_g}, S\rangle)_i$
$\textsc{sbe-to-sfe}(\langle\textsc{pre-prepare}, v, n, m\rangle_{\sigma_i}, X)_i$
$\textsc{channel-to-filter}(\langle\langle\textsc{pre-prepare}, v, n, m\rangle_{\sigma_j}, S\rangle)_i$
$\textsc{sbe-to-sfe}(\langle\textsc{view-change}, v+1, n, \mathcal{C}, \mathcal{P}, i\rangle_{\sigma_i}, X)_i$
$\textsc{channel-to-filter}(\langle\langle\textsc{view-change}, v+1, n, \mathcal{C}, \mathcal{P}, j\rangle_{\sigma_j}, \mathcal{S}\rangle)_i$
$\textsc{sbe-to-sfe}(\langle\textsc{new-view}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_i}, X)_i$
$\textsc{channel-to-filter}(\langle\langle\textsc{new-view}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_j}, \mathcal{S}\rangle)_i$
$\textsc{sbe-to-sfe}(\langle\textsc{reply}, v, t, g, i, r\rangle, X)_i$
$\textsc{channel-to-filter}(\langle\langle\textsc{reply}, v, t, g, g', r\rangle_{\sigma_j}, \{\langle j, \sigma_j\rangle\}\rangle)_i$
$\textsc{sbe-to-sfe}(m, X)_i$
$\textsc{channel-to-filter}(m)_i$

Internal:
$\textsc{request-confirm}(\langle\textsc{request}, o, t, g\rangle_{\sigma_g})_i$
$\textsc{reply-confirm}(t, g, r)_i$

Output:
$\textsc{sfe-to-sbe}(m)_i$
$\textsc{sfe-to-cfe}(m)_i$
$\textsc{filter-to-channel}(m, X)_i$

State

The set of messages to be sent out on the message channel.

channel-buffer$_i \subseteq \mathcal{M} \times \mathscr{P}R$, initially $\emptyset$

The set of messages to be sent in to the BFT node.

sbe-buffer$_i \subseteq \mathcal{M}$, initially $\emptyset$

The set of messages to be sent out to a co-located CFE node.

cfe-buffer$_i \subseteq \mathcal{M}$, initially $\emptyset$

The set of pairs of requests and signed digest bundles received from the message channel. These are originally from client replicas, but could also arrive indirectly from other replicas in the server group.

requests-incoming$_i \subseteq \mathcal{M} \times \mathcal{M}$, initially $\emptyset$

The set of pairs of requests and signed digest bundles received that correspond to operations currently in progress. There is at most one unique request in the set for each triple $\langle t, g, o\rangle$ (but the request appears multiple times, paired with different signed digest bundles).

requests-current$_i \subseteq \mathcal{M}$, initially $\emptyset$

This set stores group, operation, timestamp tuples to remember requests that were already sent to SBE*. This is used to prevent the same request from being forwarded into* SBE *more than once.*

processed-req$_i \subseteq G \times O \times T$, initially $G \times O \times \{-\infty\}$

The set of replies from other server replicas, sent to the designated responder. This state is only used at the designated responder for a given operation.

replies-incoming$_i \subseteq \mathcal{M}$, initially $\emptyset$

The timestamps of the replies to be sent by the designated responder out to the client group. This is used to prevent the same reply from being forwarded out to the client group more than once. This state is only used at the designated responder for a given operation.

$\mathit{replies\text{-}from\text{-}responder}_i \subseteq G \times T$, initially $G \times \{-\infty\}$

The timestamps of the replies to be sent to the designated responder. This is used to prevent replies from being forwarded out to the designated responder more than once.

$\mathit{replies\text{-}to\text{-}responder}_i \subseteq G \times T$, initially $G \times \{-\infty\}$

A functional relation between replicas and replica groups. A group must have at least one member to be recognized as a valid replica group. The contents of the relation are left unspecified; it is assumed that it is non-empty.

$\mathit{group}_i \subseteq R \times G$

A functional relation between replica groups and their fault-tolerance requirements (FTR). The contents of the relation are left unspecified; it is assumed that it is initially non-empty.

$\mathit{ftr}_i \subseteq G \times \mathbb{N}$

The group id of the client cfe-twin.

$\mathit{twin}_i \in G$

Auxiliary Functions

Returns the set of signed digest bundles corresponding to a particular request and designated responder. This set is drawn from the set of all incoming requests.

$$\mathsf{incoming\text{-}bundles}(r = \langle \text{REQUEST}, o, t, g\rangle_{\sigma_g}, \rho) =$$
$$\{x \mid \langle r, x = \langle d, \rho, c\rangle_{\sigma_c}\rangle \in \mathit{requests\text{-}incoming}_i\}$$

Returns the set of signed digest bundles corresponding to a request currently executing. This set is drawn from the set of current requests.

$$\mathsf{current\text{-}bundles}(r = \langle \text{REQUEST}, o, t, g\rangle_{\sigma_g}) =$$
$$\{x \mid \langle r, x = \langle d, \rho, c\rangle_{\sigma_c}\rangle \in \mathit{requests\text{-}current}_i\}$$

Returns the replica id of the designated responder for a current request being processed.

$$\mathsf{responder}(\langle \text{REPLY}, v, t, g, g', r\rangle) =$$
$$\varepsilon\{\rho \mid \langle\langle \text{REQUEST}, o, t, g\rangle_{\sigma_g}, \langle d, \rho, c\rangle_{\sigma_c}\rangle \in \mathit{requests\text{-}current}_i\}$$

Returns the set of digital signatures found with replies sent to the designated responder from other server replicas. This state is only used at the designated responder for a given operation.

$$\mathsf{reply\text{-}signatures}(t, g, r) =$$
$$\{\langle j, \sigma_j\rangle \mid \langle\langle \text{REPLY}, v, t, g, g', r\rangle_{\sigma_j}, j\rangle \in \mathit{replies\text{-}incoming}_i\}$$

Returns the set of replica identifiers that are known to belong to the group g. Server replicas use this information to verify the authenticity of signatures and to know where to send replies.

$$\mathsf{replicas}(g) =$$
$$\{x \mid \langle x, g\rangle \in \mathit{group}_i\}$$

Auxiliary Predicates

Holds iff the first element of a message tuple m is the symbol τ.

$\mathit{tag}(m, \tau) \Leftrightarrow$

$\quad \exists m'.\ m = \langle \tau, m' \rangle$

Holds iff d is the message digest of m.

$\mathit{valid\text{-}digest}(d, m) \Leftrightarrow$

$\quad d = \mathsf{digest}(m)$

Holds iff σ is a correct signature of m signed by the key of g, a group.

$\mathit{valid\text{-}group\text{-}sig}(\sigma, m, g) \Leftrightarrow$

$\quad \sigma \in \mathsf{signature}(m, g) \land \exists c.\ \langle c, g \rangle \in \mathit{group}_i$

Holds iff σ is a correct signature of m signed by the key of j, a replica, that is a member of group g.

$\mathit{valid\text{-}replica\text{-}sig}(\sigma, m, j, g) \Leftrightarrow$

$\quad \sigma \in \mathsf{signature}(m, j) \land \langle j, g \rangle \in \mathit{group}_i$

Holds iff $\langle r, b \rangle$ is a valid request, viz. that the digest d is correct for the request r, the signature σ_g on r is correct, and the signature σ_c on b is correct and by a member of group g.

$\mathit{valid\text{-}request}(r = \langle \text{REQUEST}, o, t, g \rangle_{\sigma_g}, b = \langle d, \rho, c \rangle_{\sigma_c}) \Leftrightarrow$

$\quad \mathit{valid\text{-}digest}(d, r) \land \mathit{valid\text{-}group\text{-}sig}(\sigma_g, r, g) \land \mathit{valid\text{-}replica\text{-}sig}(\sigma_c, b, c, g)$

Holds iff the set of signed digest bundles S is correct w.r.t. a request r, viz. that S has at least $f_c + 1$ signed digest bundles, and there is some consistent designated responder ρ for which each request and signed digest bundle is individually correct.

$\mathit{valid\text{-}request\text{-}bundles}(r = \langle \text{REQUEST}, o, t, g \rangle_{\sigma_g}, S) \Leftrightarrow$

$\quad |S| \geq \mathit{ftr}_i(g) + 1 \land (\mathit{tag}(o, \text{SUSPECT-FAULTY}) \Rightarrow |S| \geq 2\mathit{ftr}_i(g) + 1 \land g = \mathit{twin}_i) \land (\exists \rho.\ \forall b \in S.\ \exists c.\ b = \langle \mathsf{digest}(r), \rho, c \rangle \land \mathit{valid\text{-}request}(r, b) \land (\exists b' \in S.\ b' = \langle \mathsf{digest}(r), \rho, c \rangle \Rightarrow b = b'))$

Holds iff the set of signed digest bundles S is correct w.r.t. a request r, viz. that S has at least $f_c + 1$ signed digest bundles, and there is some consistent designated responder ρ for which each request and signed digest bundle is individually correct.

$\mathit{valid\text{-}reply\text{-}signatures}(S) \Leftrightarrow$

$\quad |S| \geq \mathit{ftr}_i(\mathrm{group}_i(i)) + 1$

Holds iff the signature is correct.

$\mathit{valid\text{-}reply}(m = \langle \text{REPLY}, v, t, g, g', r \rangle, j, \sigma_j) \Leftrightarrow$

$\quad g' = \mathit{group}_i(i) \land \mathit{valid\text{-}replica\text{-}sig}(\sigma_j, m, j, g')$

Input Transitions

A request forwarded to another server replica. We attach the set of signed digests for the request and add the tuple to the outgoing message channel set. The set of signed digests must exist because, in order to have a request to send, the automaton must first have received the request from $f_c + 1$ clients and saved their signed digests in $\mathit{requests\text{-}current}_i$.

$\text{SBE-TO-SFE}(r = \langle \text{REQUEST}, o, t, g \rangle_{\sigma_i}, X)_i$

```
Eff:
  $m \leftarrow \langle\text{FORWARD}, r, \mathsf{current\text{-}bundles}(r)\rangle$
  $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle m, X\rangle\}$
```

A request is received from a client replica, along with a signed digest bundle corresponding to the request. The pair of message and bundle are saved in the requests-incoming$_i$ *set*

```
CHANNEL-TO-FILTER($\langle r = \langle\text{REQUEST}, o, t, g\rangle_{\sigma_g}, b = \langle d, \rho, c\rangle_{\sigma_c}\rangle)_i$
Eff:
  if $\mathit{valid\text{-}request}(r, b)$
    $\mathit{requests\text{-}incoming}_i \leftarrow \mathit{requests\text{-}incoming}_i \cup \{\langle r, b\rangle\}$
```

A request is received from the message channel, along with a set of signed digest bundles corresponding to this request. The request is forwarded from another server replica. The set contains the signed digest bundles from $f_c + 1$ clients that correspond to $f_c + 1$ requests received at that server replica. The message–bundle pairs are saved in the requests-current$_i$ *set.*

```
CHANNEL-TO-FILTER($\langle\text{FORWARD}, m = \langle\text{REQUEST}, o, t, g\rangle_{\sigma_g}, S\rangle)_i$
Eff:
  if $\mathit{valid\text{-}request\text{-}bundles}(m, S) \wedge \neg\exists d, c.\ \langle d, \emptyset, c\rangle \in \mathsf{current\text{-}bundles}(m)$
    for all $b \in \mathsf{current\text{-}bundles}(m)$ do
      $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i - \{\langle m, b\rangle\}$
    for all $b \in S$ do
      $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i \cup \{\langle m, b\rangle\}$
    $\mathit{sbe\text{-}buffer}_i \leftarrow \mathit{sbe\text{-}buffer}_i \cup \{m\}$
```

The local BFT node sends a pre-prepare message to other server replicas. We attach a set of $f_c + 1$ signed digest bundles to corresponding to the request piggybacked within the pre-prepare. We know such a set exists for reasons similar to the SBE-TO-SFE(REQUEST)$_i$ *case.*

```
SBE-TO-SFE($p = \langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_i}, X)_i$
Eff:
  $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle p, \mathsf{current\text{-}bundles}(m)\rangle, X\rangle\}$
```

A pre-prepare message and set of signed digest bundles is received from the message channel, sent by a server replica. If all of the digests are valid (including that there are $f_c + 1$), then this piggybacked message and digests becomes the current request for this group and operation, and the messages and bundles are added to requests-current$_i$ *after removing the old set for this group. The pre-prepare message is then added to the set of messages to be sent into the local BFT node.*

```
CHANNEL-TO-FILTER($\langle\langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j}, S\rangle)_i$
Eff:
  if $\mathit{valid\text{-}request\text{-}bundles}(m, S) \wedge \neg\exists d, c.\ \langle d, \emptyset, c\rangle \in \mathsf{current\text{-}bundles}(m)$
    for all $b \in \mathsf{current\text{-}bundles}(m)$ do
      $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i - \{\langle m, b\rangle\}$
    for all $b \in S$ do
      $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i \cup \{\langle m, b\rangle\}$
    $\mathit{sbe\text{-}buffer}_i \leftarrow \mathit{sbe\text{-}buffer}_i \cup \{\langle\text{PRE-PREPARE}, v, n, m\rangle_{\sigma_j}\}$
```

A view-change message is sent out from the local BFT node to other server replicas via the message channel. We append a set of sets of signed digests bundles corresponding to the request piggybacked in each pre-prepare in the set $\mathcal{P}$ and add it to the set of messages to be sent on the channel. We know we have these sets of *digests for reasons similar to the* $\textsc{sbe-to-sfe}(\textsc{request})_i$ *case.*

```
$\textsc{sbe-to-sfe}(V = \langle\textsc{view-change}, v+1, n, \mathcal{C}, \mathcal{P}, i\rangle_{\sigma_i}, X)_i$
Eff:
  $\mathcal{S} \leftarrow \bigcup_{\langle\textsc{pre-prepare},v,n,m\rangle\in\mathcal{P}}\{\mathsf{current\text{-}bundles}(m)\}$
  $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle V, \mathcal{S}\rangle, \{\mathsf{primary}(v)\}\rangle\}$
  $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle V, \emptyset\rangle, \mathsf{replicas}(i) - \{\mathsf{primary}(v)\}\rangle\}$
```

A view-change message and set of sets of signed digest bundles is received from another server replica via the message channel. If for each request piggybacked on the pre-prepares in $\mathcal{P}$ *there is a valid set of signed digest bundles* $S \in \mathcal{S}$*, and if* $\mathcal{S}$ *contains nothing else, then the sets of digest bundles are added with their corresponding requests as the current requests for each group. The view-change message is added to the set of messages to be sent to the local BFT node. If this node is the primary, the signed digest bundles will be used as proof that the requests were made to be sent in the new-view messages.*

```
$\textsc{channel-to-filter}(\langle V = \langle\textsc{view-change}, v, n, \mathcal{C}, \mathcal{P}, j\rangle_{\sigma_j}, \mathcal{S}\rangle)_i$
Eff:
  if $\mathsf{primary}(v) \neq i \vee (|\mathcal{S}| = |\mathcal{P}| \wedge \forall\langle\textsc{pre-prepare}, v, n, m\rangle \in \mathcal{P}.\ \exists S \in \mathcal{S}.$
  $\mathit{valid\text{-}request\text{-}bundles}(m, S))$
    for all $\langle\textsc{pre-prepare}, v, n, m\rangle \in \mathcal{P}$ do
      if $\exists S \in \mathcal{S}.\ \mathit{valid\text{-}request\text{-}bundles}(m, S) \wedge \neg\exists d, c.\ \langle d, \emptyset, c\rangle \in \mathsf{current\text{-}bundles}(m)$
        for all $b \in \mathsf{current\text{-}bundles}(m)$ do
          $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i - \{\langle m, b\rangle\}$
        for all $b \in S$ do
          $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i \cup \{\langle m, b\rangle\}$
    $\mathit{sbe\text{-}buffer}_i \leftarrow \mathit{sbe\text{-}buffer}_i \cup \{V\}$
```

A new-view message is sent by the local BFT node to other server replicas. For each pre-prepare in $\mathcal{O}$*, a set of signed digest bundles is attached, similar to the* $\textsc{sbe-to-sfe}(\textsc{view-change})_i$ *case. Here, however, we omit certificates for the special null messages. We know the signed digest bundles exist for similar reasons to the* $\textsc{sbe-to-sfe}(\textsc{view-change})_i$ *case. The new-view message and set of sets of digest bundles is added to the set of messages outgoing on the message channel.*

```
$\textsc{sbe-to-sfe}(N = \langle\textsc{new-view}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_i}, X)_i$
Eff:
  $\mathcal{S} \leftarrow \bigcup_{\langle\textsc{pre-prepare},v,n,m\rangle\in\mathcal{P}}\{\mathsf{current\text{-}bundles}(m)\}$
  $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle N, \mathcal{S}\rangle, X\rangle\}$
```

A new-view message and set of sets of signed digests bundles is received from a server replica via the message channel. This case is nearly identical to the $\textsc{channel-to-filter}(\textsc{view-change})_i$ *case.*

```
$\textsc{channel-to-filter}(\langle N = \langle\textsc{new-view}, v+1, \mathcal{V}, \mathcal{O}\rangle_{\sigma_j}, \mathcal{S}\rangle)_i$
Eff:
  if $\forall\langle\textsc{pre-prepare}, v, n, m\rangle \in \mathcal{O}.\ m \neq \emptyset \Rightarrow \exists S \in \mathcal{S}.$
  $\mathit{valid\text{-}request\text{-}bundles}(m, S)$
    for all $\langle\textsc{pre-prepare}, v, n, m\rangle \in \mathcal{P}$ do
      if $m \neq \emptyset \wedge \exists S \in \mathcal{S}.\ \mathit{valid\text{-}request\text{-}bundles}(m, S) \wedge \neg\exists d, c.\ \langle d, \emptyset, c\rangle \in \mathsf{current\text{-}bundles}(m)$
        for all $b \in \mathsf{current\text{-}bundles}(m)$ do
          $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i - \{\langle m, b\rangle\}$
        for all $b \in S$ do
          $\mathit{requests\text{-}current}_i \leftarrow \mathit{requests\text{-}current}_i \cup \{\langle m, b\rangle\}$
    $\mathit{sbe\text{-}buffer}_i \leftarrow \mathit{sbe\text{-}buffer}_i \cup \{N\}$
```

A reply is sent from the local BFT node to the client. We forward everything to the cfe twin's scheduler queue. If the destination is not *the twin$_i$, then we if we haven't already sent a reply for this request and the responder is not $\emptyset$, the message is sent to the designated responder. Otherwise, the message is sent directly to all the clients in the replica group.*

```
$\textsc{sbe-to-sfe}(m = \langle \textsc{reply}, v, t, g, i, r\rangle_{\sigma_i}, \{g\})_i$
Eff:
  $\mathit{cfe\text{-}buffer}_i \leftarrow \mathsf{append}(\mathit{cfe\text{-}buffer}_i, m)$
  if $g \neq \mathit{twin}_i$
    $m' \leftarrow \langle \textsc{reply}, v, t, g, \mathit{group}_i(i), r\rangle_{\sigma_i}$
    $S \leftarrow \{\langle i, \sigma_i\rangle\}$
    if $\langle g, t\rangle \notin \mathit{replies\text{-}to\text{-}responder}_i \wedge \mathsf{responder}(m') \neq \emptyset$
      $\mathit{replies\text{-}to\text{-}responder}_i \leftarrow \mathit{replies\text{-}to\text{-}responder}_i \cup \{\langle g, t\rangle\}$
      $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle m', S\rangle, \{\mathsf{responder}(m')\}\rangle\}$
    else
      $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle\langle m', S\rangle, \mathsf{replicas}(g)\rangle\}$
```

A reply is received from another server replica. If the reply is valid, then the reply is added to replies-incoming$_i$.

```
$\textsc{channel-to-filter}(\langle m = \langle \textsc{reply}, v, t, g, g', r\rangle_{\sigma_j}, \{\langle j, \sigma_j\rangle\}\rangle)_i$
Eff:
  if $\mathit{valid\text{-}reply}(m, i, j)$
    $\mathit{replies\text{-}incoming}_i \leftarrow \mathit{replies\text{-}incoming}_i \cup \{\langle m, j\rangle\}$
```

A message with tag other than those previously handled is sent. The message is directly added to the set of messages outgoing via the message channel.

```
$\textsc{sbe-to-sfe}(m, X)_i$
Eff:
  if $\neg(\mathit{tag}(m, \textsc{request}) \vee \mathit{tag}(m, \textsc{pre-prepare}) \vee \mathit{tag}(m, \textsc{view-change}) \vee$
  $\mathit{tag}(m, \textsc{new-view}) \vee \mathit{tag}(m, \textsc{reply}))$
    $\mathit{channel\text{-}buffer}_i \leftarrow \mathit{channel\text{-}buffer}_i \cup \{\langle m, X\rangle\}$
```

A message with tag other than those previously handled is received from the channel. The message is directly added to the set of messages outgoing local BFT node.

```
$\textsc{channel-to-filter}(m)_i$
Eff:
  if $\neg(\mathit{tag}(m, \textsc{request}) \vee \mathit{tag}(m, \textsc{pre-prepare}) \vee \mathit{tag}(m, \textsc{view-change}) \vee$
  $\mathit{tag}(m, \textsc{new-view}) \vee \mathit{tag}(m, \textsc{reply}))$
    $\mathit{sbe\text{-}buffer}_i \leftarrow \mathit{sbe\text{-}buffer}_i \cup \{m\}$
```

Internal Transitions

A set of $f_c + 1$ valid requests exists, and we prepare to send the request into the local BFT node. We only do so if a set of sufficient size exists and we haven't already processed this request (which we know is the case if the last request time is less than the timestamp on this message). Similar to the $\textsc{channel-to-filter}(\textsc{pre-prepare})_i$ *case, the last request timestamp is updated and the request is added to the set of messages outgoing to the local BFT node.*

```
$\textsc{request-confirm}(m = \langle \textsc{request}, o, t, g\rangle_{\sigma_g})_i$
```

```
Pre:
  $\exists\rho.\ \rho \neq \emptyset \wedge \textit{valid-request-bundles}(m, \textsf{incoming-bundles}(m, \rho)) \wedge \langle g, o, t\rangle \notin \textit{processed-req}_i$
Eff:
  $\textit{processed-req}_i \leftarrow \textit{processed-req}_i \cup \{\langle g, o, t\rangle\}$
  for all $b \in \textsf{current-bundles}(m)$ do
    $\textit{requests-current}_i \leftarrow \textit{requests-current}_i - \{\langle m, b\rangle\}$
  for all $b \in \textsf{incoming-bundles}(m, \rho)$ do
    $\textit{requests-current}_i \leftarrow \textit{requests-current}_i \cup \{\langle m, b\rangle\}$
  $\textit{sbe-buffer}_i \leftarrow \textit{sbe-buffer}_i \cup \{m\}$
```

A set of $f_s + 1$ valid replies exists, and we prepare to send the reply to the set of client replicas. A set of digital signatures is attached that were originally sent with the replies from the server replicas. The last reply time is updated so that the reply is not sent again. (If the client times out and sends the requests again, server replicas will sent their reply directly, and not through the designated responder.)

```
REPLY-CONFIRM$(t, g, r)_i$
Pre:
  $S \leftarrow \textsf{reply-signatures}(t, g, r)$
  $\textit{valid-reply-signatures}(S) \wedge \langle g, t\rangle \notin \textit{replies-from-responder}_i$
Eff:
  $m \leftarrow \langle \text{REPLY}, v, t, g, \textit{group}_i(i), r\rangle_{\sigma_i}$
  $\textit{replies-from-responder}_i \leftarrow \textit{replies-from-responder}_i \cup \{\langle g, t\rangle\}$
  $\textit{channel-buffer}_i \leftarrow \textit{channel-buffer}_i \cup \{\langle\langle m, S\rangle, \textsf{replicas}(g)\rangle\}$
```

Output Transitions

This output action is used in passing messages to the SBE.

```
SFE-TO-SBE$(m)_i$
Pre:
  $m \in \textit{sbe-buffer}_i$
Eff:
  $\textit{sbe-buffer}_i \leftarrow \textit{sbe-buffer}_i - \{m\}$
```

This output action is used in sending messages to CFE *when this replica is part of the client store for some client group.*

```
SFE-TO-CFE$(m)_i$
Pre:
  $m = \textsf{head}(\textit{cfe-buffer}_i)$
Eff:
  $\textit{cfe-buffer}_i \leftarrow \textsf{tail}(\textit{cfe-buffer}_i)$
```

This output action is used in sending messages to other replicas using the message channel.

```
FILTER-TO-CHANNEL$(m, X)_i$
Pre:
  $\langle m, X\rangle \in \textit{channel-buffer}_i$
Eff:
  $\textit{channel-buffer}_i \leftarrow \textit{channel-buffer}_i - \{\langle m, X\rangle\}$
```

C.5 Message Channel Automaton: MC

This automaton specification captures the message channel.

Signature

Input:	FILTER-TO-CHANNEL$(m, X)_i$
Internal:	MISBEHAVE(m, X, X')
Output:	CHANNEL-TO-FILTER$(m)_i$

State

$\mathit{wire} \subseteq \mathcal{M} \times \mathscr{P}R$, initially $\emptyset$

Input Transitions

Replica i multicasts message m to all replicas $j \in X$

FILTER-TO-CHANNEL$(m, X)_i$

Eff:

$\mathit{wire} \leftarrow \mathit{wire} \cup \{\langle m, X\rangle\}$

Internal Transitions

The channel misbehaves by sending messages to replicas other than those intended by the sender, but we assume that it eventually delivers to all intended recipients.

MISBEHAVE$(m, X, X')_i$

Pre: $\langle m, X\rangle \in \mathit{wire}$

Eff: $\mathit{wire} \leftarrow \mathit{wire} - \{\langle m, X\rangle\} \cup \{\langle m, X \cup X'\rangle\}$

Output Transitions

Message m is delivered to replica i

CHANNEL-TO-FILTER$(m)_i$

Pre: $\exists\langle m, X\rangle \in \mathit{wire}.\ i \in X$

Eff: $\mathit{wire} \leftarrow \mathit{wire} - \{\langle m, X\rangle\} \cup \{\langle m, X - \{i\}\rangle\}$

D Exact Complexity Analysis

We count, for each communication step in the algorithm, the number of messages at that step, the number of application data payloads (requests or replies), the number of message digests, and the number of signatures. In the column headings, U (for unreplicated) represents CLBFT, which supports only unreplicated clients and replicated servers, and R (for replicated) represents our algorithm, which supports replicated clients and servers. Complexity is counted as the sum of ℓ times the number of application data payloads, d times the number of digests and s times the number of signatures. (Each message contains at least one signature.) The asymtotic complexity is calculated assuming s and d are constant size, but not ℓ.

	Messages		App Data		Digests		Signatures	
Step	U	R	U	R	U	R	U	R
1. client request to server	1	n	1	n	0	n	1	n
2. server pre-prepare	m	m	m	m	0	$f_c + 1$	m	$m(f_c + 2)$
3. server prepare	m^2	m^2	0	0	m^2	m^2	m^2	m^2
4. server commit	m^2	m^2	0	0	m^2	m^2	m^2	m^2
5. server to responder	0	m	0	m	0	0	0	m
6. responder to client	m	n	m	n	0	0	m	$n(f_s + 1)$
7. forward reply to store	0	n	0	n	0	0	0	n
8. store pre-prepare	0	n	0	n	0	0	0	n
9. store prepare	0	n^2	0	0	0	n^2	0	n^2
10. store commit	0	n^2	0	0	0	n^2	0	n^2

Figure 1. Message count and complexity, itemized by data type for each round

Totals	U (CLBFT)	R (Our algorithm)
Messages	$2(m^2 + m) + 1$	$2(n^2 + 2n + m^2 + m)$
App Data	$2m + 1$	$2(m + 2n)$
Digests	$2m^2$	$2(n^2 + m^2) + n + f_c + 1$
Complexity	$\ell(2m + 1) + d(2m^2) + s(2(m^2 + m) + 1)$	$\ell(2(m + 2n)) + d(2(n^2 + m^2) + n + f_c + 1) + s(2(n^2 + m^2) + m(f_c + 2) + n(f_s + 1) + 3n + m)$

Figure 2. Total Message Complexity