Trustful Cumulus Clouds

Technical Report
LPD-REPORT-2010-10
December-2010

Rachid Guerraoui Maysam Yabandeh
School of Computer and Communication Sciences,
EPFL, Switzerland
firstname.lastname@epfl.ch

Ali Shoker Jean-Paul Bahsoun
Toulouse Informatics Research Institute, I.R.I.T,
University of Toulouse
firstname.lastname@irit.fr

Abstract

Cloud computing offers an appealing business model and it is tempting for companies to delegate their IT services to the cloud. Yet, a company might find it risky to do so for sensible services and to depend entirely on a single provider, which can be vulnerable and constitute a clearly identified target for attackers. We explore in this paper a replication approach where copies of the same IT service are placed on several (cumulus) clouds that are not only independent but actually *unaware* of each other. Replica consistency is ensured using CBFT, a new BFT protocol designed for wide area networks. CBFT uses a primary to handle contention among multiple client requests but shares the load of multicasting and encrypting them among the clients. We evaluate CBFT on an Emulab cluster with a wide area topology and convey its scalability with respect to state of the art BFT protocols.

Keywords Distributed systems, fault-tolerance, cloud computing, Byzantine

1. Introduction

Cloud computing is appealing for its cost effectiveness and elasticity. In terms of reliability, the cloud might however not be a panacea. Whenever a cloud server fails, a service level agreement (SLA) stipulates the proportion of the paid fee to be returned by the cloud provider. The cheap price of cloud services, however, might imply that one could not expect much from the SLA. Amazon EC2 SLA [1] states for instance that only 10% of the paid bill (excluding one-time payments made for reserved nodes) will be returned, for longer than 5 minutes loss of external connectivity, if this occurs more than 50 times in the year [1]. The (modest) financial recompense does not even cover the server unavailability caused by attacks, e.g., denial of service (DoS), nor by failures in the connecting path between a client node and the cloud. Preventing DoS attacks, for which Amazon is not viable, is still an open problem, and reportedly very hard to handle in practice. Recently a code hosting service experienced for instance more than 19 hours of downtime after a distributed DoS attack on the computing infrastructure rent from Amazon [2]. Moreover, having the computation located over a wide area network (WAN), and far from the client, increases the unavailability risks, due, for example, to an intermediary network device failure.

This state of affairs calls for the good, old *replication* technique [3, 4] to obtain dependable services out of cheap, yet unreliable, components; namely services offered by the cloud. Assuming the independence of replica failures, a BFT protocol involving $3f+1$ replicas ensures the safe progress of a service, as long as less than $f+1$ of the replicas are faulty [5, 6]. In the extreme case, to promote failure independance, the replicas should however be located in geographically distributed clouds, i.e, what we call *cumulus* clouds. Failures that prevent the access to one replica in a given region would have a small probability to affect replicas in other regions. For instance, even after a DNS attack or tier-1 router failure in a given region, the replicas in other regions should be still reachable.

Nevertheless, a geographical distribution of replicas over cumulus clouds raises new challenges such as high latency and communication variance. Current state of the art BFT protocols [6, 7, 8, 9, 10] are typically designed and optimized for short network delays in local area networks (LAN); and most of them rely on multi-cast (that is not yet supported in WANs) to enhance performance. Figure 1 depicts the performance of PBFT [6], Zyzzyva [7], and Q/U [9] with 1, 10, and 50 clients in a WAN setting (60 ms delay, 1

[1] 99.95% annual uptime of 5 minute periods is less than ∼53 times 5 minutes downtime.

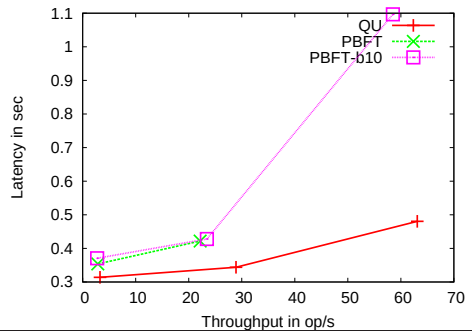

Figure 1. State of the art BFT protocols in a WAN setting.

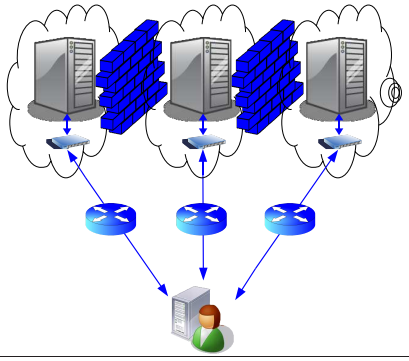

Figure 2. Cumulus clouds.

Mb bandwidth) [2]. (We faced implementation problems with Zyzzyva). The performance of PBFT drops very quickly when increasing the number of clients. Although Q/U performs much better than other state of the arts BFT protocols, because of contention between client requests, the latency quickly increases with the number of clients. In short, the limited scalability of BFT protocols in a WAN is a major barrier for adopting a replication approach over multiple clouds.

Furthermore, most of these BFT protocols involve inter-replica communication and this hampers failure independence. An attacker inside the cloud can obtain the location of all replicas after compromising or monitoring the traffic on a single replica. Locating the replicas, an attacker could focus his attacks on them. With enough resources and motivation, an attacker can at anytime orchestrate a DoS attack against a specific node, causing the unavailability of that node. Instead, an *obfuscated* BFT approach, where replicas are unaware of each other, and even if they have no idea about the replication scheme, relies on the client to increase the independence of replica failures.

However, replica communication is essential in any state machine replication protocol. Replicas can interact directly (as in traditional BFT protocols) or indirectly through a trustful mediator. The former option is not suitable for an obfuscation approach. On the contrary, the second option can have the replicas communicate through a trustful and secure mediator, i.e., the client. We do assume clients can fail by crashing but do not behave maliciously.

We argue that this assumption makes sense for applications where cloud customers are trusted members of the same organization; e.g., airline ticketing services that provide access to different agencies. In an airline ticketing system, the company hosts its service on a private/public cloud. It allows access for the ticketing agencies around the globe. The ticketing agencies access the airline service via their secured and trusted servers, which are viewed as correct clients by the airline service.

Figure 2 conveys the idea behind our approach: the clouds are oblivious to the replication procedure, driven by the client. An attacker hosted in one of the clouds cannot obtain the address of other replicas by compromising the replica inside that cloud or by monitoring its traffic. In fact, an attacker will not even know about the replication factor used by the client. The failure of a replica, hence, remains independent from the failure of other replicas, since the attacker could not locate the other replicas after compromising one of them.

This paper is a first step towards putting this approach to work. We present CBFT, a BFT protocol that offers a high independence of failures by distributing the replicas over cumulus clouds. CBFT scales to hundreds of clients due to two simple design decisions.

1. To handle contention between multiple client requests, we make use of a *primary* replica. All requests are first assigned an order by the primary before getting executed by the replicas. The requests in all replicas are thus executed in the same order regardless of the message arrival reordering.
2. A high load of client requests on the primary implies a large number of clients issuing them, which means more potential processing units. Pushing the load of encrypting and multi-casting a request from the replicas to the issuing clients, therefore, enables the system to efficiently benefit from the client processing powers.

The load on the primary in CBFT is minimal: it receives the request from the client and returns the reply. The client is then responsible for encrypting and sending the request alongside the assigned order (by the primary) to all other replicas.

In order to enhance performance, CBFT uses speculation, which however requires a recovery phase to handle failures. We address this challenge using the recently proposed notion of *abortability* [12]: Replicas in CBFT belong either to the *Active* set or to the *Passive* set. The protocol *speculative* phase operates on the Active set. Upon failure detection, the client eliminates the *suspicious* replicas from the *Active* set, and replaces them with correct ones from the *Passive* set. The protocol then resumes progress on the updated Active set.

[2] The experiments are performed in bftsim [11].

We experimented CBFT on Emulab [13]. On such system, CBFT scales to hundreds of clients, and its peak throughput doubles that of the state of the art Q/U protocol, its closest competitor.

The rest of the paper is organized as follows. The background is recalled in Section 2. Section 3 presents CBFT. After presenting the evaluation results in Section 4, we discuss some related work in Section 5, and then we conclude the paper in Section 6.

2. Background

BFT protocols are replication-based solutions to the problem of tolerating arbitrary failures of software and hardware components. A BFT protocol can ensure safety and progress up to a subset of one third of faulty replicas. Some BFT protocols requires more replicas. For example, if the application is replicated on four separate machines, then the BFT protocol can tolerate at most one faulty hard disk [5].

2.1 Quorums vs. Agreement-based BFT Protocols

BFT protocols differ in the following characteristics: being agreement-based or quorum-based, the number of required phases to commit (communication rounds), response latency, and throughput. In general, there is a trade-off between latency and throughput; to have high throughput, the contention between two competing client requests must be avoided by using a primary (basically in agreement based protocols). The primary orders the client requests, and then forwards them to the other replicas. Although this offers a high throughput, the commit latency increases because of the extra phase in communicating with the primary. On the contrary, in general, quorum based protocols like Q/U [9] need no such primary; but they have to deal with the problem of contending requests.

2.2 Authentication

To tolerate malicious attacks, the messages must be authenticated via some cryptographic techniques, such as Public Key Cryptography (PKC), which authenticates a single message, and Message Authentication Code (MAC), which authenticates a single channel (and its messages). PKC could make the BFT protocols much simpler since it is verifiable even after the message is forwarded multiple times, but it is around 100 times slower than MAC. The throughput can be bounded by the number of MAC operations per request performed by the bottleneck replica; because the sender has to authenticate a message for each destination. The node that sends more messages to the other nodes, therefore, has to do more MAC operations as well.

2.3 BFT Throughput

For large message sizes, the throughput is bounded by the input/output bandwidth of the bottleneck replica, i.e., the replica that sends/receives more messages per request. An example of such a bottleneck is multi-casting the request by the primary. The multi-cast cost can be remedied in setups such as local area network that support hardware multi-cast. Nevertheless the cloud vendors might be reluctant in offering the hardware multi-cast support due to its scalability issues [14]. Moreover, to make the replica failures as independent as possible, the replicas should be located in geographically separated clouds, where hardware multi-cast is not available.

2.4 PBFT

In PBFT [6], the client sends the request to all the replicas including the primary. The primary determines a sequence number and forwards the order to other replicas. To detect the faulty primary, all the replicas broadcast the received order to ensure that all other replica has received the same order for the request. Being ensured about the primary, all the replicas broadcast the ordered request before executing it. This phase is necessary to detect the interference of two primaries, which might happen during *view change*. The client accepts the replies if they match.

2.5 Zyzzyva

In Zyzzyva [7], the client sends the request only to the primary. However, after other replicas receive the request as well as its sequence number from the primary, they immediately execute it and send the reply to the client. The client accepts the replies if they all match. Otherwise, either the primary or some of the replicas are faulty. In this case, the first correct client can detect that and demand changing the primary.

2.6 Chain

Chain [8] also uses a primary to avoid contention. All other replicas are ordered in a chain and each one forwards the request to its successor. The last replica in the chain, i.e., the tail, sends the reply back to the client. Although this technique increases the end-to-end delay, the throughput improves as the number of MAC operations by each replica is close to one, i.e. the theoretical lower bound. The key idea is to partition the replicas into two groups, where one group only verifies the client requests and the other only authenticates the reply. After detecting the failure, the whole protocol aborts and the *abort history* is used to initialize another instance of BFT protocol.

2.7 Q/U

Quorum-based BFT protocols such as Q/U [9] do not use a primary and the clients directly communicate with the replicas. Q/U requires $5f+1$ replicas to tolerate f Byzantine faults. Nevertheless, clients can only contact a *preferred quorum* (of size $4f$+1) for optimum performance. This could result in outdated histories in some replicas, which induce the cost of *synchronization* phase to the protocol. In this phase, the outdated replica requests the up-to-date history

from f+1 other replicas (to ensure that the history is not manipulated by some faulty replicas).

To tolerate Byzantine clients, Q/U requires the clients to attach to the request a hash of the latest histories of the replicas. A replica rejects the client request if the hash does not match with the current history of the replica. In this case, the replica sends an updated version of the history to the client, in case it is a correct, but outdated client. If two clients are accessing the replicas at the same time, neither of them can complete the commit and the replicas state remain inconsistent. This again induces another complex *repair* phase to make the histories consistent. In short, the repair phase first suppresses the replicas from progress. Then, the client that has noticed the inconsistency, *Copy* the history prefix that is confirmed by at least $4f$+1 replicas.

Although Q/U has the advantage of optimal communication rounds in non-contention cases, it becomes very complex as well as expensive in the case of client request contention. As a result, the protocol is not scalable to large number of clients, where the probability of contention is very high.

2.8 Abortability

The very notion of abortability [12] was introduced to simplify the complex recovery phases in the design of speculative BFT protocols. An abort-able BFT protocol can abort at any time upon request; afterwards, no client request will be serviced by the aborted protocol. A client can initiate abort if the current running instance of BFT cannot safely progress anymore (e.g., because of contention or a failure), or the performance is not satisfactory (because of a change in the workload or the communication medium). The client requests can be safely serviced via a new BFT instance. The new BFT instance is initialized with the last state of the last BFT instance.

As long as there a client requires a service, the switching process can be finished by the client. The new instance of BFT is initialized with an *abort history* that includes all the requests globally committed by the replicas in the last BFT instance. The abort history is calculated based on the returned commit histories by the replicas. The design of an abort-able BFT protocol specifies a way to obtain the abort history, considering that (i) some replicas might be faulty, and not responding; (ii) some of the responding replicas might be faulty and return invalid histories; and (iii) the histories in the replicas could be inconsistent because of contention or a Byzantine behavior.

Quorum [12] is such an abortable BFT protocol. Similar to Q/U, Quorum has the minimum latency under no contention. Although abortability has significantly simplified the design of Quorum, it suffers from the same contention problem as Q/U. In the case of contention or a Byzantine behavior, Quorum aborts.

3. Design of CBFT

3.1 Infrastructure

CBFT is designed to be deployed on cloud infrastructures: Preferably, clouds of distinct providers, different platforms, and located in geographically distinct locations around the universe. Server farms like public/private clusters, virtualization systems, or *Grids* can also be considered. Different clouds are connected via the Internet, and they are not required to identify each other. System replicas can be chosen from distinct cloud vendors seeking high independence.

3.2 Model

Our system model complies with the traditional BFT model (e.g., PBFT [6]). We assume a message-passing distributed system using a fully connected network among nodes: clients and servers. The network may (not infinitely) fail to deliver, corrupt, delay, or reorder messages. Faulty replicas and clients may either behave arbitrarily, i.e., in a different way to their designed purposes, or they just crash (*benign* faults). A strong adversary coordinates faulty replicas to compromise the replicated service. However, we assume the adversary cannot break cryptographic techniques like collision-resistant hashes, encryption, and signatures.

Safety properties need to hold in any asynchronous environment. Clients might fail by crashing, but we don't assume them to behave maliciously (They are typically part of the same organization that delegates its IT to the cloud). Liveness, however, is guaranteed only whenever the system is *eventually* synchronous; i.e., during intervals in which messages reach their correct destinations within some fixed worst case delay.

3.3 Algorithm

CBFT is a BFT protocol that minimizes fault dependency among replicas and exhibits high throughput in WANs. It is client-based and avoids any inter-replica interaction; replicas communicate through the client instead. The protocol obtains good performance using speculation: requests are executed speculatively on replicas. Cryptographic loads and requests multi-cast are pushed towards clients to reduce overloads on replicas seeking best performance. CBFT uses *Abortability* [12] techniques to recover upon failures.

CBFT, like any optimally resilient BFT protocols, requires $3f$+1 replicas to tolerate Byzantine replicas, where no more than f replicas can be Byzantine. However, using $2f+1$ replicas only at a time, it can sustain faults, but cannot ensure progress. Thus, CBFT launches the speculative phase on $2f+1$ replicas, and upon failure it recovers by replacing the faulty replicas with the remaining f. The $2f+1$ replicas are enough to collect correct *abort history*.

Active/Passive Sets. At any time, the protocol distributes the replicas over two sets: *Active* set and *Passive* set. The former is composed of $2f+1$ replicas (we call them *Active* replicas); these replicas are used in the *speculative* phase,

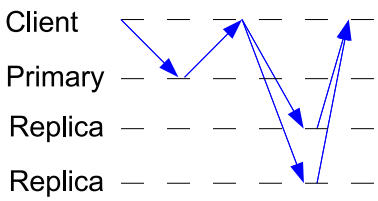

Figure 3. Message diagram of CBFT running on three replicas.

speculatively. The other f Passive replicas are used as recovery backups. Particularly, upon failure detection (i.e., in *recovery* phase), the client identifies f *Suspicious* replicas (that can include some correct replicas, and other faulty ones), and replaces them with f replicas from the Passive set. Then, the new Active set becomes correct again, and the process (the speculative phase) continues as designed.

Therefore CBFT algorithm consists of two main phases: a *speculative* phase, and a *recovery* phase. The messaging pattern of speculative phase of CBFT is depicted in Figure 3. In this section, we present the phases briefly (Details in later sections).

Speculative Phase. The communication pattern of CBFT in a failure-free scenario is simple, and it is concerned with the Active set only ($2f + 1$ replicas):

1. The client first sends its request to the primary.
2. The primary assigns a sequence number to the request, executes it, and sends a reply back to the client along with the assigned sequence number.
3. The client then sends the request together with the assigned order (previously done by the primary) to all other replicas (in Active set).
4. Each non-primary replica executes the received requests by order, and returns the results to the client.
5. A client accepts a reply only if all replica responses match; otherwise the recovery phase is launched.

Recovery Phase. This phase takes place using both Passive and Active sets.

1. When client timer expires waiting for $2f + 1$ matching replies, the client panics and sends a *Panic* message to all Active replicas.
2. At this time the client saves a list of f *Suspicious* replicas that correspond to the remainder of the first matching $f + 1$ replies (possibly a collection of faulty and non-faulty replicas)
3. Replicas, upon receiving the Panic messages, stop executing new requests and send an *Abort* message back to client with their signed *local histories*.
4. Client constructs an *Abort history* collected from the matching replies (more details later).
5. Client replaces the *Suspicious* replicas in the Active set with the f Passive replicas.
6. The updated Active set becomes correct again, and the collected abort history is used to initialize the new replicas' local histories.

3.4 Algorithm Details

We describe here the algorithm. (Because of space limitations, some details and proofs are given in the companion technical report [15]).

Client Role. To mitigate fault dependencies; CBFT clients enroll important tasks. First, the client issues the request towards the primary that assigns a unique sequence number. This is crucial to maintain consistency among different replicas. When the client receives the assigned reply from the primary, it validates its contents by verifying the MAC. The client takes the grip again to resend the signed request to the other $2f$ Active replicas, however this time, accompanied with the sequence number. At that instant the client starts a timer, waiting for replies.

The final decision is also taken by the client. Upon receiving all replies from all replicas before the timer expires; the client verifies their MACs and makes sure the replies contents (i.e., the results) are matching. If so, the client considers the request as complete; otherwise, the client launches a *recovery* phase by collecting an abort history, cleaning the Active set from *suspicious* replicas, and switching to it again when updated.

A Light Primary. CBFT pushes multi-cast and MAC overload towards its clients. In contrast to traditional BFT protocols, the primary in CBFT has almost the same load as other replicas. The only additional task is assigning an order to the requests, which requires very simple computations. On the other hand, the primary is deprived from any multi-casting duties that can transform the primary to a bottleneck, especially that individual MACs should be computed to every replica. Instead, multi-cast is done by the client that orchestrates communication among replicas as mentioned before.

Replicas. Excluding the primary, all replicas validate the client request upon its receipt. They must verify requests (with MACs, and sequence number) and then try to execute them. Replicas discard a request r_{new} in case $o(r_{last}) > o(r_{new})$; where r_{new} and r_{last} are the assigned order of current request and that of the last executed request in the replica local history, respectively. Each replica executes the request r_{new} if it has already executed all requests r_j where $o(r_j) < o(r_{new})$. Otherwise, request r_{new} is en-queued in a buffer, waiting for the missing requests that fill the gap. Final replies are authenticated via MACs and are sent by all replicas directly to the client.

Fault Independence. In spite of the diversity in the infrastructure and platforms of different clouds being used at once; the obfuscation aspects requires replicas to be unknown and

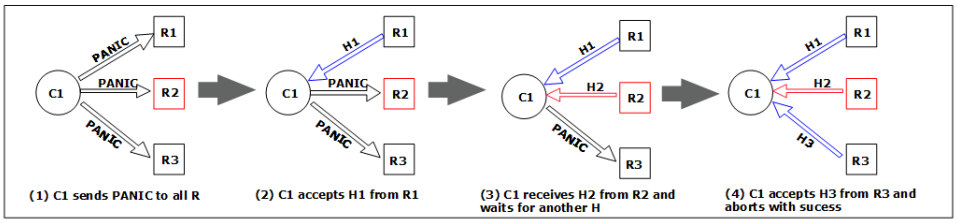

Figure 4. An example of aborting, where $f = 1$.

unaware of each other. Replicas usually, need to communicate for two purposes: to ensure a total order (atomic) request execution, and to validate response correctness (usually done by the primary). Thus, replicas in CBFT communicate with each other, however *blindly*, but guided by a third party mediator, the client. As depicted in previous sections, CBFT makes a unique total ordering possible through the client (that delivers the sequence number to all replicas), though replicas cannot directly communicate. The other issue; i.e., validating a response, is also performed by the client upon matching all replica responses. Notice that assuming no malicious clients is essential for this fault independence to hold.

3.5 Recovery Phase

The *Recovery* phase is composed of three major steps: aborting, collecting abort history, and cleaning *Active* set from *suspicious* replicas.

Aborting. A client in CBFT considers a request as complete if the received $2f + 1$ Active replica responses are matching, before the expiry of the timer. Otherwise, the client stops sending requests and sends a *Panic* message to all replicas. Each replica, upon receiving the Panic message, stops executing requests, signs an *Abort* message from its *local history* of committed requests, and sends it to the client. The latter waits until it receives a sufficient number of signed Abort messages, i.e., the first $f+1$ non-conflicting ones. The intuition is that it is necessary and sufficient for the number of received correct commit histories to exceed the number of faulty ones; knowing that faulty replicas might not respond at all. Aborting is achieved as follows:

1. The client waits for the first f+1 local commit histories to be received.
2. If no conflicting entries among the $f + 1$ received local histories found by the client, it stops receiving new histories, and collects the $f + 1$ messages in a $Proof_{AH}$ set. That is used to form the abort history AH later.
3. Otherwise if the client identified commit histories with conflicting entries it waits for new local histories (since definitely there are correct clients that did not respond yet). The loop continues from step 1 again.

Figure 4 presents an example where the first two histories returned from replicas R_1 and R_2 are conflicting, consequently the client has to wait for the history from replica R_3. Thus, the phase continued till f+1 non-conflicting local commit histories are received by the client.

Building Abort History. A correct abort history is crucial for safety. It preserves total ordering and consistency across different switching phases. The abort history is collected from the current Active set, to initialize replica local histories on a new correct Active set. Building the abort history AH is done by the client after receipt of $f + 1$ non-conflicting signed abort messages from different replicas, collected in the $Proof_{AH}$ structure (as revealed before). The steps can be summarized as follows:

1. The client generates history h such that $AH[j]$ equals the value that appears at position $j \geq 1$ of $f + 1$ different local histories LH_j that appear in $Proof_{AH}$.
2. If such a value does not exist for some position x, then x is the last index of h.
3. Finally, AH is the longest prefix of h in which no request appears twice (exclude duplicate entries).

The resulting abort history AH thus includes all the globally committed client requests as well as some partially committed ones; for example, if the request is received by at least f+1 replicas but not all of them. The AH is used to initialize the new Active replicas' histories (more details in [15])).

Eliminating Faulty Replicas. A client in CBFT attempts to replace faulty replicas in the Active set with correct ones from the Passive set; thus detecting the faulty replicas is mandatory for progress. As previously explained, upon the client timer expiry; if no $2f + 1$ matching replies were received from Active replicas; the client follows a safe process to eliminate the faulty replicas by categorizing replies senders as correct, or *suspicious* (these might be Byzantine or not). Since f maximum faulty replicas are assumed, then $f + 1$ out of the received messages should correspond to correct replicas (the matching ones). Therefore the client, to be on the safe side, considers the remaining Active replicas *suspicious*, and replaces them with the f Passive replicas.

The Active set thus comprises $2f+1$ correct replicas. The client initializes them with the abort history AH to restore system state, and the protocol continues as designed.

3.6 Checkpointing and State Transfer

Since CBFT launches a recovery phase upon fault detection, and this phase requires executing *abort history*, then switch-

ing from Active set to another (it is actually the same Active set, but cleared out from faulty replicas) will be expensive if the abort history is large. Consequently, this leads us to minimize the local history size. Thus we design a *Lightweight* checkpointing system that truncates local histories every k steps (k can be 128, 256,...).

In addition, since requests are executed on the Active set only (under normal conditions), then the Passive replicas will have old state versions, especially systems that are assumed reliable for speculative BFT protocols; in this case the abort history becomes enormously large. For that, we build our checkpointing system to keep Passive replicas states up-to-date (except for the last non-checkpointed offset requests that are fewer than k). Therefore, the Lightweight checkpointing protocol proceeds as follows (more details in [15]):

1. The client requires checkpoints from Active replicas periodically by sending R_{CHK} (for example, when the sequence number mod k becomes zero)
2. Each replica i sends its last checkpoint digest CHK_i after truncating k requests (one replica should send complete requests).
3. The client sends the received messages/digests CHK to the Passive replicas.
4. Passive replicas executes their missing requests found in CHK, and replies with ACK to the client.
5. When the client receives f ACK messages, it sends a commit COM request to all (Active and Passive) replicas for final commit.

As presented above, the checkpoints are initiated and controlled by the client. This is crucial to respect obfuscation; and here again, replicas are communicating indirectly through the client.

4. Evaluation

4.1 Overview

CBFT acquires its robustness and performance through various characteristics: (1) it needs only $3f$+1 replicas to tolerate f arbitrary faults (though speculative case communication is done on $2f+1$ Active replicas only at a time). (2) It relies on clients to multi-cast request and not on replicas. (3) It orders requests using a primary that handles consistency. Finally, (4) CBFT can be efficiently setup on *cumulus clouds*, scattered on distant locations and maybe from different vendors, to ensure higher levels of failure independence.

The closest competitor to CBFT, i.e., Q/U [9], requires at least $5f$+1 to tolerate f Byzantine faults. Additional fees shall be paid with larger f. Despite the use of preferred quorums (of size $4f$+1), Q/U provides less throughput than CBFT. This also makes Q/U more susceptible to failures under which its performance drops dramatically. This becomes lucid when the number of clients increases; partly because Q/U does not make use of a primary to order request as CBFT does. In addition, although Q/U is client-based, it enforces inter-replica interaction upon failures, which makes replica failures more dependent. Note that, as mentioned above, Q/U can overcome this by not using preferred quorums, reducing however its performance further. Regarding *fault scalability*, Q/U exhibited [9] great performance over agreement-based protocols; we expect Q/U to dominate CBFT in this sense, though we did not experiment our protocol for $f > 1$.

Quorum [8] also shares some aspects with Q/U; mostly since it is client-based and involves only two communication phases. However, Quorum also suffers from interference under contention; this makes it hard to deploy on reliable contended services. Note that we faced some difficulties in experimenting Quorum on our environment because of the use of multi-cast (especially for large number of clients). Therefore we do not provide results for Quorum except with a micro-benchmark.

On the other hand, we do not pretend perfection in CBFT. The protocol is inherently speculative and outperform others only in best cases, i.e., when there are no faults. Under failure the protocol should abort to another Active set, and this imposes additional costs represented by switching delays (Section 4.8).

4.2 Experimental Setting

CBFT experiments are performed on 23 64-bit Xeon machines with 2 GB of memory employed on Emulab [13] cluster. No virtualization is used, thus simulating cloud environment on real machines. Each replica runs on a separate machine, and the clients are scattered over 20 machines. All machines are connected using a star topology. The maximum bandwidth of the network is set to 100Mb. We set it to this high value because each machine can host up to 10 clients; and as we will see in the experimental results, the actual bandwidth that a client can use over a WAN is far below this limit. The end-to-end (E2E) delay is set to 20 ms and 60 ms, depending on the setup.

For each setting, we have run four *a/b benchmark*[3] (same benchmark used in PBFT [6]) experiments using different payload sizes: 0/0, 0/1, 1/0, and 1/1. Without a payload, the size of request and reply messages are 66 and 88 bytes, respectively. The fault factor, f, is equal to one, and the number of replicas is three.

Q/U experiments are also done on the same environment; except for a single difference where we needed six replicas ($5f$+1; for $f = 1$) instead of three, as this is the number required by Q/U [9] to operate.

4.3 Benchmark

We present here the results on a benchmark, where only one client is accessing the replicated service. Figure 5 displays

[3] In a/b benchmarks, *a* and *b* correspond to request size, and response size in KB, respectively.

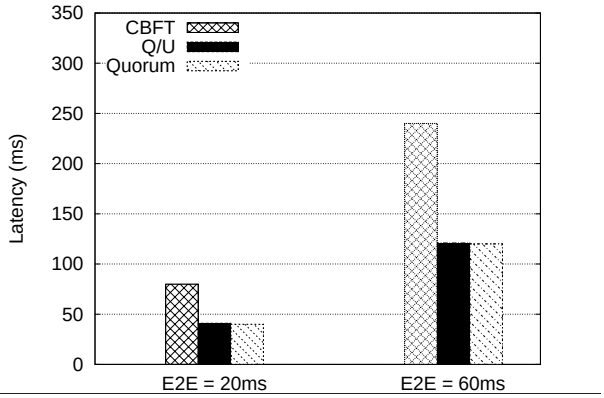

Figure 5. Commit latency when only one client is used.

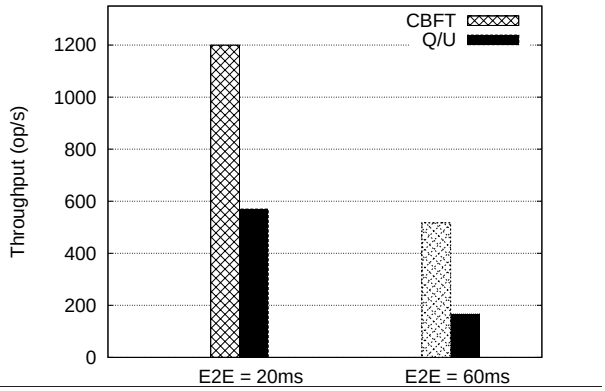

Figure 6. Peak throughput of CBFT vs Q/U for both E2E latencies: 20 ms, and 60ms.

the latency results for both CBFT and Q/U by setting the end-to-end (E2E) latency to 20 ms, and 60 ms. We do not plot the results for all benchmarks (0/0, 1/0, 0/1, 1/1) as these are very close; thus we only choose one of them (the one with minimal latency).

When the E2E latency is set to 20 ms, CBFT achieves a latency of 80 ms; Q/U on the other hand reaches half this latency as depicted in Figure 5. We relate this difference to the number of communication round-trips needed to complete an operation. In fact, CBFT needs a couple of round-trip messages; one message is sent to the primary to establish request ordering, and another is sent to communicate with other replicas and execute the request. Q/U, however, achieves this latency since it completes the operation in a single round-trip instead of two.

Again, since Quorum (like Q/U) needs only two communication phases to commit a request in a speculative way, they share same performance in a contention-free environment, the results are shown clearly in Figure 5.

Similar results are obtained upon changing the E2E latency to 60 ms. As shown on the same graph in Figure 5, the latency of CBFT becomes 240 ms. This was expected because the large E2E latency becomes the main impacting factor in the service. The graph also conveys the fact that Q/U again achieves half this latency. As mentioned above, this can be demonstrated by the number of round-trip messages in the protocols.

Analyzing the above numbers, we notice that the latency can be obtained by the number of round-trips needed for one request multiplied by the E2E latency. This means that the system delay is the major factor overhead in the communication; the operation execution and MAC handling times are almost negligible as compared to the E2E latency. Notice that, although Q/U client needs to contact at least 5 replicas (i.e., the preferred quorum [9]), this does not impact the latency as one might expect, and hence keeps Q/U leading CBFT in such experiments.

The throughput with the micro-benchmarks again shows that Q/U outperforms CBFT by almost a double. The throughput of Q/U is 8 op/s when the E2E latency is 60 ms, whereas the throughput of CBFT is 4 op/s. The ratio is almost the same in an experiment with 20 ms E2E latency. These throughput results follow from the above latency difference. Since there is only one client operating, and the client does not invoke a new request until it completes the previous one, the latency will be inversely proportional to the throughput. We do not plot a graph for the throughput since this has less importance than latency in contention-free experiments.

4.4 Peak Throughput

To experiment the peak throughput, we used up to 200 concurrent clients. Again since the results are close in different benchmarks (i.e., 0/0, 1/0, 0/1, 1/1), we mention the results for only one experiment. The throughput achieved by CBFT is very interesting and inverts the leadership with Q/U which it exhibits in contention-free cases. As depicted in Figure 6, our protocol achieves a peak throughput of 1300 op/s upon setting the E2E latency to 20 ms.

Q/U, however, could not exceed 570 op/s throughput when the E2E latency is equal to 20 ms (Figure 6). These results are justified since; (1) CBFT relies on the primary to order requests and thus avoids request collisions while accessing replicas; and (2) it pushes multi-cast and encryption overhead towards clients. On the contrary, Q/U is not resilient to a high number of clients and this forces the protocol to load excessive *Repair* and *Sync* phases, and the client *backoff* scheme. Note that this throughput is reached for 80 clients in Q/U, which is the maximum value we could get in the experiments.

By setting the E2E latency to 60 ms, as expected, the throughput drops to 520 op/s and 167 op/s in CBFT and Q/U, respectively. This change is logical since both protocols are affected by message round-trip delays. However, can be noticed in Figure 6, the throughput ratio of CBFT over Q/U changes from 1/2 to 1/3 upon updating the E2E latency from 20 ms to 60 ms. This might be explained by the extra

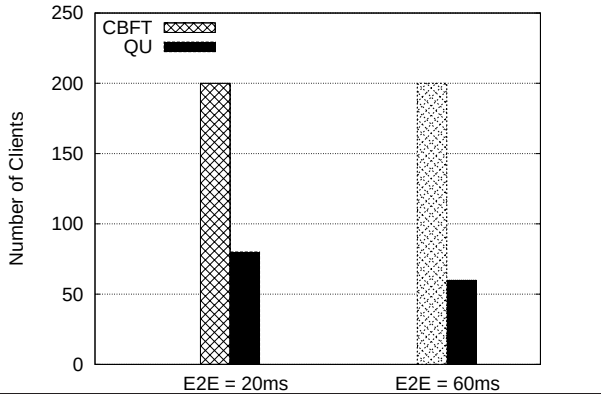

Figure 7. Client scalability of CBFT vs Q/U for both E2E latencies: 20 ms, and 60ms.

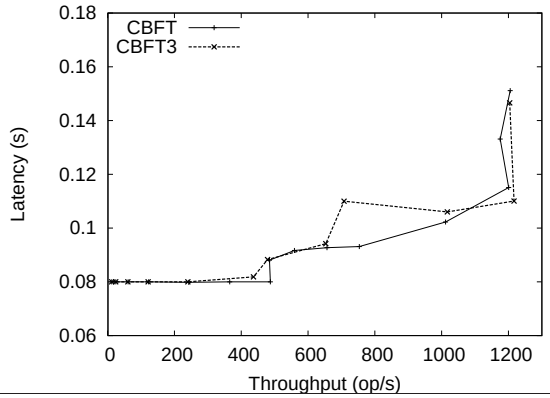

Figure 8. Performance of CBFT vs. CBFT3 for E2E latency 20 ms.

transmissions needed by Q/U because of increasing update failures.

4.5 Scalability

Yet, in another measure, i.e., client scalability, CBFT dominates Q/U (Figure 7). In the experiments, the results of Q/U started to fluctuate for more than 40 clients. Then, between 40 and 60 clients at least one run (out of four) was failing. The protocol ceased to work for a number of clients greater than 60 or 80, depending on the experiment.

However, CBFT experiments finished successfully even with 200 simultaneous clients. CBFT can handle this high number of clients since it avoids requests collisions by having a primary replica that only assigns sequence numbers (a simple operation), and by distributing the load of multicasting on clients to avoid replica bottlenecks.

By observing Figure 7, we notice that the CBFT scalability is not affected by the E2E latency; however Q/U scales for 80 clients when E2E latency is 20 ms, and could not tolerate more than 60 clients by setting it to 60 ms. This can be justified by the timeouts caused by *Repair* phase delays that Q/U launches more frequently as the number of clients grows up.

4.6 CBFT3 Observation

To explain the good performance of our client-based approach for CBFT, we run a comparison with *CBFT3*, another version of CBFT that uses $3f + 1$ Active replicas instead of $2f + 1$. Interestingly, we observed very close performance.

Figure 8 conveys this comparison. For clarity, we only plot 0/0 benchmark, given that the other benchmark results are almost the same. The results depict the very facts that the protocols performances rarely diverge starting from one client and until 200 clients. This stands as an experimental support for our claim that CBFT design features (and not the number of Active replicas) play the main role in achieving its performance.

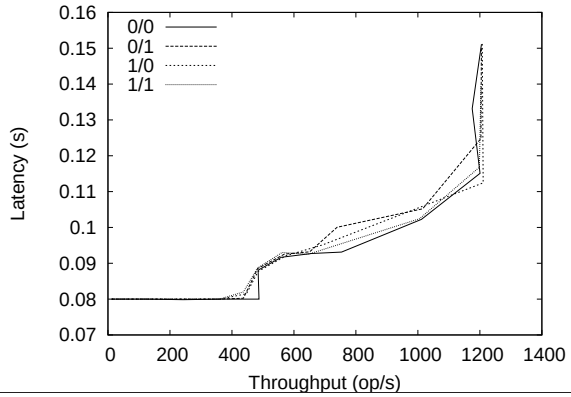

Figure 9. Response time vs. Throughput of CBFT. The E2E latency is 20ms.

4.7 Performance Traces

Figure 9 presents the evaluation results of CBFT in the setting where the E2E latency is 20 ms. The x and y axes represent the throughput and the response time for each run, respectively. The number of clients varies between 0 and 200 in the runs with 10 clients added per step. As depicted in the figure, the change in the payload size has a negligible impact on the performance since the bottleneck is caused by the network latency and not its bandwidth. The latency remains 80 ms up to the point where the throughput exceeds 400- 450 op/s. After this point, by increasing the throughput, the latency also increases up to 30% at a throughput point of 1200 op/s (160 clients). Starting from this point, increasing the number of clients leads to an exponential increase in the commit-latency.

Figure 10 conveys the results of the same experiment when setting the E2E latency to 60 ms. Compared to Figure 9 the performance drops earlier, when the throughput reaches 170 op/s. After this point, the response latency increases linearly up to 40%, as the number of clients grows.

As already mentioned, the closest competitors to our protocol are Q/U [9] and Quorum [8]. We performed the

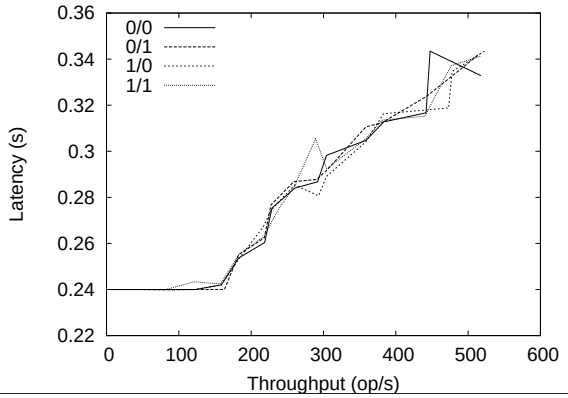

Figure 10. Response time vs. Throughput of CBFT. The E2E latency is 60ms.

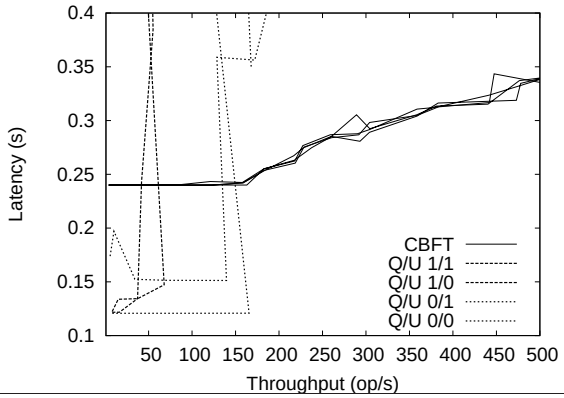

Figure 12. Response time vs. Throughput of Q/U and CBFT. The E2E latency is 60ms.

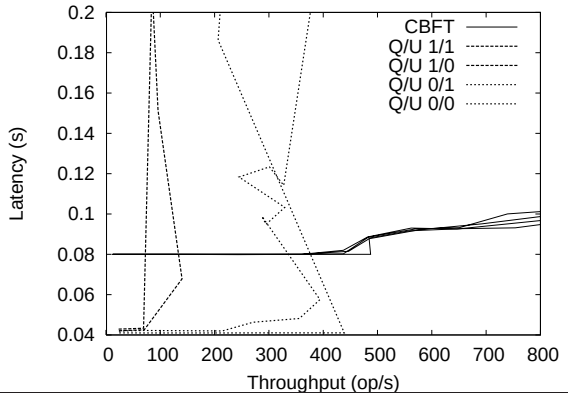

Figure 11. Response time vs. Throughput of Q/U and CBFT. The E2E latency is 20ms.

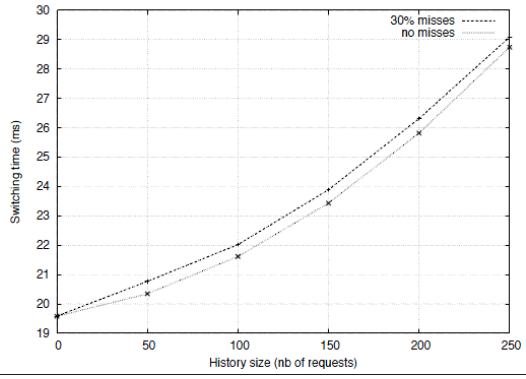

Figure 13. CBFT switching cost as a function of abort history size, and missing requests in local history.

same experiments as CBFT with Quorum. Although we have shown above the response latency was lower: 40 ms and 120 ms for the settings with an E2E latency of 20 ms and 60 ms, respectively; however we could not have more results with more clients because of multi-cast issues in the environment.

Q/U on the other hand, and despite the number of replicas required ($5f$+1) to tolerate f faults, dominates our protocol for a few number of clients; however, CBFT outperforms Q/U for more clients, where the latter's latency increases quickly. Q/U also provided little scalability in the number of clients as compared to CBFT.

We plot both Q/U and CBFT results (of E2E latency 20 ms) on the same graph in Figure 11. As can be noticed, Q/U performs differently for benchmarks 1/1 and 1/0 in contrast with 0/0 and 0/1. Q/U exhibits a constant latency of 40 ms until reaching a throughput of 80-140 op/s (depends on the experiment), where, its latency grows to higher than 200 ms quickly; this is achieved when the number of clients ranges between 20 and 30.

The other two benchmarks 0/0 and 0/1 in Q/U (also Figure 11), however, maintain a 40 ms latency until reaching the 400-450 op/s throughput; again where 20 to 30 clients are running. This was expected because Q/U needs a single communication phase to commit. Soon, this gets changed significantly under contention since more phases are needed like *Repair* and *Sync*, in addition to the *backoff* scheme applied on contending clients. However, notice on the same graph that the latency of CBFT remains almost stable (80-100 ms) even after reaching high throughput levels. Our protocol is superior to Q/U when the client count advances.

Setting the E2E latency to 60 ms shows very similar results to what has been conveyed, but of course with different numbers. Thus, we considered it enough to keep Figure 12 for the reader to analyze without including any comments. We just note that, we have no explanation to the Q/U apex it reaches, at the very beginning.

4.8 Switching Cost

As shown above, CBFT achieves a good performance in free-failure environment. However, under failures, the recovery phase should be launched. The main expensive steps in this phase are: aborting, collecting abort history, and switching to a new correct Active set that includes initializing replicas with the new abort history.

Figure 13 shows the cost of this recovery phase as a function of an abort history size for $f = 1$. We assume that the history size can grow up to 250 requests (of size 1KB). We plot two different curves: one corresponds to the case when replicas do not miss any request, i.e., all abort history requests are already executed. The other one corresponds to the case when one replica misses 30% of the abort history requests (they are not executed yet); this replica might represent the Passive set. The figure shows that the switching cost increases with the history size and that it is slightly higher in the case when replicas miss requests. More precisely, the switching cost ranges between 20ms and 30ms as the abort history size varies from 0 to 256. We consider this cost to be very reasonable, provided that faults are supposed to be rare in environments that run speculative protocols' like CBFT.

5. Related Work

5.1 Obfuscated BFT

CBFT falls into the category of *obfuscated* BFT protocols, where replicas do only see the client, thereby increasing the independence to failures. To the best of our knowledge, Quorum [8] is the only BFT protocol that also belongs to this category. However, Quorum can operate with only a few clients, while CBFT scales to hundreds of clients. Although Q/U [9] does not have any inter-replica communication in usual scenarios, after an inconsistency is caused by contention or a faulty node, the replicas have to re-synchronize.

5.2 Client-based BFT

One key to the scalability of CBFT is to push the load of agreement onto the client side. Quorum-based protocols, such as Q/U and Quorum, benefit from the same design principle. However, these notoriously do not scale with the number of clients. This is mainly because they are vulnerable to contention between multiple client requests, which makes the state of the replicas inconsistent. Upon detecting an inconsistency, the protocol has to call a recovery procedure to synchronize the replica states. CBFT benefits from a primary replica to handle contention and is hence scalable to hundreds of clients. The drawback, however, is the extra latency of communicating with the primary under contention.

5.3 Primary-based BFT

CBFT makes use of a primary replica to avoid contention between multiple client requests. This has been used in many BFT protocols such as PBFT [6] and Zyzzyva [7]. The primary, however, mostly becomes the scalability bottleneck as it performs more cryptographic operations as well as message transmissions. In particular, the lack of hardware multi-cast support in WAN puts a high load on the primary to multi-cast its messages. To address the scalability issue, CBFT pushes the multi-cast load on the client side. More clients issuing the requests implies more processing units available on the client side, and using these enables the system to handle a large volume of requests.

5.4 Abortable BFT

Admitting that a single BFT protocol cannot fit all requirements, the notion of abortability [8] has recently been proposed to enable switching between BFT protocols whenever one could perform better, e.g., because of a change in the operating environment. We make use of this abortability notion to switch between *Quorum* and *Chain*, when the load on the system changes. Furthermore, CBFT leverages abortability by switching between Active sets: after a failure is detected, the protocol replaces the suspicious replicas by correct ones from the Passive set, and uses the updated Active set again.

6. Conclusion

This paper explores an approach where reliable IT services are built on top of unreliable, yet cost-effective, *cumulus* clouds, i.e., geographically distributed clouds. Replicas of the service, located on the cumulus clouds, are oblivious of each other, as well as of the replication setting. Distributing the replicas over a WAN (with a high latency and variance) raises new challenges in the design of a scalable, high-throughput BFT protocol.

As a first step putting this approach to work, we introduce CBFT, a BFT protocol that takes up the challenges of scaling agreement over a WAN. Two simple design decisions are behind CBFT performance. First, we make use of a *primary* replica, avoiding contention between multiple client requests, to assign a sequence number to every request. Second, we push the load of encrypting and multi-casting a request from the replicas, which are the bottleneck of agreement, to the issuing clients. Our experimental results shows that CBFT scales to hundreds of clients in a WAN, while the throughput of state-of-the-art BFT protocols quickly drops with the number of clients. CBFT could tolerate both faulty replicas and clients by using $3f+1$ replicas.

With few clients, the latency of CBFT, however, is higher than that of client-based protocols such as Q/U and Quorum [8, 9]. By applying the notion of abortability [8] in the design of CBFT, and in contention-free environments, we can switch to Quorum-like protocols, provided these do not use multi-cast and achieve low-latency.

References

[1] Amazon.com, "Amazon ec2," 2010. [Online]. Available: http://aws.amazon.com/ec2/

[2] C. Metz, "Ddos attack rains down on amazon cloud," 2009. [Online]. Available: http://www.theregister.co.uk/2009/10/05/amazon_bitbucket_outage/

[3] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Comput. Surv.*, vol. 22, no. 4, pp. 299–319, 1990.

[4] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[5] G. Bracha and S. Toueg, "Asynchronous consensus and broadcast protocols," *Journal of the ACM (JACM)*, vol. 32, no. 4, pp. 824–840, 1985.

[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[7] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 45–58, 2007.

[8] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocols," in *EuroSys '10: Proceedings of the 5th European conference on Computer systems*. New York, NY, USA: ACM, 2010, pp. 363–376.

[9] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable byzantine fault-tolerant services," *SIGOPS Oper. Syst. Rev.*, vol. 39, no. 5, pp. 59–74, 2005.

[10] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "Hq replication: a hybrid quorum protocol for byzantine fault tolerance," in *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. Berkeley, CA, USA: USENIX Association, 2006, pp. 177–190.

[11] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire," in *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2008, pp. 189–204.

[12] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 bft protocol," EPFL, Tech. Rep. LPD-REPORT-2008-008, 2008.

[13] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 255–270, 2002.

[14] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, and Y. Tock, "Dr. multicast: Rx for data center communication scalability," in *LADIS '08: Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. New York, NY, USA: ACM, 2008, pp. 1–12.

[15] R. Guerraoui, M. Yabandeh, A. Shoker, and J. Bahsoun, "Trustful cumulus clouds," EPFL, Tech. Rep. LPD-REPORT-2010-10, 2010.

A. Appendix

In this Appendix, we explain CBFT in details. We present a pseudo-code for the main functions of the protocol describing the two phases: *speculative* phase and the *recovery* phase; in both client and server sides.

A.1 Notations

We denote the set of all $(3f + 1)$ replicas by Σ. At any time $2f + 1$ replicas are in the $Active$ set, and the remaining f are in the $Passive$ set. In addition, we denote the set of suspicious replicas (all faulty replicas, and possibly some correct ones) by $Suspicious$.

A message m sent by process p to the process q and authenticated with a MAC is denoted by $\langle m \rangle_{\mu_{p,q}}$. In addition, we denote the digest of the message m by $D(m)$. All processes are assumed to own the public key of every other process.

Notations for message fields and client/replica local variables used in CBFT are shown in Figure 14. To help distinguish clients requests for the same operation o, we assume that client c calls $Invoke(req, c)$, where $req = \langle o, t_c, c \rangle$ and where t_c is a unique, monotonically increasing clients timestamp. A replica r_j executes req by appending it to LH_j.

c - client ID
P - the primary replica
t_c - local time-stamp at client c
$t_j[c]$ - the highest t_c seen by replica j
o - operation invoked by the client
LH_j - a local history at replica j
sn_j - sequence number at replica j
AH - abort history

Figure 14. Message fields and process local variables

A.2 CBFT Pseudo-code

The pseudo-code of the client (c) includes a single method $Invoke()$. This method initializes a request, authenticates it to the primary P, and sends it after adjusting a timer $Timer_1$ (Pseudo 1). When the timer expires and c has not received a reply from P yet, it launches the $Recovery$ phase. In fact, some retransmission attempts are invoked before recovery; however, we do not include this in the pseudo-code for clarity.

Before recovery, the $Suspicious$ set is formed by calling $Identify()$ function, so that, faulty replicas are excluded from the $Speculative$ phase. Pseudo 4 conveys how this takes place. In words, the client looks up for the first $f + 1$ matching replies; the corresponding replicas are considered $correct$, and the other f replica are noted as $suspicious$ (though some of them are not, but they have delayed responses). The $Suspicious$ set is then passed to the $Recover()$ method that updates the $Active$ set, and launches the $Speculative$ phase on the (updated) $Active$ set again.

Pseudo 1 $Invoke(req, c)$

1: {*Sending to Primary* P}
2: $m \leftarrow \langle REQ, req\rangle_{\mu_{c,P}}$
3: $Send(m, P)$
4: **if** $Timer_1() \neq \phi$ **then** {*Until* $Timer_1$ *expires*}
5: $Receive(r_{P,sn_{new}}, P) \wedge Verifiy(r_P)$
6: **if** $sn_{new} = NULL$ **then**
7: {*empty sequence #*}
8: $Recover(P)$
9: **end if**
10: **else** {$Timer_1$ *expiry*}
11: $Recover(P)$
12: **end if**
13: {*Sending to the rest Active replicas*}
14: **for** $i \in Active \setminus \{P\}$ **do** {*Exclude primary P*}
15: {*Create messages including sequence* sn_{new}}
16: $m \leftarrow \langle REQ, req, sn_{new}\rangle_{\mu_{c,i}}$
17: $Send(m, i)$
18: **end for**
19: **if** $Timer_2() \neq \phi$ **then** {*Until* $Timer_2$ *expires*}
20: **for** $i \in Active \setminus \{P\}$ **do**
21: $Receive(r_i, i) \wedge Verifiy(r_i)$
22: **end for**
23: **else** {$Timer_2$ *expiry*}
24: $R \leftarrow \cup\{received\, r_k\ \forall k \in Active\}$
25: $Suspicious \leftarrow Identify(R)$
26: $Recover(Suspicious)$
27: **end if**
28: {*Verify if all responses are matching*}
29: $R \leftarrow \cup\{r_k\ \forall k \in Active\}$
30: {*Cardinality check*}
31: **if** $\|Matching(R)\| \neq 2f + 1$ **then**
32: {*Identify* $suspicious$ *replicas*}
33: $Suspicious \leftarrow Identify(R)$
34: $Recover(Suspicious)$
35: **end if**

Pseudo 2 $Verify(r_i)$

1: {*Verify response validity*}
2: **if** $MAC(r_i)$ is $True$ and $r_i.t_c = req.t_c$ **then**
3: **return** $True$
4: **else**
5: **return** $False$
6: **end if**

Pseudo 3 $Recover(X)$

1: $Proof \leftarrow Panic(req, c)$
2: $AH \leftarrow AbortHistory(Proof)$
3: **if** $X = P$ **then** {*Primary is faulty*}
4: $Active \leftarrow Active \setminus \{P\}$
5: $Mal \leftarrow P$
6: $I \leftarrow Passive.pop()$
7: $P \leftarrow I$ {*New Primary*}
8: $Active \leftarrow Active \cup \{P\}$
9: $Passive.push(Mal)$
10: **else** {*X is a Suspicious list*}
11: $Active \leftarrow Active \cup Passive$
12: $Active \leftarrow Active \setminus X$
13: $Passive \leftarrow X$
14: **end if**
15: $m \leftarrow \langle INIT, req, AH, Proof_{AH}\rangle_{\mu_{c,P}}$
16: $RE - INVOKE(m, AH)$

Pseudo 4 $Identify(X)$

1: {*Return* $Suspicious$ *set*}
2: **for** $i \in X$ and $\|Correct\| < f + 1$ **do**
3: **if** $\|r_i.reply = r_j.reply\| \geq f+1; \forall r_j \in X$ **then**
4: $Correct \leftarrow Correct \cup \{i\}$
5: **end if**
6: **end for**
7: $Suspicious \leftarrow Active \setminus Correct$
8: **return** $Suspicious$

Pseudo 5 $AbortHistory(Proof)$

1: {*Build abort history*}
2: $h \leftarrow$ {any $LH \in Proof$}
3: **for all** $i : 1 \rightarrow h.size()$ **do**
4: **for all** $LH \in Proof$ **do**
5: **if** $\|h[i] = LH[i]\| \geq f + 1$ **then**
6: $H[i] \leftarrow h[i]$
7: **end if**
8: **end for**
9: **end for**
10: {*Remove duplicates*}
11: **for all** $i : 1 \rightarrow H.size()$ **do**
12: **for all** $j : i \rightarrow H.size()$ **do**
13: **if** $H[i] \neq H[j]$ **then**
14: $AH \leftarrow H[i]$
15: **else**
16: **return** AH
17: **end if**
18: **end for**
19: **end for**
20: **return** AH

Pseudo 6 $Panic(req, c)$

1: {*Send PANIC to all Active replicas*}
2: **for** $i \in Active$ **do**
3: $m \leftarrow \langle PANIC, req\rangle_{\mu_{c,i}}$
4: $Send(m, i)$
5: **end for**
6: **while** $\|Proof\| < f + 1$ and $i \in Active$ **do** {*Until collecting* $f + 1$ *matching* LH}
7: $R \leftarrow Receive(r_i, i) \wedge Verifiy(r_i)$
8: **if** $r_i.type = ABORT$ **then**
9: **for all** $j \in R$ **do**
10: **for all** $k \in R$ **do**
11: **if** $\|r_j.LH = r_k.LH\| \geq f + 1$ **then**
12: $Proof \leftarrow Proof \cup \{r_j.LH\}$
13: **end if**
14: **end for**
15: **end for**
16: **end if**
17: **end while**
18: **return** $Proof$

Pseudo 7 $Server(id)$

1: **while** True **do**
2: $Receive(m_c, c)$
3: $Handle(m_c, id)$
4: **end while**

Pseudo 8 $Matching(X)$

1: {*Check if replies match*}
2: **for** $i \in X$ **do**
3: **if** $r_i.reply = r_j.reply$ and $r_i.D(LH_i) = r_j.D(LH_j)\ \forall r_j \in X$ **then**
4: $M \leftarrow M \cup \{r_i\}$
5: **end if**
6: **end for**
7: **if** $\|M\| = 2f + 1$ **then**
8: **return** $True$
9: **else**
10: **return** $False$
11: **end if**

Pseudo 9 $Handle(m_c, i)$

1: {*Handling client request*}
2: **if** $MAC(m_c)$ is $False$ or $m_c.t_c \leq t_i[c]$ **then**
3: **return** $False$
4: {*request is not valid*}
5: **end if**
6: **if** $m.type = REQ$ **then** {*Request message*}
7: **if** $i = P$ **then** {*if Primary*}
8: $sn_i \leftarrow sn_i + 1$
9: $Exec(m_c)$ {*execute request, append to* LH}
10: $r \leftarrow \langle REP, rep, D(LH_i), sn_i\rangle_{\mu_{i,c}}$
11: $Send(r, c)$
12: **else** {*not a Primary*}
13: **if** $m_c.sn = sn_i + 1$ **then**
14: $sn_i \leftarrow m_c.sn$
15: $Exec(m_c)$ {*execute request, append to* LH}
16: $r \leftarrow \langle REP, D(rep)\rangle_{\mu_{i,c}}$
17: $Send(r, c)$
18: **end if**
19: **end if**
20: **else if** $m.type = PANIC$ **then** {*Panic message*}
21: $r \leftarrow \langle ABORT, m_c.t_c, D(LH_i)\rangle_{\mu_{i,c}}$
22: $Send(r, c)$
23: **else if** $m.type = INIT$ **then** {*Init message*}
24: **if** $LH_i = \phi$ **then** {*Empty local history*}
25: $LH_i \leftarrow AH$
26: **end if**
27: $Exec(m_c)$
28: $r \leftarrow \langle REP, D(LH_i)\rangle_{\mu_{i,c}}$
29: $Send(r, c)$
30: **end if**
31: **return** $True$

Then, after collecting the local histories in $Proof$. The client builds the abort history AH by calling $AbortHitory()$ method (Pseudo 5). AH represents the matching entries of all the $f + 1$ local histories in $Proof$ (duplicates are also removed). Note that, some entries of the local history will not be included in AH. Actually, since these are not included in all the local histories, then some clients have definitely not completed the corresponding requests, and they will simply, re-invoke them again.

The recovery phase is launched by the client upon invoking $Recover()$ method (Pseudo 3). The client starts by calling the $panic()$ method; thus, it sends a $PANIC$ message to all $Active$ replicas. As depicted in Pseudo 6, the client waits for the first $f + 1$ $ABORT$ messages. Then, it checks whether the replies are non-conflicting; if they are, it waits for further replies to be received. However, when $f + 1$ of the received replies match, the client stores them (as well as the local histories LH) in a $Proof$ set (Pseudo 6; lines: 11 and 12). $Proof$ is used later to build the abort history AH. Since a maximum of f replicas can be Byzantine, the client must eventually receive $f + 1$ correct replies. In addition, $PANIC$ messages are retransmitted if the network delays are longer than expected; however, we strongly assume that this should not happen in the environments that are supposed to run speculative protocols like CBFT.

On the contrary, when c receives a reply from the primary P, it verifies its MAC and the corresponding time-stamp (Pseudo 2). The client then, prepares to send the request req to the remaining replicas in the $Active$ set ($Active \setminus P$); however this time, it appends the sequence number sn_P (already assigned by P) to req, and signs it with the MACs of the $Active \setminus P$ replicas. The client (c) then waits for $2f$ replies to be received from the replicas (thus the total count becomes $2f + 1$). Again, if the timer $Timer_2$ expired before receiving the supposed replies, recovery will be invoked (some retransmissions have been removed for clarity). Otherwise, c checks if all replies are matching (Pseudo 8) to complete the request; if they are not, the client recovers after collecting $suspicious$ replicas.

Finally, the client in Pseudo 3 removes the $Suspicious$ replicas from the $Active$ set and replaces them with the $Passive$ ones. Therefore, the active set becomes correct again, and ready to enter the $Speculative$ phase by invoking the request (after initializing the local histories of the new replicas). Note that, the special case of faulty primary is treated in a bit different way than other replicas. In fact, if the client has not received a reply from P before $Timer_1$ expiry, P is simply replaced by another replica from the $Passive$ set (i.e., no need to replace f replicas).

The pseudo-code of the server side is simple in CBFT. It is represented by receiving requests from clients and handling them (Pseudo 7). Handling requests of various types occurs differently: REQ messages are ordinary requests; after validating their MACs, and checking out the validity of their time-stamps, the request is executed (i.e, appended to LH). If the replica is a primary, it increments its sequence number sn_P and appends it to the reply. The non-primary replicas, however, need to make sure that the received request is not already executed (Pseudo 9; lines 13 and 14). In addition, if some requests sn_i with $sn_i < sn_{new}$ are not executed yet, the replicas buffer the new request until the missing (actually delayed) requests arrive (We do not convey buffering in the pseudo-code for simplicity).

Replicas handle $PANIC$ requests by ceasing to execute further requests, and then sending $ABORT$ replies that comprises their local history digests $D(LH)$ (Pseudo 7). On the other hand, $INIT$ requests are used by replicas to execute any missing requests in their local histories after recovery.

B. Correctness

B.1 Commit Certificate

Proposition. Any completed request by the client has been committed by the Active replicas.

Proof. The client in CBFT completes a request only if it has received $2f + 1$ matching responses including the local history digests $D(LH_i)$ of the Active replicas (Pseudo 9, line 31), among which $f + 1$ replicas are correct. Recall that, local histories (LH_i) are uniquely defined sequences of requests, which represent the replica state at any time. Then, since a correct replica appends the new request (upon execution) to its local history before sending the LH digest ($Exec$ method in Pseudo 9), then these digests represent an indication for the client certifying that its request has been committed successfully.

B.2 Validity

Proposition. Any request that is found in the commit/abort history must have been sent by some client.

Proof. A client commits a request only if all the received commit histories (LH) of the Active replicas are matching (Pseudo 1, line 31). Thus, at least $f+1$ correct replicas must have executed the request, and appended it to LH. On the other hand, a replica executes (i.e., appends to LH) a request message REQ (or $INIT$ message) only after validating its sender identity; that should be some client. In addition, to avoid duplicates in LH, a replica always maintains and checks the last client time-stamp $t_i[c]$ (Pseudo 9, line 2).

As for the abort history AH, since it is collected from $f + 1$ matching LH (Pseudo 5); thus, all requests in the AH are sent by some client (follows from the previous paragraph). As for AH duplicates, they are removed by construction (Pseudo 5, lines: 10 to 15).

B.3 Termination

Proposition. Aborting from the $Specultive$ phase eventually occurs.

Proof. CBFT runs the Speculative phase until: (1) the client detects non matching responses from replicas or (2) its timer expires. In both cases, the client should abort the Speculative phase by sending a $PANIC$ messages to all the $Active$ replicas (for progress, it keeps sending such messages until receiving the needed $ABORT$ messages). The replicas should eventually receive the $PANIC$ messages (according to our assumption that sent messages are not infinitely delayed or dropped by the network). Thus, at least $f + 1$ correct replicas should send $ABORT$ messages to the client (Pseudo 6, line 11). When the client (eventually) receives $f + 1$ matching $ABORT$ messages, it aborts the request (since the clients can not be malicious).

B.4 Lemma 1

Denote the state of the local history of replica r_i upon appending request req to LH_i by LH_i^{req} . Then, for any message m sent by r_i upon appending to LH_i with history LH_i^m , LH_i^{req} is a prefix of LH_i^m. In other words, LH_i^{req} remains a prefix of LH_i forever.

Proof. Let the current state LH_i of some replica r_i be LH_i^{req}. A correct replica r_i modifies its local history LH_i by sequentially appending any new request m to LH_i; in particular, appending to its prefix LH_i^{req} ($Exec$ function in Pseudo 9). Hence, $\forall$ new request m, LH_i^{req} remains a prefix of LH_i^m forever.

B.5 Commit Ordering

Proposition. Commit histories can not contain requests in conflicting orders.

Proof. Assume, by contradiction, that there are two committed requests req and $req' \neq req$ (sent by two clients c and c', respectively) with different commit histories h_{req} and $h_{req'}$, such that, neither is the prefix of the other. Since a correct client commits a request only when it receives $2f + 1$ identical LH digests from replicas (Pseudo 1; line 31); then, there must be a correct replica r_j that sent $D(h_{req})$ to c and $D(h_{req'})$ to c' such that h_{req} is not a prefix of $h_{req'}$ nor vice versa. A contradiction with Lemma 1.

B.6 Abort Ordering

Proposition. For any committed request req, every commit history h_{req} is a prefix of any abort history AH.

Proof. Considering a single replica r_i; suppose that $\exists$ a request req and $ABORT$ message m such that h_{req} is not a prefix of LH_i^m. Since req is already committed, then it must be included in $2f + 1$ local histories from the Active replicas including r_i (Pseudo 1; line 31). But r_i does not send $ABORT$ messages unless after it stops executing new requests; thus r_i executed req before m. Hence, by Lemma 1, h_{req} is a prefix of LH_i^m. On the other hand, since an abort history is constructed from $f + 1$ matching LH digests sent by correct replicas (Pseudo 5), then h_{req} must be a prefix to all these LH, i.e., to the abort history AH.

B.7 Init Ordering

Proposition. INIT history is a prefix for any commit/abort history.

Proof. Every correct process must initialize its local history with some valid $Init$ history before sending any message (Pseudo 9, line 23). Since any common prefix CP of all valid $Init$ histories is a prefix of every single $Init$ history I, thus CP is a prefix for every local history sent by a correct replica. $Init$ ordering for commit histories immediately follows.

In the case of abort histories; $f + 1$ correct $ABORT$ messages are received by a client upon aborting a request. The $ABORT$ message contains the replicas LH (Pseudo 9, line 21) that have CP as a prefix. Thus, CP is a prefix of any abort history AH.

B.8 Progress

Proposition. Clients eventually receive replies to their requests.

Proof. Recall: we assume that the network can not delay messages infinitely. Thus, we suppose any sent messages to reach its destination within a maximum delay δ. In the $Speculative$ phase of CBFT, clients wait for responses from replicas twice: (1) waiting for the primary ($Timer_1$, Pseudo 1, line 4), and then (2) waiting for the other $Active$ replicas ($Timer_2$, Pseudo 1, line 23). Adjusting $Timer_1$ and $Timer_2$ for a duration of $2\delta+t$ (i.e, the delay for a complete request round trip + the expected execution time at replicas) ensures that the client will eventually receive $2f + 1$ replies from the $Active$ replicas.

In the $Recovery$ phase, the client also waits for $f + 1$ non-conflicting $ABORT$ messages from replicas to switch (Pseudo 6). This must occur since at least $f + 1$ $Active$ replicas must be correct. A timer in this case can be adjusted to $2\delta + t$ also to ensure progress. This represents the time for the $PANIC$ messages to reach the replicas + handling time at replicas + the delay of $ABORT$ messages from the replicas to the client (however this is not shown in the pseudo-code). Upon the expiry of the timer , retransmission of $PANIC$ messages is invoked again.

Regarding the detection and replacement of $Suspicious$ replicas, some progress issues might appear if the faulty replicas are distributed among both $Active$ and $Passive$ sets. Section C addresses this issue, and provides a switching optimization to ensure liveness.

C. Switching Optimization

The concept of switching from one $Active$ set to another is possible by the replacement of f $Suspicious$ replicas with other f $Passive$ ones (Pseudo 4). These $Suspicious$ replicas can be faulty or just slow; thus, it is worth replacing them with new correct/fast replicas. This is suitable for small f (for instance, $f = 1$). However, when f is large, and the faulty replicas are distributed among the $Passive$ and

$Active$ sets; liveness problems might show up, since faulty replicas can force continuous switching. A slight optimization for switching solves this issue with paying a very little cost: Instead of electing the first $f+1$ matching replies to distinguish the $Suspicious$ replicas, we give way to the client to contact the $3f+1$ replicas in order to choose the $Suspicious$ set.

In particular, after the client receives the $ABORT$ messages from $f+1$ replicas, it sends the $INIT$ request to all $(3f+1)$ replicas instead of the $Active$ ones only. The first $2f+1$ received matching replies correspond to correct replicas, that are designated as $Active$ replicas by the client. The remaining f replicas form the $Passive$ set. Starting from that instant, the client sends its REQ messages to the $Active$ replicas as usual. By this optimization; choosing the $Active$ replicas is always done by consulting all replicas at once, and thus, obtaining a possibly different $Active$ set upon distinct $Recovery$ phases to maintain progress.

D. Lightweight Checkpointing

The objective of Lightweight checkpointing is to maintain small size local histories LH on replicas in order to reduce the communication cost, in particular, upon moving from the $Speculative$ phase to the $Recovery$ phase. Lightweight checkpointing can minimize the local history sizes (256 for instance); and thus reducing the switching time. Figure 13 conveys the switching time as a function of abort history size. Checkpointing appears more crucial as $Passive$ replicas have to be updated continuously, so that, they maintain an up-to-date state as $Active$ replicas.

Lightweight checkpointing pseudo-code is presented in Pseudo 10 and Pseudo 11. The client triggers checkpointing every k requests (we use $k=256$). It sends R_{CHK} message to $Active$ replicas (Pseudo 10, lines: 1 to 5) and starts a timer $Timer_1$. Each $Active$ replica validates the R_{CHK} (MAC and times-tamp), and sends a CHK message to the client. A CHK_i message sent by replica i contains: the digest of 256 requests starting from its last checkpoint of number n_i, and the new supposed checkpoint counter n_i+1 (Pseudo 11, lines: 10 and 13). The primary P is an exception; it sends the CHK messages with the whole LH_P (not a digest; Pseudo 11, line 8).

When the client receives $2f+1$ CHK messages with valid MACs and matching digests, it sends a R_{CHK} message to the $Passive$ replicas with the LH_c piggybacked (LH_c is equivalent to LH of the primary). Otherwise, if $Timer_1$ expired, the checkpoint is postponed to the next attempt. After validating the request, a $Passive$ replica appends LH_c to its local history (Pseudo 11, line 12), and replies to the client with a CHK message. When the client receives f such messages from the $Passive$ replicas, it sends COM message to all replicas ($Active$ and $Passive$) to commit the checkpoint (Pseudo 10, lines: 28 to 30). The replicas, then, truncate the 256 requests from their LH and increment their checkpoint counter n_i (Pseudo 11, lines: 17 and 18).

As we notice, a checkpoint does not succeed unless all the replicas responded correctly. Thus, if a checkpoint failed, replicas should truncate 2×256 requests instead of 256 in the next attempt. More generally, $i\times256$ requests are truncated if i consecutive failed attempts occurred.

Pseudo 10 $CHK_{client}()$

```
1:  {Send R_CHK to all Active replicas}
2:  for i ∈ Active do
3:      m ← ⟨R_CHK⟩_{μ_{c,i}}
4:      Send(m, i)
5:  end for
6:  if Timer_2() ≠ φ then {Until Timer_1 expires}
7:      while ‖Proof_CHK‖ < 2f + 1 and i ∈ Active do {Until collecting 2f + 1 matching CHK}
8:          Receive(r_i, i) ∧ Verifiy(r_i)
9:          if r_i.LH = j.LH and r_i.n_i = r_j.n_j; ∀j ∈ R_1 then
10:             R_1 ← R_1 ∪ {r_i}
11:         end if
12:     end while
13: end if
14: {Send R_CHK to all Passive replicas}
15: for i ∈ Passive do
16:     m ← ⟨R_CHK, LH_c⟩_{μ_{c,i}}
17:     Send(m, i)
18: end for
19: if Timer_2() ≠ φ then {Until Timer_2 expires}
20:     while ‖Proof_CHK‖ < f + 1 and i ∈ Passive do {Until collecting f matching ACK CHK}
21:         Receive(r_i, i) ∧ Verifiy(r_i)
22:         if r_i.LH = r_j.LH and r_i.n_i = r_j.n_j; ∀j ∈ R_1 ∪ R_2 then
23:             R_2 ← R_2 ∪ {r_i}
24:         end if
25:     end while
26: end if
27: {Commit checkpoint n_c}
28: for i ∈ Passive ∪ Active do
29:     m ← ⟨COM, n_i⟩_{μ_{c,i}}
30:     Send(m, i)
31: end for
```

Pseudo 11 $CHK_{replica}(m_c)$

```
1:  {Handling checkpoint requests}
2:  if MAC(m_c) is False or m_c.t_c ≤ t_i[c] then
3:      return False
4:      {request is not valid}
5:  end if
6:  if m.type = R_CHK then {R_CHK message}
7:      if i = P then {if Primary}
8:          r ← ⟨CHK, LH_i, n_i + 1⟩_{μ_{c,i}}
9:      else if i ∈ Active then {Send digest only}
10:         r ← ⟨CHK, D(LH_i), n_i + 1⟩_{μ_{c,i}}
11:     else {Passive replicas update their LH}
12:         LH_i ← m_c.LH_c
13:         r ← ⟨CHK, D(LH_i), n_i + 1⟩_{μ_{c,i}}
14:     end if
15:     Send(r, c)
16: else if m.type = COM then {Commit request}
17:     n_i ← n_i + 1
18:     Truncate(LH_i, n_i) {truncate 128 LH entries}
19: end if
20: return True
```