Stretching BFT

Rachid Guerraoui
EPFL
Lausanne, Switzerland
rachid.guerraoui@epfl.ch

Nikola Knežević
EPFL
Lausanne, Switzerland
nikola.knezevic@epfl.ch

Vivien Quéma
CNRS
Grenoble, France
vivien.quema@inrialpes.fr

Marko Vukolić
Institut Eurécom
Sophia-Antipolis, France
marko.vukolic@eurecom.fr

***Abstract*—State-of-the-art BFT protocols remain far from the maximum theoretical throughput. Based on exhaustive evaluation and monitoring of existing BFT protocols, we highlight few impediments to their scaling. These include the use of IP multicast, the presence of bottlenecks due to asymmetric replica processing, and an unbalanced network bandwidth utilization.**

To better evaluate the actual impact of these scalability impediments, we devised Ring, a new BFT protocol, which circumvents them. As its name suggests, Ring uses a ring communication topology, where, in the fault-free case, each replica only performs point-to-point communications with two other replicas, namely its neighbors on the ring. Moreover, all replicas equally accept requests from clients and perform symmetric processing.

Our experiments show that on a Fast Ethernet network Ring achieves an aggregate throughput of 118 Mbps, which is 27% higher than most efficient state-of-the-art BFT protocols.

Ring approaches but does not reach the throughput theoretical maximum. Yet, its very performance makes it possible to envision a new generation of BFT protocols that might reach the actual theoretical maximum.

***Keywords*-BFT protocol, fault-tolerance, throughput**

I. Introduction

Byzantine fault tolerance (BFT) enhances the availability and reliability of replicated services in which faulty nodes may behave arbitrarily. Many BFT protocols have been recently devised [1], [2], [3], [4], [5], [6], and their performance are usually considered acceptable, for they get close to the performance of non-replicated systems in best-case executions (i.e., synchronous executions with no failures). Arguably, these best-case execution scenarios are achieved frequently in practice.

A closer look at the performance of state-of-the-art BFT protocols reveals however that even in a best-case execution, their performance is far from the theoretical maximum. For instance, our experiments show that, when deployed on a Fast Ethernet network, the most efficient BFT protocols achieve a throughput of 93 Mbps, which is far from the theoretical maximum [7] (124 Mbps)[1].

This throughput performance issue becomes even more relevant with recent works on deterministic execution on multicore machines [8], [9], making it possible to leverage multicore architectures to achieve high CPU execution performance. Basically, the bottleneck will soon no longer be the execution speed of the replicated service, but the throughput of the agreement phase of the underlying replication protocol, hence the pressing need for BFT protocols achieving higher performance. This paper is precisely about studying whether BFT protocols can get closer to their theoretical maximum and what would prevent them from that.

In order to understand the feasibility of throughput-efficient BFT protocols, we conducted an extensive study of the most efficient state-of-the-art BFT protocols. The goal of this study was to identify the bottlenecks of current BFT protocols. Our study (detailed in Section III) reveals the following limiting factors: (1) **Asymmetric replica processing:** existing protocols do not equally balance the processing load on different replicas (some replicas perform up to 20% higher CPU processing than other replicas), (2) **Unbalanced network utilization:** existing protocols do not equally use available networking resources (some replicas do either not send or receive any data), and (3) **IP multicast packet drops:** most BFT protocols rely on IP multicast which is often inefficient in highly loaded environments as it may result in high ratios of packet drops (30% on our hardware).

To better evaluate the actual impact of these scalability impediments, we devised a new BFT protocol, called Ring, which circumvents them. Ring avoids IP multicast: it uses a point-to-point ring topology for request dissemination and ordering. Moreover, replicas in Ring are CPU symmetric, since they perform (almost) identical processing, which avoids bottlenecks. Notably, any replica can receive a client's request. Finally, there are no underutilized network link in Ring: the load is fully balanced on all available network links. This is a consequence of the ring topology and the fact that each replica sends and receives the same amount of data.

The idea of using a ring-based topology to improve the throughput of broadcasting protocols is not new: it was adopted for instance in LCR [7] and Ring Paxos [10]. However, LCR and Ring Paxos focus on crash failures. The technical difficulty in designing Ring is to tolerate Byzantine faults, while maintaining a ring-based communication

[1]The theoretical maximum for a replicated service is $\frac{n}{n-1}B$, where n is the number of replicas, and B is the maximal throughput of a single network link (93 Mbps on the Fast Ethernet network we are using, as reported by the *netperf* tool).

pattern. This is challenging in various aspects. For instance, a faulty replicas may trick correct replicas into not executing correct requests. Faulty replicas can also try to force correct replicas to execute requests that were not issued by clients or try to bypass replicas in the ring.

We evaluated Ring using the Emulab [11] testbed and compared its performance to that achieved by the three most-efficient BFT protocols, namely PBFT [1], Zyzzyva [2] and Chain [6]. Our performance evaluation shows that Ring significantly outperforms other protocols in terms of throughput (+27%), and that it achieves up to 14% lower response time than state-of-the-art protocols. Yet, we do not claim that Ring is the ultimate protocol throughput-wise, since our implementation does not reach (it only approaches) the theoretical maximum (we discuss this further in Section VII). Nevertheless, the performance of Ring makes it possible to envision a new generation of BFT protocols that would approach the theoretical maximum.

To summarize, this paper makes the following contributions:

- We analyze state-of-the-art BFT protocols under high load and pinpoint their underlying scalability impediments.
- We propose a protocol called Ring, which sustains very high throughput, and we highlight thereby the actual impacts of those impediments.

The rest of the paper is organized as follows. In Section II, we overview state-of-the-art BFT protocols. Section III presents our analysis of the bottlenecks of these protocols. Section IV contains the description of Ring. Section V contains experimental evaluation, while in Section VI we discuss related work. Finally, in Section VII we conclude this paper.

II. Background

In this section, we overview state-of-the-art BFT protocols that we later use in the evaluation. We focus on protocols known to provide high throughput: Chain [6], Zyzzyva [2], and PBFT [1]. These three protocols rely on a dedicated replica that receives requests, called the *primary* or the *head*. This replica also assigns sequence numbers to requests and forwards them to other replica. Note that all these protocols require $3f+1$ replicas to tolerate f faults (which is optimal [12]). We do not describe quorum-based protocols [3], [4][2], which are known to perform poorly under contention [13].

The communication pattern implemented in Chain [6] is depicted in Figure 1. Chain relies on two distinct replicas: the *head* and the *tail*. All replicas are arranged in a chain (hence the protocol name). A client sends a request to the *head*, which assigns a sequence number to the request. The *head* then forwards the request to the next replica in the chain. Each replica executes the request, appends it to its local history, and forwards the request until the request reaches the *tail*. Finally, the *tail* replies to the client. The last $f+1$ replicas include the digest of their history in the forwarded request, which the tail sends to the client. If these digests match, the client commits the request. Otherwise, the client resorts to a backup protocol to commit the request. We do not describe this backup protocol as it is not used in the normal case (synchronous network, no faults).

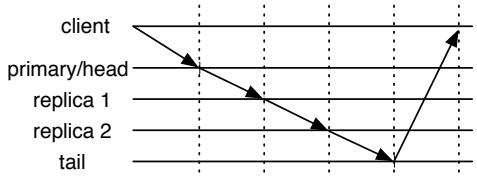

Figure 1. Communication pattern of the Chain protocol.

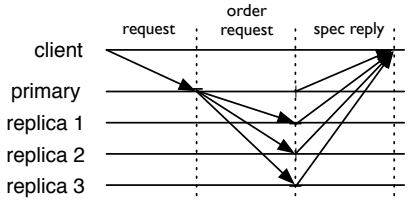

Figure 2. Communication pattern of the Zyzzyva protocol.

The communication pattern implemented in Zyzzyva [2] is depicted in Figure 2. Zyzzyva relies on a dedicated replica, called *primary*, to order requests. To issue a request, clients in Zyzzyva send it to the *primary*. The *primary* assigns a sequence number to the request and multicasts it to other replicas[3]. All replicas (including the *primary*) speculatively execute the request and reply to the client. Replicas include the digest of their history in their reply. If the client receives $3f+1$ matching replies, it commits the request. Otherwise, the protocol executes a slower path, in order to reconcile replicas. This part of the protocol is not executed in the common case (synchronous network, no faults). We do thus not describe it in this section.

Finally, the communication pattern implemented in PBFT [1] is depicted in Figure 3. Similarly to Zyzzyva, PBFT relies on a dedicated replica, called *primary* to order requests. To issue a request, a client sends it to the *primary*. The latter appends a sequence number to the request and broadcasts a *PRE-PREPARE* message to all replicas containing the ordered request. When a replica receives the *PRE-PREPARE* message, it acknowledges it by broadcasting a

[2]These protocols do not rely on a dedicated replica to order requests.

[3]Both Zyzzyva and PBFT define an optimization for large requests, which consists in having clients multicast their requests to all replicas. Nevertheless, on our hardware setup, this optimization drastically decreases performance (due to IP multicast packet drops as explained in Section III-C).

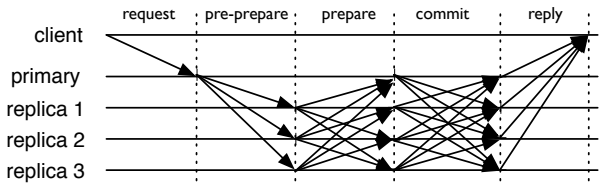

Figure 3. Communication pattern of the PBFT protocol.

PREPARE message to all other replicas. As soon as a replica receives a quorum of $2f+1$ *PREPARE* messages, it promises to commit the request (at the sequence number appended by the *primary*) by broadcasting a *COMMIT* message. When a replica receives a quorum of $2f + 1$ *COMMIT* messages, it executes the request and replies to the client. A client commits the request if it receives $f + 1$ matching replies. Otherwise, the client retransmits the request. If the request does not commit after a certain time, the protocol executes a leader election protocol to change the primary. This part of the protocol is not executed in the common case (synchronous network, no faults). We do thus not describe it in this section.

III. Obstacles towards high throughput

We benchmarked available implementations of PBFT [1], Zyzzyva [2][4], and Chain in order to understand what prevents them from achieving higher throughput with large number of clients. Although we do not claim that our list is exhaustive, we highlight main obstacles to achieving high throughput: asymmetric replica processing, unbalanced network utilization, and IP multicast packet drops.

We run the experiments on Emulab [11]. In each experiment, we used *pc3000* machines – a Dell PowerEdge 2850s systems, with a single 3 GHz Xeon processor, 2 GB of RAM, and 4 available network interfaces. Each machine runs Ubuntu 8.04, with the default kernel (2.6.24-28). Replicas are each running on a separate machine, while clients are deployed on a total of 15 machines. In all our experiments we use a topology where replicas belong to one Fast Ethernet LAN, and clients communicate with replicas over a second Fast Ethernet LAN. The reason for choosing this topology is that it yields significantly better performance, especially for Zyzzyva and PBFT. This is explained by the fact that it reduces the number of IP multicast packet drops. Finally, we use the closed-loop benchmark used to evaluate state-of-the-art BFT protocols [1], [2], [6]. In this benchmark, a set of clients are deployed and issue requests in a closed-loop manner: each client issues a new request only after it has received a reply to its current request. The benchmark allows modifying the size of the requests that are issued by clients and the size of replies that are generated by the replicas.

[4]We could actually not conduct experiments with the original Zyzzyva code base, as (1) the implementation is incomplete, and (2) there are bugs that prevent running experiments with a high input load . We did thus use our own implementation of Zyzzyva, called ZLight [6].

A. Asymmetric replica processing

As we have seen in the previous section, Chain, Zyzzyva and PBFT all rely on a dedicated replica to handle incoming client requests. We monitored the CPU load at each replica to detect whether these replicas have a higher CPU load than other replicas and are thus bottlenecks.

To monitor the CPU load, we use the benchmark described above. Requests issued by clients are 8 bytes large. We vary the number of clients to inject different levels of load. Each client sends 10'000 requests, and we measure the CPU load of the different replicas with the `sar` utility [14]. Results are depicted in Figure 4 for 40, 120, and 200 clients, respectively. In each protocol, the replica handling incoming requests (*primary* in PBFT and Zyzzyva and *head* in Chain) is replica 0.

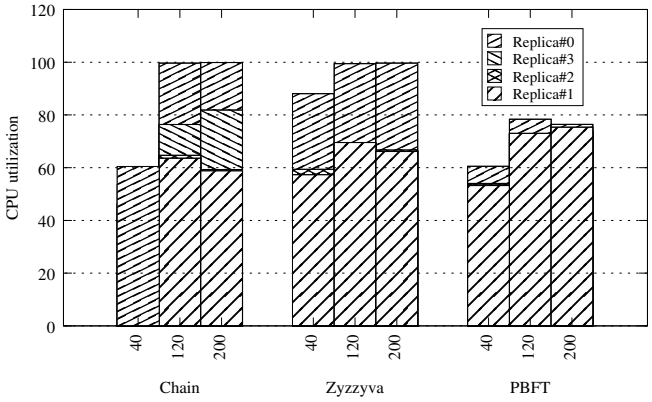

Figure 4. CPU utilization on different replicas, for different numbers of clients.

We observe that for each protocol, the replica receiving client requests has higher CPU load. The difference is quite important for Chain and Zyzzyva (about 20% higher CPU load). This can be explained by the fact that this replica receives all client requests (and manages all client connections) and that it performs more cryptographic operations than other replicas. Regarding Chain, we can also observe that the tail also performs more work than other replicas, which we explain by the fact that it sends replies to clients. Interestingly, we remark that PBFT has lower CPU consumption than other protocols and that the CPU usage increase observed at the primary is negligible. We explain this behavior by the fact that, for every received message, nodes in PBFT have 4 communication rounds involving IP multicast. Thus, replicas in PBFT spend more time sending requests, than actually processing them.

B. Unbalanced network utilization

Throughput inefficiency can also be caused by an unbalanced utilization of the available network bandwidth.

More precisely, if network links are not being used equally, some may become bottlenecks, limiting performance, while others remain underutilized. We study a setup with four replicas. As explained before, each replica has two network interfaces: one for client-to-replica communications, and one for replica-to-replica communications. We monitor the number of bytes that are sent/received by replicas for replica-to-replica communications. Requests issued by clients are 4 kB large. Figure 5 conveys the normalized amount of sent and received bytes over each link. In other words, the Figure shows how many bytes are sent (or received) for each byte received from a client. Bars *in* (resp. *out*) denote normalized amount of data on incoming (resp. outgoing) links to the replica.

We observe that every protocol exhibits unbalanced network utilization. In Chain, the incoming link of the *head* is not used. Indeed, no replica sends messages to the *head*. For similar reasons, the outgoing link of the *tail* is not used. In Zyzzyva, the *primary* only uses its outgoing link (it does not receive any message from other replicas), whereas all other replicas only use their incoming link (they do not send messages to other replicas). Finally, PBFT uses all links, but the incoming link of the *primary* and the outgoing links of all other replicas are underutilized: the slight difference with Zyzzyva stems from *PREPARE* and *COMMIT* messages.

C. IP multicast packet drops

The last source of throughput inefficiency that we considered is the usage of IP multicast. Both Zyzzyva and PBFT use IP multicast to send a message to a group of replicas. This optimization might however be hazardous to performance due to packet drops. To quantify the potential impact of IP multicast, we run a simple experiment, where a set of machines are simultaneously multicasting messages. We vary the number of machines (3, 6, 9). Each machine multicasts 4 kB packets to one machine, which only listens. We also vary the sending rate to achieve a total aggregate throughput in range 70-110 Mbps. We choose values higher than the maximum throughput on the Fast Ethernet network (100 Mbps) to model the fact that senders cannot be coordinated in Byzantine environments. Figure 6 depicts the loss rate when the sending rate of each sender increases.

We observe that the loss rate increases non-linearly when the aggregate throughput goes over the link speed. Moreover, the loss rate increases with the number of servers in the group, although the rate stays constant. For example, with 3 servers sending at 36.6 Mbps, almost every 4^{th} packet is dropped. In contrast, with 9 senders serving a total aggregate rate of 110 Mbps (each server sends only 12.2 Mbps), every 3^{rd} packet is dropped (these results are consistent with similar experiments for Gigabit Ethernet networks presented by Marandi et al. [10]).

The packet drops are explained by the fact that IP multicast is an unreliable protocol: under high contention,

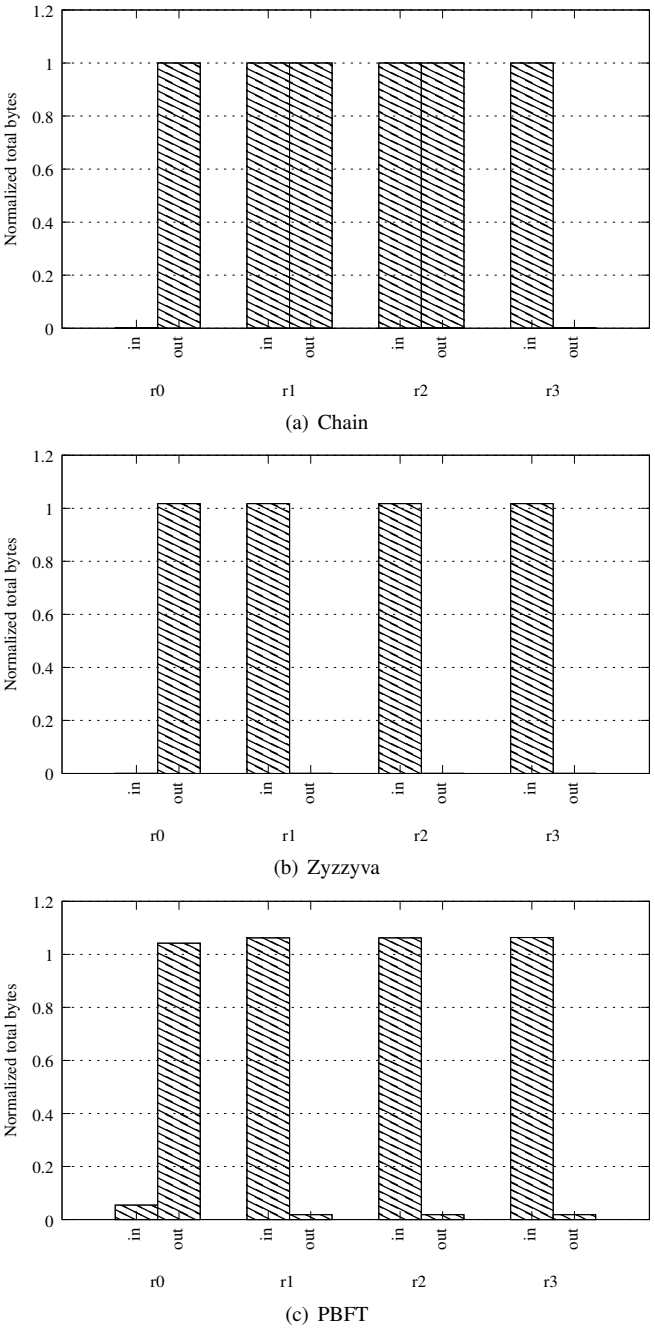

Figure 5. Network link utilization in the (a) Chain, (b) Zyzzyva, and (c) PBFT protocols.

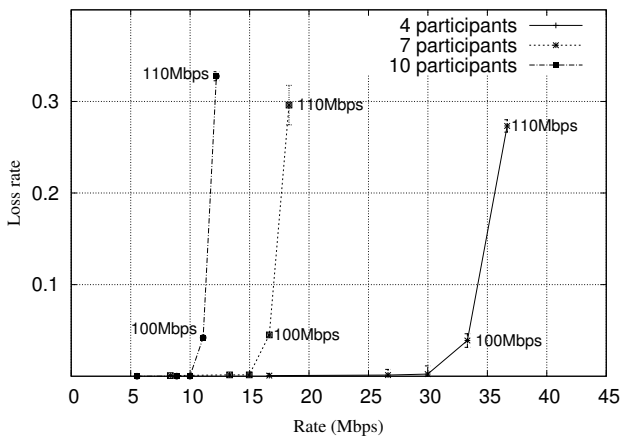

Figure 6. Percentage of IP multicast packet drops.

either machines or the connecting switches drop excess packets [10]. This leads to retransmissions, which in turn congest the network even more. Moreover, the ratio of new versus retransmitted messages drops, which lowers the throughput. These effects are known as multicast storms, and are well known to disrupt entire data centers [15], [16][5].

Note that with the network topology we use (i.e. a different Fast Ethernet LAN for clients-to-replicas communications and for replicas-to-replicas communications), multicast problems mostly affect PBFT. Zyzzyva is not affected as there is only a single sender in the multicast group. In contrast, we have observed that in a configuration with only one Fast Ethernet LAN, Zyzzyva is affected by the clients-to-replicas traffic, which creates contention and leads to IP multicast packet drops. Finally, let us note that these experiments explain why we observed very poor performance when enabling the client-multicast optimization implemented in PBFT and Zyzzyva. Indeed, when enabling this optimization, all clients can potentially multicast requests concurrently, which yields many packet drops and does thus drastically decrease performance.

D. Summary

Table I summarizes our analysis of the obstacles towards achieving high throughput in state-of-the-art BFT protocols. All protocols but PBFT suffer from asymmetric replica processing. They all use the network in an unbalanced way. Finally, PBFT is subject to IP multicast losses.

IV. Ring protocol

Based on the observations reported in the previous section, we devised Ring, a new BFT protocol that aims at achieving very high throughput. Ring, as its name indicates, uses a ring topology for message dissemination between replicas. In this sense, Ring shares similarities with the LCR [7] protocol. A major difference with LCR is that Ring tolerates Byzantine failures (of both replicas and clients), whereas LCR only tolerates crash failures. The extension to Byzantine faults is complex, as the protocol must ensure that: (1) no replica in the ring can be bypassed, (2) Byzantine clients sending malformed requests cannot corrupt the total order on correct requests, and (3) the reply sent by the last process in the Ring is not forged . Ring uses two modes: a *fast* mode that is executed when there are no replica faults and a *resilient* mode that is executed only when one or more replicas in the Ring are faulty.

[5]IP multicast losses can be reduced by carefully configuring buffer sizes, and/or synchronizing distributed senders (as in the Spread communication toolkit [17]). However, this is a difficult, if not impossible, task in a Byzantine environment, as malicious replicas can simply send traffic at high rate, disrupting complete communication in the group.

	CPU asymmetry	Underutilized replicas	IP multicast
PBFT	-	√	√
Zyzzyva	√	√	-
Chain	√	√	-

Table I
Summary of obstacles towards achieving high throughput

We start the section by describing the system model. We then present an overview of the protocol, followed by two subsections describing the *fast* and *resilient* modes, respectively. Finally, we describe various optimizations that we implemented to improve the performance of Ring.

A. System model

Our model and assumptions are similar to those made by BFT protocols studied in Section II. We assume a Byzantine failure model where (faulty) replicas or clients may behave arbitrarily. Replicas are assumed to fail independently, and we assume an upper bound f on the number of faulty replicas in a given window of vulnerability. There is no upper bound on the number of faulty clients. We assume a strong adversary that may coordinate the actions of faulty nodes in an arbitrary manner. However, the adversary cannot subvert standard cryptographic assumptions about collision-resistant hashes, encryption and digital signatures. Moreover, we assume that the state-machine replicated using Ring is deterministic. Finally, Ring ensures safety in an asynchronous network that can drop, delay, corrupt, or reorder messages. Liveness is guaranteed only under eventual synchrony [18].

B. Protocol overview

Ring is named after the ring topology it uses for communications between replicas. Unlike most BFT protocols, Ring does not use IP multicast: it only relies on unicast message exchange. Each replica in Ring has exactly one predecessor, and exactly one successor. Communication flows in one direction over the ring, with each replica forwarding requests to its successor.

Ring has two operational modes: a fast mode and a resilient mode. Ring uses the ABSTRACT framework [6] to switch between the two modes when faults are detected. The fast mode is very efficient during executions where there are no faulty replicas. Note that, in the fast mode, Ring allows committing requests even if there are faulty clients. The resilient mode ensures progress in the presence of faulty replicas.

Ring alternates between the fast and resilient modes as follows: it first runs in the fast mode, with high performance, until a fault occurs. When a fault occurs on a replica, Ring switches to the resilient mode. Since the resilient mode does not ensure high performance, Ring stays in the resilient mode until it processes 2^k requests. Parameter k represents the invocation number of the resilient mode. It is reset after reaching a threshold.

C. Fast mode

The message pattern used in the fast mode is depicted in Figure 7. A client can submit a request to *any* replica, which is called the *entry* replica for that particular request (for instance, replica 2 is the entry replica for the request in the example in Fig. 7). Each submitted request is forwarded around the ring until it reaches the predecessor of the entry replica (replica 1 in the example). At the end of this first round, each replica owns a copy of the request. One replica in the Ring, called the *sequencer* (replica 0 in the example), is in charge of assigning a sequence number to each new request it receives. This sequence number is added to the header of the message. In order for each replica to learn this sequence number, the predecessor of the entry replica, called the *exit* replica (for that particular request) generates an acknowledgement (*ACK*) for the request that is forwarded around the ring (dashed arrow in the example). The *ACK* message only contains the header of the message. The *ACK* message is forwarded until it reaches the exit replica (replica 1 in the example). This replica does then reply to the client. Note that each replica executes the request only when it receives the *ACK* message.

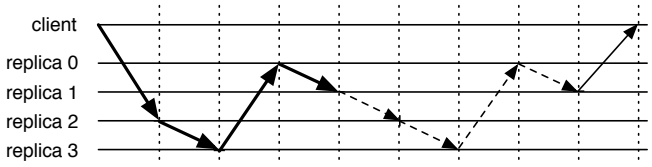

Figure 7. Ring communication pattern in the fast mode.

The protocol must ensure that no replica is bypassed and that messages are not corrupted by replicas before being forwarded around the ring. This is achieved using Ring Authenticators (RA), which share similarities with Chain Authenticators presented in [6] but have significant differences due to the presence of ACK messages. Ring Authenticators are implemented with message authentication codes (MACs). Roughly speaking, to be able to tolerate f faults, each replica generates (resp. verifies) $f + 1$ MACs for (resp. from) its $f + 1$ successors (resp. predecessors).

Figure 8 depicts the flow of a request, along with involved MAC operations. The red *underlined* text represents generated MACs, while the green *strikedthrough* text represents verified MACs. In step 1, the client sends its request (and chooses replica 2 as the entry replica). The client generates two MACs, one for replica 2, and one for replica 3 (in total, $f + 1$ MACs). These two MACs represent the RA generated by the client. Replica 2 receives the message and verifies the MAC generated by the client. In step 2, replica 2 generates its RA – containing two MACs, one for replica 3, and one for replica 0 – and forwards the request to replica 3. Replica 3 receives the request, verifies the MAC from the client, and one MAC from its predecessor – replica 2. Steps 3 and 4 are similar. In step 5, replica 1 (the exit replica, i.e., the predecessor of the entry replica) generates an ACK for the given request and forwards the acknowledgement to its successor – replica 2. Before sending the ACK, replica 1 generates MACs for replica 2 and replica 3. Replica 2 receives the ACK and verifies the MAC from replica 0 (generated for the request the replica already received), and a MAC from replica 1. In step 6, replica 2 forwards the ACK, after generating MACs for replica 3 and replica 0. Steps 7 and 8 are similar. Finally, in the last step, replica 1 verifies MACs for the ACK from replica 3 and replica 0. Replica 1 then generates one MAC for the client and sends the reply to the client. The client receives the reply and verifies two MACs – one from replica 0, and one from the replica 1. If these MACs are correct, the client commits the reply.

In case a client does not receive a correct reply (see last step in Figure 8), or in case the client does not receive a reply at all, it sends a *panic* message to all replicas after a timeout. A *panic* message contains the uncommitted request that timed out without committing. The goal of the *panic* message is to switch from the fast to the resilient mode. Byzantine clients might deliberately generate fake *panic* messages to force the system to switch to the resilient mode. To prevent this attack, before switching to the resilient mode, Ring uses the following, novel mechanism: upon receiving a *panic* message from a client, a replica handles the request *on behalf* of the client. It forwards the request to the sequencer, waits until the request gets processed along the ring, (possibly) receives the response and replies to the client. If the replica does not receive a response, this means that it was indeed necessary to switch to the resilient mode. The replica does thus broadcast a message to other replicas to ask them to switch to the resilient mode. As soon as $2f+1$ replicas send such messages, Ring switches to the resilient mode.

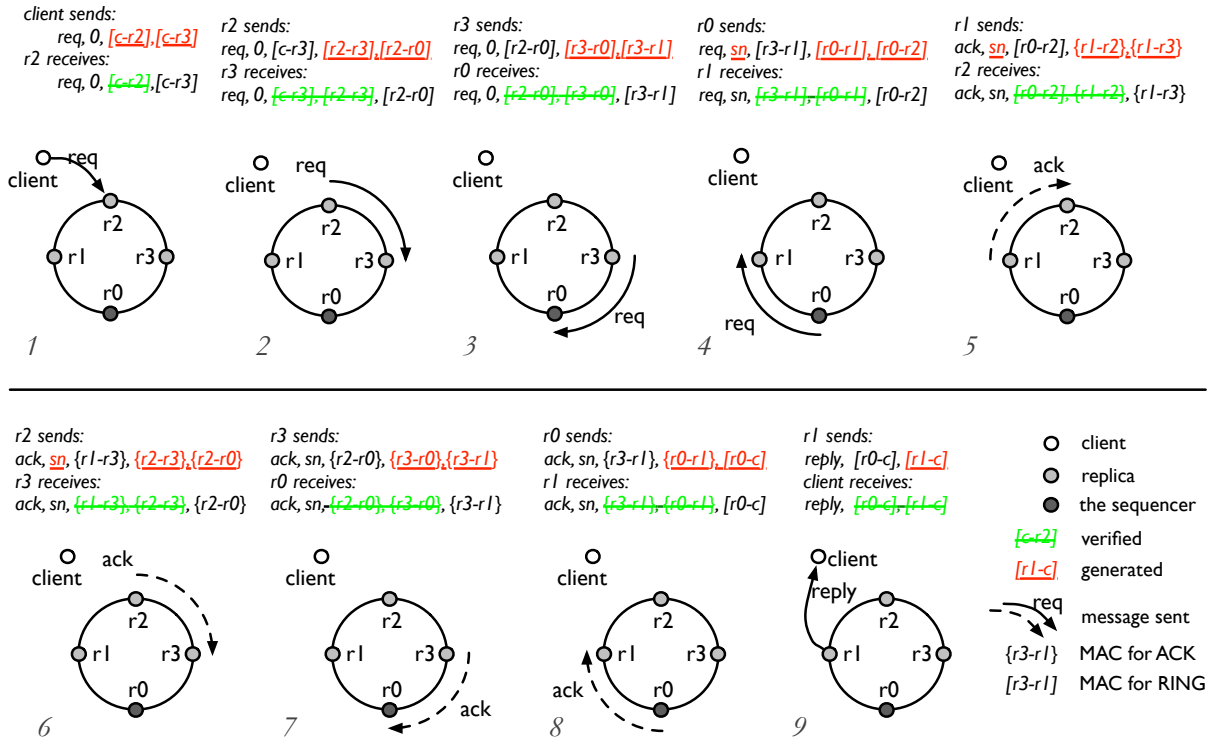

Figure 8. Illustration of Ring authenticators (f=1).

D. Resilient mode

In the resilient mode, clients and replicas sign all requests, instead of using MACs. Requests are handled as in the fast mode but replicas verify and generate signatures rather than MACs. The main difference in the resilient mode is the way *on behalf* requests are handled. The flow of an *on behalf* requests is depicted in Figure 9 (for the sake of clarity, only the steps performed by replica 2 are represented). Replica 2 sends an *on behalf* request to the sequencer. The sequencer assigns a sequence number and forwards the request to the next $f+1$ replicas (replicas 1 and 2). This step is necessary to prevent malicious replicas from blocking the request flow. Similarly, each node in the Ring, upon receiving an *on behalf* request, authenticates and forwards the request to its $f+1$ successors. When the originating replica (replica 2) receives the *on behalf* request with at least $2f+1$ signatures from different replicas, it replies back to the client.

If the sequencer is faulty, replicas will detect it (either with a timeout, or with a malformed *on behalf* request). In that case, they vote to switch to a new configuration with a different sequencer. As soon as $2f+1$ replicas issued such a vote, the order of nodes in the Ring is changed (which changes the sequencer). After the resilient mode committed k requests, Ring switches to the fast mode.

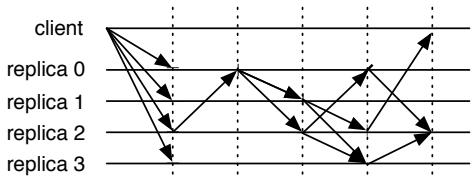

Figure 9. Processing of an *on behalf* request in Ring (in the resilient mode). Only requests from replica 2 are shown. The client broadcasts the *PANIC* message to all replicas. Replica 2 generates an *on behalf* request and sends the request to the sequencer. From that point, each replica forwards the request to the next $f+1$ replicas. Once replica 2 receives $2f+1$ *on behalf* requests signed by different replicas, it replies back to the client.

E. Optimizations

We have implemented a set of optimizations to improve the performance of Ring. These optimizations mostly aim at reducing the number of performed MAC operations per request and the number of sent messages. Designing these optimizations has been challenging because Ring Authenticators carry dependencies on the request for the next $f+1$ communication steps. For example, consider a request entering the system at replica 1. Replica 1 receives the acknowledgement from the sequencer (replica 0), and needs to authenticate the request using the MAC from both replica 3 and replica 0. Replica 3 created the MAC for the request without the sequence number. Replica 0 created the

MAC for the acknowledgement, with the sequence number. Replica 1 needs to take into account both these facts when verifying MACs. Consequently, the first two optimizations we present (piggybacking and batching) have been quite difficult to implement.

Piggybacking. The goal of this optimization is to reduce the number of messages that are sent over the network. The optimization works as follows: when a replica generates the *ACK*, it takes one (or more) client request(s), and piggybacks the *ACK* to the request. The replica then generates the RAs for the union, and sends the request. When the *ACK* reaches the last replica, the latter needs to take special care to generate proper MAC for the client, and also to generate proper MAC for the request(s) to which the *ACK* was piggybacked. Note that this optimization can be considered fragile, as malicious clients can try to disrupt the performance of Ring by sending malformed messages, which will be dropped at later replicas. Indeed, when an acknowledgement is piggybacked onto a new request, and the request authentication fails, both the *ACK* and the request will be dropped. For that purpose, we decided to disable this optimization when the number of committed requests between two switches to the resilient mode is below a configurable threshold.

Batching. The goal of this optimization is both to decrease the number of messages that are sent over the network and the number of MAC operations that are performed per request. When a replica receives a request from a client, the replica checks whether there are other pending requests from other clients. If there are such requests, the replica batches them together, generates the RAs for the union, and forwards the batch. Note that the first $f + 1$ replicas need to verify client MACs for each single request, and a joint MAC for the whole batch. Moreover, the last $f + 1$ replicas need to generate MACs for the whole batch for their successors in the Ring, and separately a MAC for every client. Finally, note that when generating the *ACK* for the batch, the replica creates a batch of *ACK*s, to allow for message fragmentation. Also, using batch of *ACK*s eases handling of checkpoints, and client request retransmissions. Similarly to piggybacking, this is a fragile optimization that we disable when the number of committed requests between two switches to the resilient mode is below a configurable threshold.

Read optimization. The goal of the last optimization is to reduce the latency of read requests. Read requests do not need to be totally ordered. Consequently, they do not need to go twice around the ring. Consequently, read requests exit the Ring after reaching $f + 1$ replicas. When a client receives the reply to its read request, it compares the $f + 1$ MACs contained in the reply. If they match, the client commits the reply. Otherwise, the client sends the read request as a write request for it to be totally ordered. Note that read requests can be batched with write requests. However, that complicates authentication and verification of requests (generation of MACs). Hence, in order to keep the protocol implementation simple, read requests are only batched together.

Note that we also tested the read optimization used in state-of-the-art BFT protocols [1], [2], [5], [19], where clients multicast their read requests to $f + 1$ replica and wait for $f + 1$ matching replies. We observed that this approach was not yielding good performance. The reason is that we had a very high number of requests retransmission, due to mismatching MACs (as different replicas on the ring were in different states). The reason is that the pipe-lining approach used for request propagation interferes with the parallel approach used to send read requests.

V. Evaluation

In this section we report the results of our performance evaluation of Ring and of the three protocols described in Section II: PBFT, Chain, and Zyzzyva. We implemented Ring in C++. Replicas and clients communicate via TCP. In order to be able to handle a large number of client connections, we use the $epoll$ event-notification mechanism. Indeed, we observed that $epoll$ is more efficient than the $select$ mechanism. Moreover, in order to prevent malicious participants from exhausting all network resources, Ring uses a token bucket [20] for establishing fairness among TCP flows. In our implementation, the token bucket splits incoming throughput in ration 3:1 between the predecessor and (all) client traffic.

The section starts by a description of the experimental setup we used. We then show that, unlike existing protocols, Ring equally balances both the CPU utilization on the various replicas, and the network utilization on the various network links. Finally, we compare the performance of Ring to that achieved by state-of-the-art protocols. More precisely, we show that Ring significantly outperforms other protocols in terms of throughput (+27%), and that it achieves up to 14% lower response time than state-of-the-art protocols when a large number of clients issue requests.

A. Experimental setup

As described in Section III, we performed experiments on the Emulab [11] testbed. In each experiment, we used *pc3000* machines – a Dell PowerEdge 2850s systems, with a single 3 GHz Xeon processor, 2 GB of RAM, and 2 NICs. Replicas are systematically running on their own, separate machine, while clients are collocated on a total of 15 machines. Moreover, in all experiments we use 4 replicas (which consists in tolerating $f = 1$ fault). Finally, we use a topology where replicas belong to one LAN, and clients communicate with replicas over a second LAN.

We use the benchmarks described in PBFT [1]: clients perform requests in a closed-loop manner. We can vary the size of the request issued by clients and the size of the replies produced by the replicas. In the presented experiments, the size of replies was set to 8 Bytes. This is motivated by the fact that the size of the replies do not impact the presented results because: (1) replicas and clients are located on different LANs (with 2 NICs per replica), and (2) replies do not circulate among replicas (they are simply sent on the LAN connecting clients to replicas, contrarily to requests that are exchanged between replicas). Concerning the size of requests, we varied their size in the range $[8\text{B}, 16\text{kB}]$. Each experiment was performed three times. On each graph, we report the average of these three executions.

B. CPU utilization

Figure 10 depicts the CPU utilization for Ring, along with the CPU utilization of other protocols (as in Figure 4 in Section III), for comparison. We can observe that all replicas in Ring are equally loaded. This comes from the fact that there is no asymmetry in replica processing: all replicas perform the same computation (the only minor difference is that the sequencer appends a sequence number to the requests it receives) and each replica receives the same amount of client requests (provided clients homogeneously balance their requests on the different replicas, which is trivially achieved by having clients choose the entry replica in a round-robin manner). The consequence is that no replica is bottleneck in Ring.

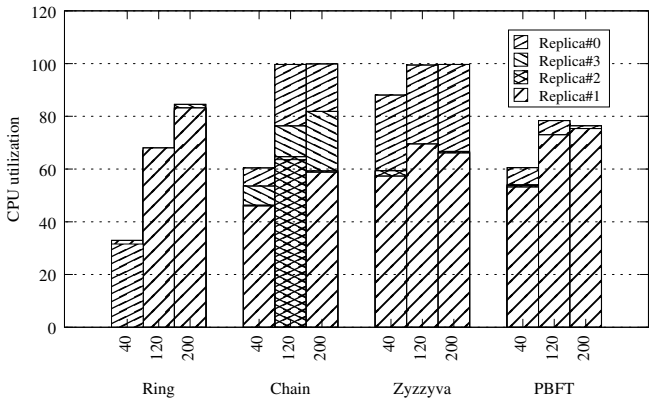

Figure 10. CPU utilization of Ring (and of other protocols).

C. Network utilization

Figure 11 depicts the number of bytes that are sent/received by Ring replicas for replica-to-replica communications (recall that each replica has two network interfaces: one for client-to-replica communications, and one for replica-to-replica communications). Requests issued by clients are 1 kB large. Moreover, as in Figure 5, we represent in Figure 11 the number of bytes that are sent (or received) for each byte received from a client. Bars *in* (resp. *out*) denote normalized amount of data on incoming (resp. outgoing) links to the replica.

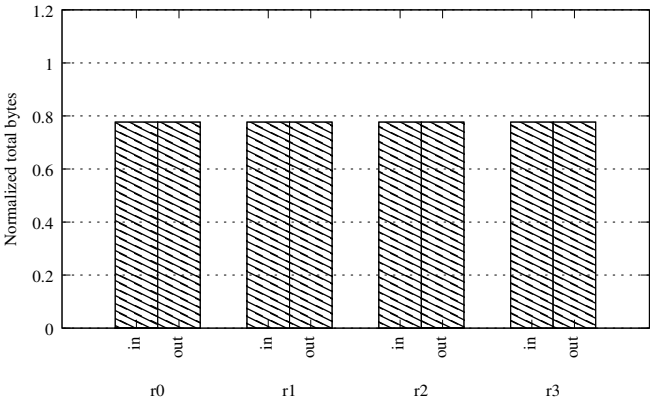

Figure 11. Network link utilization in the Ring protocol.

The first observation we can make is that the network utilization is perfectly balanced on the different links: each replica equally uses its incoming and outgoing links. The reason comes from the fact that each replica sends/receives, on average, the same number of messages. This is a consequence of the fact that each replica acts on average the same number of times as "entry replica" (and also as "exit replica"). Consequently, from a "network utilization" point of view, each replica has the same "role" in the protocol.

The second observation we can make is that for each Byte transmitted by a client, a replica only transmits (resp. receives) 0.78 Bytes on its outgoing (resp. incoming) link. This is explained by the fact that there are 4 replicas-to-replicas links, and only 3 of them are used to disseminate request payloads (the link from the exit replica to the entry replica is not used). As the role of "entry replica" is equally played by replicas, for each request, each replica has the same probability to have one of its link not used. Consequently, the average number of Bytes that is transmitted on each single link should be $\frac{3}{4} = 0.75$ Bytes, which is very close to the 0.78 Bytes we observe. The slight difference comes from the fact that messages have headers and that an acknowledgement is produced for every message, thus increasing the number of Bytes that transit over network links.

D. Performance evaluation

We have seen in the previous two Sections that Ring replicas do all perform similar processing, and do all send/receive similar number of Bytes. In this Section, we evaluate the impact of this balanced CPU and network utilization on the performance of the protocol. We first evaluate the peak throughput that can be achieved by

each protocol as a function of the message size. Then, we evaluate the throughput when varying the number of clients for 1 kB requests. Finally, we evaluate the response time of the various protocols when varying the number of clients

Peak throughput as a function of request size. We first study how the throughput of the different protocols is impacted by the size of requests issued by clients. Results are reported in Figure 12. Note that the X axis uses a logarithmic scale. The first observation we can make is that the behavior we observe is similar to that observed using simulations in work by Singh et al [13]: PBFT and Zyzzyva perform very similarly. We can also observe that reported results are quite different from those reported by Guerraoui et al [6]: we observe a much lower performance difference between Zyzzyva and Chain with large messages. This comes from the network setting we are using. Clients communicate with replicas using a separate, dedicated LAN. This drastically reduces the load on the LAN used for inter-replica communications. The consequence is that it reduces the number of IP multicast packet drops, which drastically improves the performance of Zyzzyva.

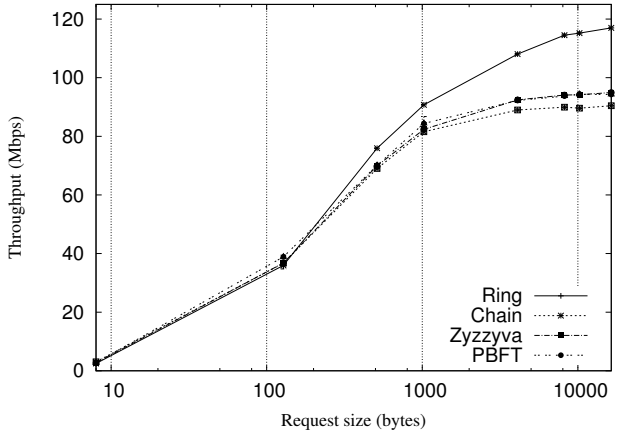

Figure 12. Peak throughput as a function of the request size.

Figure 12 also shows that with small requests (below 512 B), all protocols perform similarly. With larger requests, Ring significantly outperforms other protocols. More precisely, state-of-the-art protocols have a peak throughput ranging between 90 Mbps for PBFT and 93 Mbps for Zyzzyva and Chain. Ring, on the other hand, has a peak throughput of about 118 Mbps, which represents a 27% performance improvement over the most efficient state-of-the-art BFT protocols. Let us note that the fact that Ring achieves a throughput of 118 Mbps on a Fast Ethernet network comes from the fact that Ring replicas only send/receive 0.78 Bytes for each Byte contained in a client requests. As a conclusion, we can say that with large messages, the throughput of Ring is very close to the optimal throughput [7] that can be achieved on a Fast Ethernet LAN: 124 Mbps.

Throughput as a function of the number of clients. The next experiment we performed was to assess the throughput achieved by the various protocols when varying the number of clients. Results are reported in Figure 13. The size of requests that are issued by clients is 4 kB. Note that we do not issue 16 kB requests (which yields the best results for all protocols as illustrated in Figure 12) because both Zyzzyva and PBFT were crashing when stressed with a large number of clients (> 120) issuing 16 kB requests. We observe on Figure 13 that with low number of clients (below 80), PBFT, Zyzzyva and Chain outperform Ring. When the number of clients is higher than 80, Ring clearly outperforms other protocols. The reason is that Ring uses a pipeline pattern to disseminate requests. To be efficient, this pipeline needs to be fed, i.e. the link between any replica and its successor in the Ring must be saturated. As clients issue requests in a closed-loop manner (i.e. meaning that a client does not invoke a new request before it commits a previous one), it is necessary to have a sufficient number of clients to feed the pipeline.

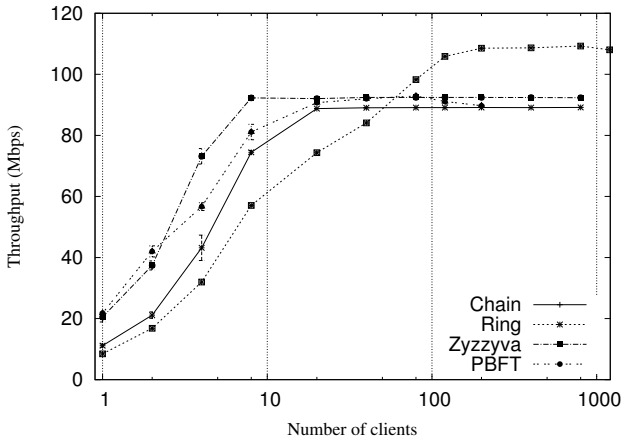

Figure 13. Throughput as a function of the number of clients.

Response time. The last experiment we conducted was to assess the response time of the different protocols as a function of the number of clients. Each client issues 4 kB requests (for the same reasons as the one mentioned in the previous paragraph). Results are depicted in Figure 14. Note that both the X and Y axes use a logarithmic scale. With a low number of clients, Zyzzyva achieves the lowest response time. This is due to the communication pattern it uses, which involves three one-way message delays. In contrast, Chain and Ring have a higher response time, which is a consequence of the pipe-lining pattern they use to disseminate requests. Nevertheless, we observe that when the number of clients increases (> 80), the response time

achieved by Ring becomes lower than that achieved by other protocols (14.5% lower with 800 clients). This is easily explained by the fact that under high contention, the response time is impacted by the throughput: the higher the throughput, the lower the time spent by requests in waiting queues.

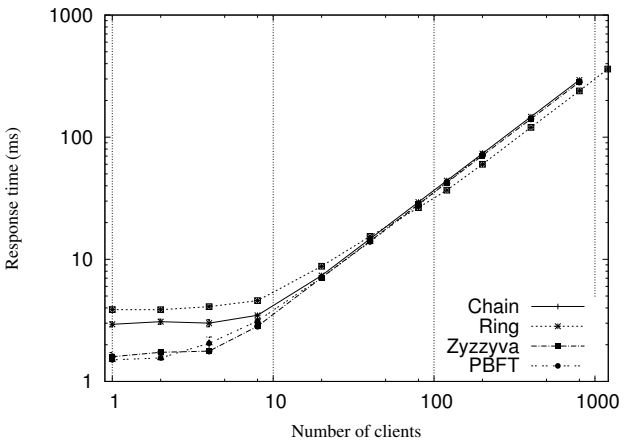

Figure 14. Response times for different benchmarks.

VI. Related work

PBFT [1] was the first practical implementation of a BFT state machine replication protocol. It was later followed by many other protocols, e.g. Zyzzyva [2], HQ [4], Q/U [3], Prime [19], Aardvark [5], Spinning [21], Zyzzyvark [22] or Scrooge [23]. Each of these protocols brought some improvement over the original design. However, none of them reports performance results similar to that achieved by Ring.

PBFT [1], Zyzzyva [2] and Chain [6] are known to be the most efficient BFT protocols in terms of throughput under high load. They have been extensively described in Section II, and evaluated in Sections III and V. We have shown that, unlike Ring, none of these protocols features both symmetric CPU processing across replicas and balanced network utilization across different links. Moreover, we have seen that Ring achieves up to 27% higher throughput than all these protocols.

Scrooge is a primary-based protocol similar to Zyzzyva and PBFT, that reduces the number of replicas needed to achieve low-latency despite faults. Scrooge has the same performance as Zyzzyva in the best case [23].

Quorum-based protocols like HQ [4], Q/U [3], and Quorum [6] offer low latency under very low load, when requests are spontaneously ordered by the LAN. When the load increases, these protocols fail to achieve high performance: the spontaneous order observed by the different replicas is often different, which requires replicas to be frequently reconciled, thus degrading performance.

A set of so-called robust BFT protocols have been designed: Aardvark [5], Prime [19], Spinning [21] and Zyzzyvark [22]. These protocols aim at offering good throughput when faults occur. These protocols, unlike Ring, do thus not optimize performance for the non-faulty case. An interesting research challenge would be to design a robust version of the Ring protocol.

A very recent position paper addresses the problem of building scalable BFT protocols [24]. The idea is to improve the throughput of replicated state machine protocols by executing multiple times the same protocol on different (intersecting) sets of machines. This idea is complementary to the one presented in this paper. Indeed, to get the most benefit out of this multiple-execution mechanism, it is necessary to have a very efficient base protocol. We do thus believe that it would be interesting to combine the technique proposed in [24] with Ring.

Finally, let us also remark that some previous works have proposed the use of ring topology in the context of total order broadcast protocols: Ring Paxos [10] and LCR [7]. Ring is not a simple extension of these protocols. The main difference between Ring and these protocols is actually that Ring tolerates Byzantine faults (of both replicas and clients), whereas Ring Paxos and LCR only tolerate crash faults, which makes their design significantly easier. Note also that another difference between Ring Paxos and Ring is that the former relies on IP multicast to disseminate sequence numbers.

VII. Concluding remarks

It is crucial to design throughput-efficient BFT protocols. State-of-the-art BFT protocols are far from achieving an optimal throughput. Indeed, the most efficient BFT protocols achieve 93 Mbps, although the theoretical maximum on such networks is 124 Mbps [7]. We studied existing protocols and implementations and identified impediments to their scalability. We found three impediments: asymmetric replica processing, unbalanced network utilization, and IP multicast packet drops.

To evaluate the benefits of circumventing these impediments, we proposed a new protocol, called Ring, which achieves very high performance when used with large messages and large number of clients. We have evaluated the performance of Ring and shown that its performance (118 Mbps on a Fast Ethernet LAN) approaches the theoretical maximum.

We believe that an interesting area for future work is to design protocols achieving performance close to the theoretical maximum when clients issue small requests. With small requests, the challenge is that cryptographic costs become dominating. Our experience shows that batching messages (as done in most existing BFT protocols) is not sufficient to achieve high throughput in that context. We do thus believe that to sustain high throughput with small messages, it will

be necessary to design protocols that are able to efficiently leverage multicore computers (e.g. protocols that perform cryptographic operations and network I/Os in parallel, on distinct cores).

REFERENCES

[1] Castro, M., Liskov, B.: Practical Byzantine Fault Tolerance. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI). (1999)

[2] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., Wong, E.: Zyzzyva: speculative Byzantine fault tolerance. In: Proceedings of the Symposium on Operating Systems Principles (SOSP), ACM (2007)

[3] Abd-El-Malek, M., Ganger, G.R., Goodson, G.R., Reiter, M.K., Wylie, J.J.: Fault-scalable Byzantine fault-tolerant services. In: Proceedings of the Symposium on Operating Systems Principles (SOSP), ACM (2005)

[4] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., Shrira, L.: HQ replication: a hybrid quorum protocol for Byzantine fault tolerance. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association (2006)

[5] Clement, A., Wong, E., Alvisi, L., Dahlin, M., Marchetti, M.: Making Byzantine fault tolerant systems tolerate Byzantine faults. In: Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI), USENIX Association (2009)

[6] Guerraoui, R., Knezevic, N., Quema, V., Vukolic, M.: The Next 700 BFT Protocols. In: Proceedings of the ACM European conference on Computer systems (EuroSys). (2010)

[7] Guerraoui, R., Levy, R., Pochon, B., Quéma, V.: Throughput optimal total order broadcast for cluster environments. Transactions on Computer Systems (TOCS) **28** (2010)

[8] Aviram, A., Weng, S.C., Hu, S., Ford, B.: Efficient system-enforced deterministic parallelism. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI). (2010)

[9] Bergan, T., Hunt, N., Ceze, L., Gribble, S.D.: Deterministic process groups in dos. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI). (2010)

[10] Jalili Marandi, P., Primi, M., Schiper, N., Pedone, F.: Ring paxos: A high-throughput atomic broadcast protocol. In: Proceedings of the Conference on Dependable Systems and Networks (DSN). (2010)

[11] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI), USENIX Association (2002)

[12] Lamport, L.: Lower bounds for asynchronous consensus (2004)

[13] Singh, A., Das, T., Maniatis, P., Druschel, P., Roscoe, T.: BFT protocols under fire. In: Proceedings of the Symposium on Networked Systems Design and Implementation (NSDI), USENIX Association (2008)

[14] SYSSTAT utilities. http://sebastien.godard.pagesperso-orange.fr/ (2010)

[15] Birman, K., Chockler, G., van Renesse, R.: Toward a cloud computing research agenda. SIGACT News **40** (2009)

[16] Vigfusson, Y., Abu-Libdeh, H., Balakrishnan, M., Birman, K., Burgess, R., Chockler, G., Li, H., Tock, Y.: Dr. multicast: Rx for data center communication scalability. In: Proceedings of the European conference on Computer systems (EuroSys), ACM (2010)

[17] Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J., Stanton, J.: The Spread Toolkit: Architecture and performance. Technical report, Johns Hopkins University (2004)

[18] Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. J. ACM **35** (1988)

[19] Amir, Y., Amir, Y., Coan, B., Kirsch, J., Lane, J.: Byzantine replication under attack. In: Proceedings of the Conference on Dependable Systems and Networks (DSN). (2008)

[20] Shenker, S., Wroclawski, J.: General characterization parameters for integrated service network elements (1997)

[21] Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C.: Spin one's wheels? Byzantine Fault Tolerance with a spinning primary. In: Proceedings of International Symposium on Reliable Distributed Systems (SRDS), IEEE Computer Society (2009)

[22] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., Riche, T.: Upright cluster services. In: Proceedings of the Symposium on Operating Systems Principles (SOSP), ACM (2009)

[23] Serafini, M., Bokor, P., Dobre, D., Majuntke, M., Suri, N.: Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In: Proceedings of the Conference on Dependable Systems and Networks (DSN). (2010)

[24] Kapritsos, M., Junqueira, F.P.: Scalable agreement: Toward ordering as a service. In: Proceedings of the Sixth Workshop on Hot Topics in System Dependability (HotDep). (2010)

APPENDIX

A. Notation

A message m sent by node p to the node q, authenticated with a MAC is denoted by $\langle m \rangle_{\mu_{p,q}}$. A node p can use vectors of MACs (called authenticators) to simultaneously authenticate message m for multiple recipients, members of some set S. Such message is denoted by $\langle m \rangle_{\alpha_{p,S}}$, and contains MACs $\mu_{p,q}$ for every $q \in S$. In addition, we denote by $D(m)$ the digest of message m, while $\langle m \rangle_{\sigma_p}$ represents a digitally signed message, i.e. a message that contains $D(m)$, signed with the private key of node p and message m. We assume that all nodes have public keys of all other nodes in the system in order to verify the signatures. Further, we assume that during synchronous periods there exists some time Δ, which represents the maximal propagation delay between any two correct nodes in the system. Finally, Σ represents the set of all $3f+1$ replicas.

Every instance[6] of *Ring* has one replica designated as the sequencer, and a fixed ordering of replica IDs (called the *ring order*), known to all processes. The sequencer precedes all replicas in the ring order, and the last replica in the ring order is the sequencer's physical predecessor on the ring. Without loss of generality, we assume that the sequencer is replica r_0. To simplify the notation, as there is a finite number of replicas, we treat the ring order as a sequence of numbers in the finite group of modulo order $3f+1$. Thus, the successor of node r_i is $r_{i \oplus 1}$, where $\oplus$ is addition modulo $3f+1$. When a replica receives a request from a client, the replica becomes the *entry replica* for the request. The replica which replies back to the client is the *exit replica* for a given request. The exit replica is the predecessor of the entry replica ($r_{exit} = r_{entry} \ominus 1$).

We indicate the *predecessor (resp., the successor) set* of replica r_j as $\overleftarrow{r_j}$ (resp., $\overrightarrow{r_j}$). Also, we denote the *sequenced predecessor set* of replica r_j, by $\widehat{r_j}$. Sequenced predecessor set of replica r_j are all replicas which may have received a request along with the sequence number from the sequencer. We will precisely define these sets in Section B. We also reference by Σ_{last} the set of the last $f+1$ replicas in the ring order, i.e., $\Sigma_{last} = \{r_j \in \Sigma : j \geq 2t\}$. Further, we denote by Σ_{last}^{req} the set of the last $f+1$ replicas in the ring order, with respect to request req, i.e., $\Sigma_{last}^{req} = r_j \in \Sigma : j \in \{(req.entry \ominus (f+1) \dots req.entry \ominus 1\}$.

B. Protocol overview

In Ring, a client sends request req to *any* replica r_i. In turn, each replica passes the request to its *successor*, i.e. replica r_j forwards the request to $r_{j \oplus 1}$. Upon reaching replica r_0 (the sequencer) on this path, request req gets a *sequence* number. When the request reaches $r_{i \ominus 1}$ (the exit replica for the request), $r_{i \ominus 1}$ sends an acknowledgement for the request to its successor. Replicas forward this acknowledgement in the same way as the original request. After $r_{i \ominus 1}$ receives the acknowledgement back, all replicas are aware of the request's sequence number. Then, replica $r_{i \ominus 1}$ replies back to the client. Each replica in Ring accepts only messages (both requests and acknowledgements) sent by the replica's predecessor, or the client[7]. Now, we introduce a couple of definitions we use later in the text.

Ring has two operating modes: a best case execution mode, called the fast mode, and the resilient mode. The fast mode is intended for situations where Ring should provide the best possible performance, under the assumption that there are no faulty replicas. On the other hand, the resilient mode provides resilience (and good performance) when faulty replicas are present in the system. In this section, for the sake of brevity, we call the fast mode Ring$^-$, while Ring$^+$ denotes the resilient mode.

Definition For a given request processed at a replica, we define the *distance* of the request as the number of replicas the request was processed at, since the entry replica.

Hence, a request at the entry replica has distance 0. At the time of replying to a client, the request is at the distance $2n-1$. Clearly, the distance of an acknowledgement is always greater than $3f$, as we have two rounds of communication (one to propagate the request, and the second one to acknowledge the sequence number).

Definition The *predecessor set* of replica r_j (with respect to some request req), denoted by $\overleftarrow{r_j}$ represents at most $f+1$ replicas which are direct predecessors of r_j.

The predecessor set of replica r_j is: (a) if $distance(r_j, req) \leq f+1$, $\overleftarrow{r_j} = \{r_{j \ominus distance(r_j, req)} \cdots r_{j \ominus 1}\}$, else (b) $\overleftarrow{r_j} = \{r_{j \ominus (f+1)} \cdots r_{j \ominus 1}\}$.

Definition The *successor set* of replica r_j (with respect to some request req), denoted by $\overrightarrow{r_j}$ represents at most $f+1$ replicas which are direct successors of r_j.

The successor set of replica r_j is: (a) if $distance(r_j, req) \leq 2n-f-1$, $\overrightarrow{r_j} = \{r_{j \oplus 1} \cdots r_{j \oplus (f+1)}\}$, else (b) $\overrightarrow{r_j} = \{r_{j \oplus 1} \cdots r_{exit\_replica}\}$.

[6] we refer to an ABSTRACT instance

[7] clients do not send the acknowledgement

Variable	Purpose
RASET	Set of Ring Authenticators, used by both clients and replicas
MACSET	Set of *MAC*s, authenticating the reply, generated by replicas
LH	Replica's local history
$self$	Variable holding replica's id
sn	Sequence number associated with a request
$lastreq$	Array indexed by a client id, holding the last request sent by the client
$lastsn$	Array indexed by a client id, holding a last sequence number given to a request from the client
$lasthist$	Array indexed by a client id, holding the last history sent to the client
$active$	A boolean representing a running state of a particular ABSTRACT instance
$sequencer\_id$	A variable containing the id of the sequencer in the current ABSTRACT instance
$pending$	A list of pending requests at the replicas
$OBRpending$	A list of pending *OBR* requests at the replica

Table II
LEGEND OF USED VARIABLES

Field name	Purpose
o	Replicated state machine command
t_c	Client's timestamp for the request
cid	Client's id
$entry$	The id of the *entry* replica to which the client sent the request

Table III
FIELD NAMES FOR THE REQUEST

For example, the predecessor of the exit replica has only one replica (the exit replica) in the successor set for the acknowledgement. Likewise, the predecessor set of one replica after the entry replica, for the request, contains only the entry replica.

Definition The *sequenced predecessor set* of replica r_j (with respect to some request req), denoted by $\widehat{r_j}$ represents at most $f+1$ direct predecessors of r_j which may have received the request with a sequence number from the sequencer.

The sequenced predecessor set of replica r_j is $\widehat{r_j} = \{r_i \in \Sigma : i \geq max(0, j-(f+1))\}$.

Every replica r_i uses a Ring Authenticator (RA) to authenticate a message (either a request or an acknowledgement) for all replicas in its *successor set* $\overrightarrow{r_i}$. Consequently, when a replica in Ring receives a message m, the replica *verifies* m, i.e., the replica checks whether m is correctly authenticated by the all replicas in the predecessor set.

C. Legend

Before giving the pseudo code, we list the variables we use in our algorithm, along with their explanation.

Now, we give the pseudo code for the client, and two versions for the server: one for the normal mode, and other for the resilient mode. In Appendix E and F we give the explanation of the pseudo code.

D. Pseudo code

Algorithm A.1: Ring: client pseudo-code

```
(1)  procedure initialization() ≡
(2)    t_c, entry, rreplica ← 0; T_Ring := (2(3f + 1) + 2)Δ
(3)
(4)  procedure invoke(o) ≡
(5)    t_c ← t_c + 1;
(6)    entry ← any number in 0..3f; rreplica ← entry ⊖ 1        predecessor on the ring answers
(7)    req ← ⟨o, t_c, self as cid, entry⟩
(8)    send⟨RING, req, nil, ∅, ∅⟩_σc to r_entry
(9)
(10) upon received⟨⟨REPLY, req, MACSET⟩, LH⟩ from r_rreplica ≡
(11)   if ∀r_i : (r_i ∈ ←r_entry) ⇒ (MAC(r_i, self, ⟨req, D(LH)⟩) ∈ MACSET
(12)     then trigger(COMMIT(req, LH)); cancel(T_Ring) endif
(13)
(14) upon T_Ring expires ≡
(15)   send⟨PANIC, req_σc⟩to all servers
(16)
(17) upon received⟨GET-A-GRIP, h, req⟩ from f + 1 different servers with the same h ≡
(18)   trigger(COMMIT(req, h))
(19)
(20) upon received⟨ABORT, LH_i, req, r_i⟩ from 2f + 1 different servers, with the matching req ≡
(21)   LH ← extract_history(∪_i LH_i)
(22)   trigger(ABORT(req, LH))
(23)
```

Algorithm A.2: Ring$^-$: server r_i pseudo-code

```
(1)  procedure initialization() ≡
(2)    pending ← ∅
(3)    sn ← 0
(4)    active ← true
(5)    T_OBR ← (3f + 1)Δ
(7)    ∀c ∈ Clients : lastreq[c] ← nil; lastsn[c] ← 0; lasthist[c] ← nil
(8)
(10) procedure sequence_request(sn′, req) ≡
(11)   if i = sequencer_id then sn, sn′ ← sn + 1 endif
(12)
(14) procedure execute(sn′, req) ≡
(15)   if lastreq[req.c].t_c ≥ req.t_c then return endif
(16)   sn ← sn′
(18)   lasthist[req.c] ← (LH ← LH ∘ ⟨req⟩)
(19)   lastreq[req.c] ← req; lastsn[req.c], sn_j ← sn
(21)   for req′ ∈ OBRPending ∧ req′.c = req.c do
(22)     if req′.t_c < lastreq[req.c].t_c
(23)       then OBRPending ← OBRPending \ {req′}
(24)            stop(T_OBR_req′)
(25)     endif
(27)
(29) upon received⟨RING, req, sn′, RASET, MACSET⟩_σc from r_i⊖1 ∨ client c ≡
(30)   when ⋀ active
(31)        ⋀ distance(r_entry, r_i) ≤ f ⟹ valid signature
(33)        ⋀ checkRASET(RASET, req)
(35)        ⋀ req.t_c > lastreq[req.c].t_c ∧ (sn′ ≠ nil ⟹ sn′ = sn + 1)
(36)    begin
(37)      pending ← pending ∘ {req}
(38)      sequence(sn′, req)
(39)      if sn′ ≠ nil then execute(sn′, req) endif
(40)      RASET ← updateRAs(RASET, req, sn′, ⊤)
(41)      if i = predecessor(req.entry)
(42)        then send⟨ACK, sn′, D(req), req.c, RASET, ∅⟩ to r_i⊕1
(43)        else send⟨RING, req, sn′, RASET, ∅⟩ to r_i⊕1
(44)      endif
(45)    end
(46)
(48) upon received⟨ACK, sn′, D′, c, RASET, MACSET⟩ from r_i⊖1 ≡
(49)   when ⋀ active
(50)        ⋀ ∃req ∈ pending | req.c = c ∧ D(req) = D
(52)        ⋀ checkRASET(RASET, req)
(54)        ⋀(sn′ = sn + 1)
(56)    begin
(57)      if sn′ = sn + 1 then execute(sn′, req) endif
(58)      pending ← pending \ req
(59)      myMACSET ← updateMACs(MACSET, req, req.c, LH, ⊥)
(60)      RASET ← updateRAs(RASET, req, sn, ⊥)
(61)      if predecessor(req.entry) = self
(62)        then send⟨⟨REPLY, req, myMACSET⟩, LH⟩ to req.c
(63)        else send⟨ACK, sn, D(req), RASET, myMACSET⟩ to r_i⊕1 endif
(64)    end
```

```
(65)
(67) upon received⟨PANIC, req⟩_{σ_c} from client c ≡
(68)   when ⋀ active
(69)        ⋀ req.t_c ≥ lastreq[req.c].t_c
(70)        ⋀ req is valid
(71)    begin
(72)      OBRPending ← OBRPending ∪ {req}
(73)      SIGSET ← σ_self(self, req.c, D(req))
(74)      send⟨OBR, self, req, 0, SIGSET, ∅⟩σ_self to r_sequencer_id
(75)      trigger(T_OBR_req)
(76)   end
(77)
(79) upon received⟨PANIC, req⟩_{σ_c} from client c ≡
(80)   when active = false
(81)    begin
(82)      send⟨ABORT, LH_{σ_{s_i}}, req, self⟩σ_self to the client req.c
(83)   end
(84)
(86) upon received⟨OBR, r_j, req_{σ_req.c}, sn′, SIGSET, MACSET⟩ from r_k ≡
(87)   when ⋀ active
(88)        ⋀ req_{σ_req.c} is valid
(90)        ⋀ ∀r_j ∈ self : ∃sig ∈ SIGSET : sig = σ_{r_j}(r_j, req.c, D(req))
(91)        ⋀ req.t_c ≥ lastreq[req.c]_t.c
(93)        ⋀ i > 0 ⟹ sn′ = sn + 1
(94)    begin
(95)      if req = lastreq[req.c] ∧ lastsn[req.c] ≠ nil
(96)        then sn_OBR ← lastsn[req.c]; LH_OBR ← lasthist[req.c]
(97)        else
(98)             sequence(sn′, req)
(99)             execute(sn′, req)
(100)            sn_OBR ← sn; LH_OBR ← LH
(101)     endif
(102)     MACSET′ ← updateMACs(MACSET, req, r_j, LH_OBR, ⊤)
(103)     SIGSET ← SIGSET ∪ σ_self(self, req.c, D(req))
(104)     if i = sequencer_id ⊖ 1
(105)       then send⟨⟨OBR, r_j, req_{σ_req.c}, sn_OBR, ∅, MACSET′⟩, LH_OBR⟩ to r_j
(106)       else send⟨OBR, r_j, req_{σ_req.c}, sn_OBR, SIGSET, MACSET′⟩ to r_{i⊕1}
(107)     endif
(108)  end
(109)
(111) upon received⟨⟨OBR, self, req_{σ_req.c}, ∗, ∗, MACSET⟩, h⟩ from r_{sequencer_id⊖1} ≡
(112)  when active
(113)   begin
(114)     if ∀r_j : (r_j ∈ ←r_sequencer_id) ⟹ (MAC(s_j, self, D(h)) ∈ MACSET)
(115)       then
(116)            send⟨GET_A_GRIP, h, req⟩σ_self to req.c
(117)            stop(T_OBR_req) endif
(118)  end
(119)
(121) upon T_OBR_req expires ≡
(122)  active ← false
(123)  send⟨STOP, LH_{σ_{s_i}}, req, self⟩σ_self to every replica r_k
(124)  send⟨ABORT, LH_{σ_{s_i}}, req, self⟩σ_self to the client req.c
(125)
(127) upon received⟨STOP, LH, req, r_j⟩ from 2f + 1 different servers with matching req ≡
(128)  active ← false
(129)  send⟨STOP, LH, req, r_i⟩ to every replica r_k
(130)
```

Algorithm A.3: Ring$^+$: server r_i pseudo-code for resilient mode (changes compared to Ring$^-$)

```
(1)  procedure initialization() ≡
(2)    pending ← ∅
(3)    sn ← 0
(4)    active ← true
(5)    stored_sigs ← ∅
(6)    T_OBR ← (3f + 1)Δ
(7)
(9)  upon received⟨OBR, r_j, req_{σ_req.c}, sn′, SIGSET, MACSET⟩ from r_k ≡
(10)   when ⋀ req.t_c ≥ lastreq[req.c]_t.c
(12)        ⋀ req_{σ_req.c} is valid
(14)        ⋀ i > 0 ⟹ (sn′ = sn + 1)
(15)    begin
(16)      SIGS ← valid signatures in SIGSET from different servers
(17)      stored ← stored_sigs[req]
(18)      if SIGS ⊂ stored then break upon endif
(19)      stored_sigs[req] ← stored ∪ SIGS
(20)      if ‖stored‖ ≥ 2f + 1
(21)        then
(22)             if req = lastreq[req.c] ∧ lastsn[req.c] ≠ nil
```

```
(23)                then sn_OBR ← lastsn[req.c]; LH_OBR ← lasthist[req.c]
(24)                else
(25)                     sequence(sn', req)
(26)                     execute(sn', req)
(27)                     sn_OBR ← sn; LH_OBR ← LH
(28)                endif
(29)                SIGSET ← SIGSET ∪ σ_self(self, req.c, D(req)) ∪ stored
(30)                if (j = i) then send⟨GET_A_GRIP, h, req⟩σ_self to req.c endif
(31)                send⟨OBR, r_j, req_σreq.c, sn_OBR, SIGSET, MACSET'⟩ to r_i→
(32)           else
(33)                SIGSET ← SIGSET ∪ σ_self(self, req.c, D(req)) ∪ stored
(34)                send⟨OBR, r_j, req_σreq.c, sn', SIGSET, MACSET'⟩ to r_i→ endif
(35)      end
(36)
```

Algorithm A.4: Ring: misceleneous functions for server code

```
(2)  function distance(id_1, id_2) ≡ return (id_2 ⊖ id_1)
(4)  function checkRASET(RASET, req) ≡
(5)    comment: checks the well-formedness of RASET, as well as MACs
(6)    RASET' ← sort(<_(3f+1,req.entry)), RASET)
(8)    if ∃R, S ∈ RASET' : R = ⟨*, *, sn_1, *⟩, S = ⟨*, *, sn_2, *⟩,
(9)      sn_1 ≠ nil ∧ sn_2 ≠ nil ⇒ sn_1 ≠ sn_2
(10)     then return false endif
(11)   if ∃R, S ∈ RASET' : R = ⟨j_1, i_1, sn_1, *⟩, S = ⟨j_2, i_2, sn_2, *⟩,
(12)     (j_1, i_1) <_(3f+1,i)) (j_2, i_2) ⇒ sn_1 ≠ nil ∧ sn_2 = nil then return false endif
(14)   if ⋀ distance(req.entry, i) ≤ f + 1 ⇒
(16)     ∃RA ∈ RASET | RA = ⟨i, *, sn', MAC'⟩, MAC' = MAC(self, req.cid, ⟨req, sn'⟩)
(18)     ⋀ ∀r_j ∈ self→ :∃RA ∈ RASET | RA = ⟨i, j, sn', MAC'⟩, MAC' = MAC(self, j, ⟨req, sn'⟩)
(19)     then return true endif
(21)   return false
(22)
(24) function updateRAs(RASET, req, sn, is.req) ≡
(25)   myRASET ← RASET; myRA ← ∅
(26)   for all RA ∈ RASET do
(27)     RA' ← RA
(28)     if RA' =< self, *, *, * > then myRASET ← myRASET \ RA' endif
(31)   if distance(req.entry, self) ≤ 2f ∨ is.req = ⊤
(32)     then end ← self ⊕ f ⊕ 1
(33)     else end ← req.entry ⊖ 1 endif
(34)   for j = self ⊕ 1 to end do                    iterate clockwise on the circle
(35)     if is.req = ⊤
(36)       then myRA ← myRA ∪ ⟨j, self, sn, MAC(r_j, self, ⟨Type^REQ, req, sn⟩)⟩
(37)       else myRA ← myRA ∪ ⟨j, self, sn, MAC(r_j, self, ⟨Type^ACK, req, sn⟩)⟩
(38)     endif
(39)   myRASET := myRASET ∪ myRA
(40)   return myRASET
(41)
(43) function updateMACs(MACSET, req, c, LH, is.req) ≡
(44)   myMACSET ← MACSET
(45)   if distance(req.entry, i) > 2f ∧ is.req = ⊥
(46)     then myMACSET := myMACSET ∪ MAC(c, self, ⟨req, D(LH)⟩) endif
(47)   return myMACSET
```

E. Ring⁻ algorithm steps

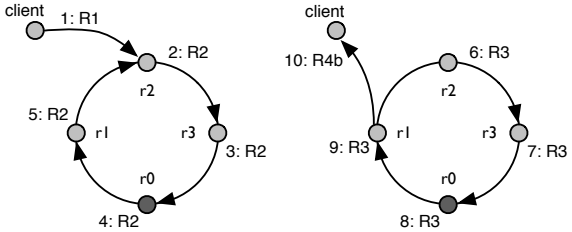

Figure 15. A client invokes a request, by sending a message to replica r_2, and afterwards receives a reply from replica r_1. The number before the label denotes the order of steps, while the label corresponds to the label in the description.

Step R1. *A client can send a request to any replica (Algorithm A.1, lines 4–8)*

When invoking operation o, client c first chooses i, the index of the *entry* replica (line 6 of Algorithm A.1). Then, client c forms a $req = \langle o, t_c, c, i \rangle$, which contains the operation in question, the identifier for the request, client's id, and the id of

the entry replica. Next, the client creates a request (a *RING* message) for replica r_i (line 8). The client does not fill the last three fields (sequence number, *RASET*, and *MACSET*) of the message (line 8), as these fields are set by replicas. Finally, the client signs and sends the message to replica r_i.

Upon sending a *RING* message to the entry replica, the client starts the timer T_{ring}, set to expire after period $(2(3f + 1) + 2)\Delta$ (line 2). The expiration time is set to match the maximum response delay when the system is synchronous. If the timer expires before the client receives a response, the client will panic (line 14) and notify replicas. We assume that clients wait for the replica's response before issuing new requests.

Step R2. *Upon receiving a request (a RING message), replica r_i updates the message fields and forwards the message to its successor (Algorithm A.2, lines 29–45)*

Upon receiving (line 29 of Algorithm A.2) a $\langle RING, req, sn', RASET, MACSET\rangle$ message from the predecessor (or from the client in case replica is the *entry replica* for the request), replica r_i first checks whether r_i can successfully authenticate and accept the message. The check consists of several conditions (lines 30–35):

1) The first $f + 1$ replicas check whether some Ring Authenticator (*RA*) in the *RASET* contains a valid authenticator (a signature) for the request req, generated by the client;
2) every replica r_i checks whether *RASET* contains a RA with a valid MAC for every replica r_j in the predecessor set, $\overleftarrow{r_j}$, authenticating req and sn'. (Note: by definition, the predecessor set of the entry replica is empty $\overleftarrow{r}_{req.entry} = \emptyset$);
3) every replica accepts a *RING* message only if the client's timestamp of the request req ($req.t_c$) is higher than the last seen (and executed) request timestamp from that client ($lastreq_i[req.c].t_c$);
4) finally, if the sequence number sn' of the message is either equal to nil or $sn_i + 1$, replica accepts the *RING* message.

If these checks succeed, then replica r_i proceeds to processing the request. First, every replica stores req in $pending[req.c]$. The sequencer increments the local sequence number sn_0, and sets $sn' = sn_0$ (line 38). Otherwise, if replica is not the sequencer, and sn' is not nil, then the replica stores sn' into the local variable sn_i (line 19). Moreover, if sn' is not nil, then every replica:

1) executes the request and stores the reply (lines 14–25),
2) appends req to its local history LH_i (line 18), and
3) updates the data that reflects the execution of last request by the client $req.c$ by storing req and sn_i into corresponding data structures: $lastsn_i[req.c]$, and $lasthist_i[req.c]$ (line 19).

Further, each replica updates the information about the last known request from the client, by storing the request into $lastreq_i[req.c]$ (line 14).

After processing req, replica r_i forwards the request unless r_i is the exit replica. Replica r_i sends the *RING* message containing req and sn', as well as updated set *RASET* (calculated in line 40). Replica r_i updates *RASET* by removing all the MACs destined to itself (line 28 of Algorithm A.4), and by adding RA authenticating the tuple $\{Type^{REQ}, req, sn'\}$ (line 36 of Algorithm A.4) for every replica in its successor set, $\overrightarrow{r_i}$. The first element of the tuple can have values of either $Type^{REQ}$ or $Type^{ACK}$ and serves as a protection against copy attacks. Thus no replica can forge *ACK* messages, by using RA of the original *RING* message. Finally, replica r_i sends the *RING* message, containing req, sn', *RASET*, and $\emptyset$ in place of *MACSET* (line 43).

If replica r_i is the exit replica, instead of forwarding the request, the replica generates an acknowledgement – an *ACK* message (line 42). The replica first updates the *RASET*, this time authenticating the tuple $\{Type^{ACK}, req, sn'\}$ (line 37 of Algorithm A.4). Finally, replica r_i sends the *ACK* message to the successor. The *ACK* message initially contains the following fields: $D(req)$, $req.c$, sn', *RASET*, and empty set (as the *MACSET* field).

RING message verification failure: If a verification of the received *RING* message fails, a correct replica r_i can safely discard the received message.

Step R3. *Upon receiving an acknowledgement (an ACK message), replica r_i updates the message fields and forwards the message to its successor. (Lines 48–64 of Algorithm A.2)*

Replica r_i receives $\langle ACK, D, sn', c', RASET, MACSET\rangle$ from the predecessor, and processes the message in a similar fashion as the *RING* message. First, the replica checks whether it can successfully authenticate and accept the message (lines 49–54). The conditions differ from the *RING* message case. Namely:

1) if there is no stored request req in $pending$ list (line 50), corresponding to the *ACK* message, the message is discarded; otherwise, req is taken from $pending$ list,

2) if *RASET* contains a RA with a valid MAC for every replica r_j in the predecessor set $\overleftarrow{r_i}$, authenticating req and sn'; otherwise, the replica discards the message,
3) finally, every replica accepts the *ACK* message only if the sequence number of the message (sn') equals $sn_i + 1$ (line 54).

If the request was not executed previously by replica r_i, the replica does the same steps as when handling the *RING* message:

1) appends req to replica's local history LH_i, and
2) updates the data that reflects the last request by the client $req.c$ by storing req, sn_i, LH_i into corresponding data structures: $lastreq_i[req.c]$, $lastsn_i[req.c]$, and $lasthist_i[req.c]$

.

After executing the request, replica r_i forwards the *ACK* message. Beforehand, the replica updates the *RASET*, and the *MACSET* fields of the message. The *MACSET* is effectively updated only by the $f + 1$ predecessors of the entry replica (line 59). These replicas authenticate the pair $\{req, D(LH_i)\}$, where $D(LH_i)$ denotes the digest of the replica's local history. If replica r_i is not the the exit replica, the replica forwards the *ACK* message, containing $D(req)$, sn', $req.c$, *RASET*, and *MACSET* (as seen in line 63). Otherwise, replica r_i sends a *REPLY* message (a reply) to the client named in $req.c$, containing replica's full local history LH_i (line 62).

ACK message verification failure: If any of the check conditions does not hold, the replica may safely discard the request. At this point, there is no MAC from the client in the *RASET*, hence the replica may assume that some of the predecessors are Byzantine, and simply discard the request.

Step R4a. *Upon receiving the REPLY message from the exit replica before the expiration of the timer, if the client successfully verifies the reply, the client commits the request. (Lines 10–12 of Algorithm A.1)*

If client c receives the $\langle\langle REPLY, req, *, *, *, MACSET\rangle, LH\rangle$ message (line 10) from the exit replica ($r_{entry \ominus 1}$), that can be successfully verified, then the client commits request req with *Ring commit history* LH (line 12). A successful verification (described at line 11) means that the set *MACSET* contains valid MACs from the last $f + 1$ replicas in the ring order (predecessors of the exit replica), destined to client c, that authenticate the pair $\langle req, d\rangle$, where d is the digest of the history ($d = D(LH)$).

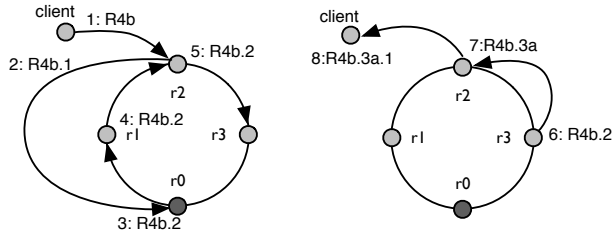

Figure 16. A client invokes a request, but does not receive a reply. The client panics, sending a *PANIC* message to all replicas. Majority of replicas successfully answer. For clarity, we present only the actions of replica r_2.

Step R4b. *The client **does not** receive the RING message from the exit replica, and/or the client can not verify the message, before the expiration of the timer. (Lines 14–15 of Algorithm A.1)*

If the client does not receive the message before timer T_{Ring} expires, or the message cannot be verified, the client *panics* (line 15). Client c sends a $\langle PANIC, req_{\sigma_c}\rangle$ message to all replicas. The *PANIC* message is digitally signed by the client. Moreover, the client periodically resends the *PANIC* message to the replicas, until the client commits or aborts the request.

Step R4b.1. *A replica receives a PANIC message from the client, and the replica retries Ring on behalf of the client. (Lines 67–75 of Algorithm A.2)*

Replica r_i, on receiving a $\langle PANIC, req_{\sigma_{req.c}}\rangle$ message (line 67), if the message contains a valid signature, tries to commit the request by invoking Steps R1-R4a on behalf of the client. Toward that end, replica r_i acts as a client and sends the $\langle OBR, r_i, req_{\sigma_{req.c}}, sn_{OBR} = nil, RASET_{OBR} = \emptyset, MACSET_{OBR} = \emptyset\rangle_{\mu_{r_i,r_0}}$ message to the sequencer (line 74).

Subsequently, the replica starts the timer $T_{OBR_{req}}$. If the timer expires before the replica receives a response for the *OBR* request, the replica will abort the protocol (lines 121–124 of Algorithm A.2).

The *OBR* message is similar to the *RING* (and the *ACK*) message, albeit some differences:

- the *OBR* contains an additional field which the replica r_i (the originator of the *OBR* request) populates with its own ID.
- the *RASET* field is initially empty, as the client's signature is included in the message. Note that the replica authenticates the message with a MAC.

Finally, it is important to note that a correct replica r_i sends an *OBR* request to the sequencer *iff* the request is not old (the $req.t_c$ field of the *PANIC* message is greater or equal than $lastreq_i[req.c].t_c$, as seen at line 69). Moreover, replica r_i abandons waiting for the *RING* message from replica r_{3f} (predecessor of the sequencer), and cancel its timer, if $tr_i[c]$ becomes greater than $req.t_c$ (suggesting there is a new request from client c).

Step R4b.2. *A replica receives an OBR message and processes the message in a similar way as both the RING (Step R2) and ACK message (Step R3.). (Lines 86–107 of Algorithm A.2)*

Replica r_i first checks whether it can successfully authenticate and accept a message, upon receiving the $\langle OBR, r_k, req, sn', SIGSET, MACSET\rangle$ message from the predecessor (or from replica r_k in case r_i is the sequencer). This check consists of several conditions:

1) replica r_i checks whether the client's signature matches the request (line 90);
2) replica r_i checks (at line 90) whether the *SIGSET* contains a RA with a valid MAC for every replica r_j in the predecessor set, $\overleftarrow{r_j}$, authenticating req and sn'. (Note: the predecessor set for the sequencer in this case is empty $\overleftarrow{r_0} = \emptyset$);
3) the replica accepts the *OBR* message only if the client's timestamp of request req ($req.t_c$) is greater or equal than the timestamp of the last seen (and executed) request from the client ($lastreq_i[req.c].t_c$), as shown at line 91;
4) finally, every replica except the sequencer accepts the *OBR* message if the sequence number sn' of the message is equal to $sn_i + 1$.

If these checks succeed, then replica r_i proceeds to the execution part of processing (line 99). The replica only executes the request if the request is new (i.e., $req.t_c$ is higher than the last stored request from the client, line 15 of Algorithm A.2). Otherwise, the replica takes the stored response and the sequence number, and skips the next step (line 96).

Like with the *RING* and the *ACK* messages, the sequencer updates the local sequence number sn_0, and sets $sn' = sn_0$ (line 98). Otherwise, the replica stores sn' into replica's local variable sn_i. Moreover, every replica r_i: (1) appends req to its local history LH_i (shown at line 100), and (2) updates the data that reflects the last request by the client $req.c$ by storing req, sn_i and LH_i into $lastreq_i[req.c]$, $lastsn_i[req.c]$, and $lasthist_i[req.c]$, respectively . Finally, every replica stores req in $pending[req.c]$.

Upon executing req, the replica forwards (line 106) the *OBR* message, unless the replica is the predecessor of the sequencer ($r_i \neq r_{3f}$). In that case (line 105), replica r_i sends the reply back to replica r_k (indicated as one field of the *OBR* message). Beforehand, the replica updates the *RASET*.

Step R4b.3a. *The replica commits the request on behalf of the client and forwards the commit history to the client. (Lines 111–117 of Algorithm A.2)*

If r_i receives an *OBR* message from the predecessor of the sequencer (line 111), containing MACs for the pair $\langle req, D(h)\rangle$ for the last $f + 1$ replicas in the *MACSET*, as well as the full history h (line 114), then replica r_i simply sends the $\langle GET\_A\_GRIP, h, req\rangle_{\mu_{r_i,req.c}}$ message to client named in $req.c$ (line 116). We say that r_i *commits* the *OBR* for req with the history h.

To counter for possible message losses, if a replica receives repeated *PANIC* messages for req after committing the *OBR* for req, the replica replies to these messages by re-sending the *GET_A_GRIP* message to the client.

Step R4b.3a.1. *The client receives $f+1$ GET_A_GRIP messages containing the same history and commits the request. (Lines 17–18 of Algorithm A.1)*

If the client received $f + 1$ $\langle GET\_A\_GRIP, h, req\rangle$ messages from different replicas, with the same history, the client commits the request by returning $Commit(req, h)$.

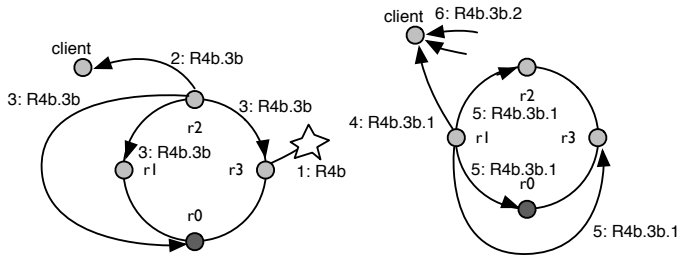

Figure 17. Alternative execution to one shown on Figure 16: Replica r_2 cannot commit the request. The replica aborts, and stops the protocol.

Step R4b.3b. *The replica **does not** commit the request on behalf of the client, stops processing new request, and sends a signed history to the client. (Lines 121–124 of Algorithm A.2)*

If replica r_i does not receive the *OBR* request before the expiration of the timer, the replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests (line 122); (b) sends a signed local history to client $req.c$ using an $\langle ABORT, LH_{i\sigma_{r_i}}, req.t_c, r_i\rangle_{\mu_{r_i,req.c}}$ message (line 123); and (c) stops all OBR timers . In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica (line 124). Again, to counter possible message losses, we assume that r_i periodically retransmits the $STOP$ message.

Step R4b.3b.1. *The replica receives a $STOP$ message from some other replica, stops processing new requests, and sends a signed history to the client. (Lines 127–129 of Algorithm A.2)*

Replica r_i now aborts all clients requests, similarly as in the Step R4b.3b. Replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests; (b) sends a signed local history to all clients referenced in the active OBR timers[8], using an $\langle ABORT, LH_{i\sigma_{r_i}}, req'.t_c, r_i\rangle_{\mu_{r_i,req'.c}}$ message; and (c) stops all OBR timers . In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica. Again, to counter possible message losses, we assume that r_i periodically retransmits the $STOP$ message.

Step R4b.3b.2. *A client receives $2f + 1$ matching $ABORT$ messages, extracts the abort history, and aborts the request. (Lines 20–22 of Algorithm A.1)*

A matching $ABORT$ message for a $\langle PANIC, req\rangle$ message is any $ABORT$ message with a matching request identifier $req.t_c$. When a client receives a matching $ABORT$ message from $2f + 1$ different replicas, the client extracts the abort history AH in the following way:

- the client generates the history AH_1 such that AH_j equals the value that appears at position $j \geq 1$ of $f + 1$ different histories LH_i received in the $ABORT$ messages. If such value does not exist for a position k, then AH_1 does not contain a value at position k or higher.
- the longest prefix AH_2 of AH_1 is selected such that no request appears in AH_2 twice.
- if $req = \langle o, t_c, c\rangle$ does not exist in AH_2, the request is appended to AH_2. The resulting sequence is an abort history AH.

Then, client c aborts req by returning $Abort(req, AH)$. To prove validity of the AH, the abort history is accompanied by the set of $2f + 1$ $ABORT$ messages.

F. Ring$^+$ algorithm steps

Ring$^+$ handles the *RING*, the *ACK*, and the *PANIC* messages is the same way as Ring$^-$. For clarity, we present only the steps related to handling the *OBR* messages – the main difference between Ring$^-$ and Ring$^+$.

Step R$^+$4b. *The client **does not** receive the RING message from the exit replica, and/or the client can not verify the message, before the expiration of the timer. (Lines 14–15 of Algorithm A.1)*

If the client does not receive the message before timer T_{Ring} expires, or the message cannot be verified, the client *panics*.

[8]there is a timer for every outstanding *OBR* request req'

Client c sends a $\langle PANIC, req_{\sigma_c} \rangle$ message to all replicas. The *PANIC* message is digitally signed by the client. Moreover, the client periodically resends the *PANIC* message to the replicas, until the client commits or aborts the request.

Step R⁺4b.1. *A replica receives a PANIC message from the client, and the replica retries Ring on behalf of the client.*

Replica r_i, on receiving a $\langle PANIC, req_{\sigma_{req.c}} \rangle$ message, if the message contains a valid signature, tries to commit the request by invoking Steps R1-R4a on behalf of the client. Toward that end, replica r_i acts as a client and sends the $\langle OBR, r_i, req_{\sigma_{req.c}}, sn_{OBR} = nil, RASET_{OBR} = \emptyset, MACSET_{OBR} = \emptyset \rangle_{\sigma_{r_i}}$ message to the sequencer. Subsequently, the replica starts the timer $T_{OBR_{req}}$. If the timer expires before the replica receives a response for the *OBR* request, the replica will abort the protocol.

The *OBR* message is similar to the *RING* (and the *ACK*) message, albeit some differences:

- the *OBR* contains an additional field which the replica r_i (the originator of the *OBR* request) populates with its own ID.
- the *RASET* field is initially empty, as the client's signature is included in the message. Note that the replica authenticates the message with the signature.

Finally, it is important to note that a correct replica r_i sends an *OBR* request to the sequencer iff the request is not old (the $req.t_c$ field of the *PANIC* message is greater or equal than $lastreq_i[req.c].t_c$). Moreover, replica r_i abandons waiting for the *RING* message from replica r_{3f} (predecessor of the sequencer), and cancel its timer, if $tr_i[c]$ becomes greater than $req.t_c$ (suggesting there is a new request from client c).

Step R⁺4b.2. *A replica receives an OBR message and processes the message, possibly executing the request. The replica then forwards the message to $f + 1$ successors. (Lines 9–34 of Algorithm* ***??****)*

Replica r_i first checks whether it can successfully authenticate and accept a message, upon receiving the $\langle OBR, r_k, req, sn', RASET, MACSET \rangle$ message from one of its predecessors (or from replica r_k in case r_i is the sequencer). This check consists of several conditions:

1) replica r_i checks whether the client's signature matches the request (line 12);
2) the replica accepts the *OBR* message only if the client's timestamp of request req ($req.t_c$) is greater or equal than the timestamp of the last seen (and executed) request from the client ($lastreq_i[req.c].t_c$, shown at line 10);
3) finally, every replica except the sequencer accepts the *OBR* message if the sequence number sn' of the message is equal to $sn_i + 1$ (line 14).

If these checks succeed, then replica r_i collects all signatures in the *OBR* message, and verifies each in turn (line 16). If at least one of the signatures has not been seen previously, then the replica continues processing the request. Otherwise, the replica drops the request (line 18).

If there are more than $2f + 1$ valid signatures, the replica proceeds to the execution part of processing (line 20). The replica only executes the request if the request is new (i.e., $req.t_c$ is higher than the last stored request from the client). In this case, the sequencer updates the local sequence number sn_0, and sets $sn' = sn_0$, while other replicas just store sn' into replica's local variable sn_i (line 25). Otherwise, if the request is old, the replica takes the stored response and the sequence number, and skips the execution step.

Each replica executes the request, if the request is new. Every replica r_i: (1) appends req to its local history LH_i, and (2) updates the data that reflects the last request by the client $req.c$ by storing req, sn_i and LH_i into $lastreq_i[req.c]$, $lastsn_i[req.c]$, and $lasthist_i[req.c]$, respectively . Finally, every replica stores req in $pending[req.c]$.

Upon executing req, replica adds its own signature to the message (line 29). Then, the replica forwards the request to $f + 1$ successor (line 31).

If there were less than $2f + 1$ valid signatures in the request, the replica adds its own signature, and forwards the updated request to $f + 1$ successor (lines 33–34).

Step R⁺4b.3a. *The replica commits the request on behalf of the client and forwards the commit history to the client. (Lines 29–31)*

If r_i receives an *OBR* message with at least $2f + 1$ valid signatures from other replicas, then the replica simply sends the $\langle GET\_A\_GRIP, h, req \rangle_{\mu_{r_i, req.c}}$ message to client named in $req.c$ (line 30). We say that r_i *commits* the *OBR* for req with the history h.

To counter for possible message losses, if a replica receives repeated *PANIC* messages for req after committing the *OBR* for req, the replica replies to these messages by re-sending the *GET_A_GRIP* message to the client.

Step R⁺4b.3a.1. *The client receives $f+1$ GET_A_GRIP messages containing the same history and commits the request. (Lines 17–18 of Algorithm A.1)*

If the client received $f + 1$ $\langle GET\_A\_GRIP, h, req\rangle$ messages from different replicas, with the same history, the client commits the request by returning $Commit(req, h)$.

Step R⁺4b.3b. *The replica **does not** commit the request on behalf of the client, stops processing new request, and sends a signed history to the client. (Lines 121–124 of Algorithm A.2)*

If replica r_i does not receive the *OBR* request with at least $2f + 1$ valid signatures from other replicas, before the expiration of the timer, the replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring⁺ to stop accepting requests (line 122); (b) sends a signed local history to client $req.c$ using an $\langle ABORT, LH_{i\sigma_{r_i}}, req.t_c, r_i\rangle_{\mu_{r_i,req.c}}$ message (line 123); and (c) stops all OBR timers . In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica (line 124). Again, to counter possible message losses, we assume that r_i periodically retransmits the $STOP$ message.

Step R⁺4b.3b.1. *The replica receives a $STOP$ message from some other replica, stops processing new requests, and sends a signed history to the client. (Lines 127–129 of Algorithm A.2)*

Replica r_i now aborts all clients requests, similarly as in the Step R⁺4b.3b. Replica: (a) stops accepting new *RING*, *ACK*, and *OBR* messages, by setting a global flag which forces Ring to stop accepting requests; (b) sends a signed local history to all clients referenced in the active OBR timers[9], using an $\langle ABORT, LH_{i\sigma_{r_i}}, req'.t_c, r_i\rangle_{\mu_{r_i,req'.c}}$ message; and (c) stops all OBR timers . In addition, the replica sends $\langle STOP, req\rangle_{\mu_{r_i,r_j}}$ to every other replica. Again, to counter possible message losses, we assume that r_i periodically retransmits the $STOP$ message.

Step R⁺4b.3b.2. *A client receives $2f + 1$ matching $ABORT$ messages, extracts the abort history, and aborts the request. (Lines 20–22 of Algorithm A.1)*

A matching $ABORT$ message for a $\langle PANIC, req\rangle$ message is any $ABORT$ message with a matching request identifier $req.t_c$. When a client receives a matching $ABORT$ message from $2f + 1$ different replicas, the client extracts the abort history AH in the following way:

- the client generates the history AH_1 such that AH_j equals the value that appears at position $j \geq 1$ of $f + 1$ different histories LH_i received in the $ABORT$ messages. If such value does not exist for a position k, then AH_1 does not contain a value at position k or higher.
- the longest prefix AH_2 of AH_1 is selected such that no request appears in AH_2 twice.
- if $req = \langle o, t_c, c\rangle$ does not exist in AH_2, the request is appended to AH_2. The resulting sequence is an abort history AH.

Then, client c aborts req by returning $Abort(req, AH)$. To prove validity of the AH, the abort history is accompanied by the set of $2f + 1$ $ABORT$ messages.

Both operational modes of Ring are an implementation of ABSTRACT , each with their own *Non-Triviality* property. Non-Triviality property in ABSTRACT model defines the conditions under which a protocol should commit client requests.

For the sake of brevity, we will call the normal mode Ring the Ring⁻ , while the resilient mode will be called Ring⁺ . Next, we present the properties every ABSTRACT instance should satisfy, followed with the correctness proofs for Ring⁻ and Ring⁺ .

Definition *(Ring⁻ Non-Triviality)*

If (a) a correct client c invokes a request m, (b) there are no replica failures, and (c) the set of replicas (Σ) is synchronous, then client c commits m.

Definition *(Ring⁺ Non-Triviality)*

[9]there is a timer for every outstanding *OBR* request req'

If (a) a correct client c invokes a request m, (b) the sequencer is not faulty, and (c) the set of replicas (Σ) is synchronous, then client c commits m.

1) *(Validity)* In every commit/abort history, no request appears twice and every request is a valid request, or an element of a valid init history.
2) *(Termination)* If a correct client c invokes a valid request m, then c eventually commits or aborts m.
3) *(Non-Triviality)* If a correct client c invokes a valid request m and some predicate NT holds, then c commits m.
4) *(Init Order)* Any common prefix[10] of valid init histories is a prefix of any commit or abort history.
5) *(Commit Order)* Let h and h' be any two commit histories: either h is a prefix of h' or vice versa.
6) *(Abort Order)* Every commit history is a prefix of every abort history.
7) *(Switching Monotonicity)* For every *Abstract* instance i, $i < next(i)$.

In addition, we say that correct replica r_j executes req *at position* pos if $sn_j = pos$ when r_j executes req. Before proving ABSTRACT properties, we first prove a set of auxiliary lemmas.

Definition *(Ring order)* The *ring order* defines the total order of replicas on the ring. We say that this ordering *starts* at a particular replica r_j, and define total order operation such that: $j < j+1 < \cdots < j+3f$.

Figure 18 shows Ring's circular topology. For the ring order which starts at replica r_0, we have the following relation: $r_0 < r_1 < r_2 < r_3$. On the other hand, if the order starts at r_2, we would have: $r_2 < r_3 < r_0 < r_1$.

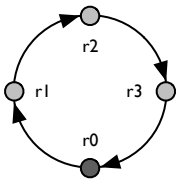

Figure 18. Ring circular topology

G. Ring⁻ correctness proof

In this section, we prove that Ring⁻ implements ABSTRACT with *Ring⁻ Non-Triviality*. To do so, we need to show that Ring⁻ satisfies properties listed in Section F. First, we prove some necessary lemmas.

Lemma A.1: Let r_j be a correct replica and LH_j^{req} the state of LH_j when r_j executes req. Then, LH_j^{req} remains a prefix of LH_j forever.

Proof: A correct replica r_j modifies its local history LH_j only in *Step R2* or *Step R3* or *Step R4b.2* by sequentially appending requests to LH_j. Hence, LH_j^{req} remains a prefix of LH_j forever. ■

Lemma A.2: If a correct replica r_i accepts a request req (via the *RING* message), at time t_1, then all correct replicas r_j ($req.entry \leq j < i$)[11] accepted the request before t_1. Note that we do not discuss execution of the request. If replica *accepts* a request, it means that the replica verified the request, and stored it in some internal structure.

Proof: By contradiction, assume the lemma does not hold and fix r_j to be the first correct replica that accepts req, such that there is a correct replica r_x ($x < j$) that never accepts req. We say that r_j *animates* req. Since *RING* messages are authenticated using RAs, r_j accepts req only if r_j receives a *RING* message with MACs authenticating req from all replicas from $\overleftarrow{r_j}$, i.e., only *after* all correct replicas from $\overleftarrow{r_j}$ have accepted req. If $r_x \in \overleftarrow{r_j}$, r_x must have accepted req — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica which animates req, since any correct replica (at least one) from $\overleftarrow{r_j}$ animates req — a contradiction. ■

Lemma A.3: If a correct replica r_i accepts a request req, then the request was invoked by a client.

Proof: By contradiction, assume that some correct replica accepted a request not invoked by any client and let r_j be the first correct replica to accept such a request req' in Step R2. In case $j \in \{req'.entry \ldots req'.entry \oplus (f+1)\}$, r_j accepts the req' only if r_j receives a *RING* message with a signature from the client, i.e., only if some client invoked req, or if req is contained in some valid *INIT* history. On the other hand, if j is not in that set, Lemma A.2 yields a contradiction with our assumption that r_j is the first correct replica to accept req'. ■

[10]In this paper, unless explicitly stated otherwise, "prefix" refers to a non-strict prefix.

[11]If not stated otherwise, we use the ring order.

Lemma A.4: If a correct replica receives a non-nil sequence number (sn) for a request req, either through a ***RING***, an ***ACK***, or ***OBR*** message, that sn was generated by the sequencer.

Proof: By construction. The guard conditions in Step R2, and R3 prevent such case, along with the check of Ring Authenticators. ■

Lemma A.5: If a correct replica r_i executes a request req, at position sn, at time t_1, then all correct replicas r_j ($0 \leq j < i$) executed the request at position sn before t_1. Note that we refer to the ring order.

Proof: By contradiction, assume the lemma does not hold and fix r_j to be the first correct replica that executes req (at position sn), such that there is a correct replica r_x ($x < j$) that never executes req. We say that r_j is the first replica for which req *skips*. Since ***RING*** (and ***ACK***) messages are authenticated using RAs, r_j executes req at position sn only if replica r_j receives a ***RING*** (or an ***ACK***) message with MACs authenticating the pair $\langle req, sn \rangle$[12] from all replicas from $\overleftarrow{r_j}$, i.e., only *after* all correct replicas from $\overleftarrow{r_j}$ have accepted req. If $r_x \in \overleftarrow{r_j}$, r_x must have accepted req — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica at which req skips, since at any correct replica (at least one) from $\overleftarrow{r_j}$ req skips — a contradiction. The similar reasoning applies to handling an ***OBR*** request.

Note that the sequence number sn associated by the sequencer is indeed equivalent to the position at which a replica executes req, since (1) if the replica is the sequencer, sn is incremented by one, and (2) if the replica is not the sequencer, the replica accepts req with associated sn', only if $sn' = sn + 1$ (Step R2, R3, and R4a) ■

Lemma A.6: If a correct replica r_i receives an ***ACK*** for request req, at position sn and time t_1, then all correct replicas r_j ($req.entry \leq j < i$) executed request req at position sn, before t_1. Note that we use the ring order, which starts at $req.entry$.

Proof: If replica r_i receives a valid ***ACK***, that means that all correct replicas have received the request (execution condition in Step R3, and Lemma A.2). From Step R3, and Lemma A.5, we have that all correct replicas r_j ($0 \leq j < i$) executed the request. Let fix the ring order, so that the sequence starts from 0, and ends at $3f$. We consider two cases:

1) if $0 \leq req.entry < i$, then the claim follows immediately from Lemma A.5;
2) if $0 \leq i < req.entry$, from Step R2, we get that ***ACK*** was generated at $req.entry \ominus 1$. It holds that $0 \leq i \leq req.entry \ominus 1$. From Step R3, by construction, we have that all correct replicas r_x ($x \in req.entry \ominus 1 \ldots i$) have received the ***ACK***. From the previous case, we have that request is executed on all correct replicas r_k ($req.entry \leq k < 0$), and from Lemma A.5 we have that request is executed on all correct replicas r_j ($0 \leq j < i$).

■

Lemma A.7: If a benign client c commits request req with history h (at time t_1), then all correct replicas in Σ_{last}^{req} execute req (before t_1) and the state of their local history upon executing req is h.

Proof: To prove this lemma, notice that a correct replica $r_j \in \Sigma_{last}^{req}$ generates a MAC for the client authenticating req and $D(h')$ for some history h' (Step R2, or Step R3): (1) only after r_j executes req and (2) only if the state of LH_j upon execution of req equals h'. Moreover, by Step R2/R3, no correct replica executes the same request twice. By Step R4a, a benign client (resp., a replica) cannot commit req with h unless it receives a MAC authenticating req and $D(h')$ from every correct replica in Σ_{last}. From Lemma A.5 we get the claim. By Step R4b.3a.1, a benign client (resp., a replica), cannot commit req with h unless it receives a ***GET_A_GRIP*** message with a MAC authenticating req and $D(h')$ from every correct replica in Σ_{last}. Again, from Lemma A.5, we get the claim.

■

Well-formed commit indications. By Step R4a, in order to commit a request the client needs to receive MACs authenticating $Digest_{LH} = D(h')$ for some history h' and a reply digest from all replicas from Σ_{last}^{req}, including at least one correct replica. By Step R3, the digest of the reply sent by a correct replica is $D(rep(h'))$. Hence, h' is exactly the commit history h and is uniquely defined due to our assumption of collision-free digests.

Moreover, since a correct replica executes an invoked request before sending an ***ACK*** message in Step R3 (or a ***GET_A_GRIP*** message in Step R4b.3a), it is straightforward to see that if req is committed with a commit history h_{req}, then req is in h_{req}.

■

Validity. For any request req to appear in an abort (resp., commit) history h, at least $f + 1$ replicas must have sent h (resp., a digest of h) in Step R3 (or in Step R4b.3a.1), such that $req \in h$. Hence, at least one correct replica executed req.

Directly from Lemma A.6 we observe that all correct replicas execute only requests invoked by clients.

Moreover, by Step R2 or Step R3 or Step R4b.1, no replica executes the same request twice (every replica maintains a list of last seen identifiers — $t_j[c]$). Hence, no request appears twice in any local history of a correct process, and consequently,

[12] where sn is not nil

no request appears twice in any commit history. In the case of abort histories, no request appears twice by construction. ■

Termination. By assumption of a quorum of $2f+1$ correct replicas and fair-loss links: (1) correct replicas eventually receive a PANIC message sent by a correct client c (in Step R4b) and (2) c eventually receives $2f+1$ abort messages from correct replicas (sent in Step R4.2b). Hence, if correct client c panics, the client eventually aborts invoked request req, in case c does not commit req beforehand.

Moreover, to see that a committed request req must be in its commit history h_{req}, notice that the client needs to receive a MAC for the same local history digest $D(h_{req})$ from all $f+1$ replicas from Σ_{last}^{req} including at least one correct replica r_j. By Step R2/R3, r_j executes req and appends the request to the replica's local history LH_j before authenticating the digest of LH_j; hence, $req \in h_{req}$. By Step R4b.2, replica r_j executes req and appends the request to replica's local history LH_j. Further, the replica embeds the history in the *OBR* message. Only after these steps, replica r_j authenticates the digest of LH_j, prior to sending the *GET_A_GRIP* message to the client. Hence, $req \in h_{req}$.

■

Commit Order. Assume, by contradiction, that there are two committed requests req (by a benign client c) and $req' \neq req$ (by a benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. By Lemma A.7, all correct replicas in Σ_{last}^{req} (resp. $\Sigma_{last}^{req'}$) executed request req (resp. req') with history h_{req} (resp. $h_{req'}$). Let r^{req} be the first correct replica in Σ_{last}^{req}, and let $r^{req'}$ be the first correct replica in $\Sigma_{last}^{req'}$. There are two distinct cases:

- these replicas are the same ($r^{req} = r^{req'}$). A contradiction with Lemma A.1.
- one preceeds the other, in ring order which starts from the sequencer. Without loss of generality, we assume $r^{req} < r^{req'}$. By Lemma A.5, r^{req} executed all requests $r^{req'}$ has had executed, at the same position. A contradiction. ■

Abort Order. Assume, by contradiction, that there is a committed request req_C (by some benign client) with commit history h_{req_C} and an aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. By Lemma A.7 and the assumption of at most f faulty replicas, all correct replicas (at least one) from $\Sigma_{last}^{req_C}$ execute req_C and their state upon executing req_C is h_{req_C}. Let $r_j \in \Sigma_{last}^{req_C}$ be a correct replica with the highest (w.r.t. the ring order which starts at $req_C.entry$) index among all replicas in $\Sigma_{last}^{req_C}$. By Lemma A.6, all correct replicas r_k ($req_C.entry \leq k < j$) execute all the requests in h_{req_C} at the same positions these requests have in h_{req_C}.

In addition, a correct replica executes all requests in h_{req_C} before sending any $ABORT$ message (Step R4b.3b.1); indeed, before sending any $ABORT$ message, a correct replica must stop further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an $ABORT$ message, h_{req_C} is a prefix of LH_j.

Finally, by Step R4b.3b.2, a client that aborts a request waits for $2f+1$ $ABORT$ messages including at least $f+1$ from correct replicas. By construction of the abort history every commit history, including h_{req_C} is a prefix of every abort history, including h_{req_A}, a contradiction.

■

Init Order. Under the constraint that, if a replica's local history is empty, the first request to which the sequencer can assign the sequence number, and the first request a replica may execute, must be an *INIT* request, then we get that replicas initialize their local histories before sending any *RING*, *ACK* or *ABORT* request.

Since any common prefix CP of all valid init histories is a prefix of any particular init history IH, CP is a prefix of every local history sent by a correct replica in an *RING* or $ABORT$ message. Init Order for commit histories immediately follows. In the case of abort histories, notice that at least out of $2f+1$ $ABORT$ messages received by a client on aborting a request in Step R4.2b.2, at least $f+1$ are sent by correct processes and contain local histories that have CP as a prefix. Hence, by Step R4.2b, CP is a prefix of any abort history.

■

Non-Triviality. Non-Triviality relies on the fact that client's timer triggered in Step R1 is set such that it does not expire in case when the set of replicas, including the client, is synchronous.

Assume by contradiction that there is a correct client c that panics and denote the first such time by t_{PANIC}. Client c has invoked request req at $t = t_{PANIC} - (2(3f+1)+2)\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that: (i) by $t+\Delta$ the entry replica receives req and takes Step R2, and (ii) by time $t + 3f\Delta < t_{PANIC}$ all replicas take Step R2 for req, and (iii) by time $t + (2(3f+1)-1)\Delta < t_{PANIC}$ all replicas take Step R3. Since the sequencer is correct then all replicas execute all requests received before t_{PANIC} in the same order (established by the sequence numbers assigned by the sequencer). Hence, by $t + (2(3f+1)+2)\Delta = t_{PANIC}$, c receives $f+1$ identical replies (Step R4a), commits request req and never panics. A contradiction.

In addition, a correct replica r_i executes Step R4b.1b and stops appending new requests, only if r_i fails to commit an OBR request for a *RING* message signed by some client. Since such an OBR request cannot raise a verification failure, r_i can fail to commit such request only in case asynchrony in the set of replicas, or in case some replica fails.

■

H. Ring⁺ correctness proof

In this section, we prove that Ring⁺ implements ABSTRACT with *Ring⁺ Non-Triviality*. First, we prove a couple of auxilary lemmas.

Lemma A.8: If a correct replica r_i receives a request req (via the *OBR* message), at time t_1, then all correct replicas r_j $(0 \leq j < i)$ received the request before t_1.

Proof: By contradiction, assume the lemma does not hold and fix r_j to be the first correct replica that receives req, such that there is a correct replica r_x $(x < j)$ that never receives req. We say that r_j is the first replica for which req *obr-skips*. Correct replica sends a request to its $f+1$ successors. Hence, If $r_x \in \overleftarrow{r_j}$, r_x must have received req — a contradiction. On the other hand, if $r_x \notin \overleftarrow{r_j}$, then r_j is not the first replica for which req *obr-skips*, since any correct replica (at least one) from $\overleftarrow{r_j}$ obr-skips req — a contradiction. ■

Lemma A.9: When processing *OBR* requests, after at most $min(f+1, 4)$ communication steps from the time the non-malicious sequencer receives an *OBR* request, all replicas will receive the message.

Proof: By contradiction, assume that it takes more than four steps for all replicas to receive the request. Let R_1 be the last replica in the ring order, to receive the request in the first step. Similarly, let R_2 (resp. R_3, R_4, R_5) be the last replica to receive the request in the second (resp. third, fourth, fifth) step. Let d_0 be the distance between r_0 and R_1. Likewise, let d_1 be the distance between R_1 and R_2, d_2 be the distance between R_2 and R_3, etc... We have the following equations:

$$d_0 + d_1 + d_2 + d_3 + d_4 < 3f + 1 \tag{1}$$
$$1 \leq d_0, d_1, d_2, d_3, d_4 \leq f + 1 \tag{2}$$
$$f + 1 \leq d_0 + d_1 \tag{3}$$
$$f + 1 \leq d_1 + d_2 \tag{4}$$
$$f + 1 \leq d_2 + d_3 \tag{5}$$
$$f + 1 \leq d_3 + d_4 \tag{6}$$
$$2f + 1 \leq d_0 + d_1 + d_2 \tag{7}$$
$$2f + 1 \leq d_1 + d_2 + d_3 \tag{8}$$
$$2f + 1 \leq d_2 + d_3 + d_4 \tag{9}$$

Equation 1 states that after five communication steps we reach all correct nodes on the ring (at most $3f+1$). Equations 3-6 state that a replica reached in two steps, could not have been reached in a single step. Similarly, Equations 7-9 state that a replica reached in three steps could not have been reached in less steps. From Equations 7 and 6 we get a contradiction with Equation 1:

$$(d_0 + d_1 + d_2) + (d_3 + d_4) \geq 3f + 2 \tag{10}$$

When $f = 1$ or $f = 2$, we take less equations into consideration. In case $f = 1$, only d_0, d_1, and d_2 exist. Similarly, when $f = 2$, only d_0–d_3 exist.

■

Lemma A.10: When processing *OBR* requests, after at most $min(2f + 2, 8)$ communication steps from the time the non-malicious sequencer receives an *OBR* request, all replicas will receive the message with $2f + 1$ correct signatures.

Proof: When processing *OBR* requests, a replica memorizes the set of previously seen signatures for the request (Line 19, for Algorithm A.3). If we treat all replicas which receive an *OBR* request in the last round as sources (i.e. the sequencer), then directly from Lemma A.8 and Lemma A.9, we get the claim. ■

Well-formed commit indications. The proof is the same as for the Ring⁻ case.

Validity. The proof is similar as the proof for the Ring⁻ case.

Init Order. The proof is the same as for the Ring⁻ case.

Termination. The proof is the same as for the Ring$^-$ case.

Commit Order. Assume, by contradiction, that there are two committed requests req (by a benign client c) and $req' \neq req$ (by a benign client c') with different commit histories h_{req} and $h_{req'}$ such that neither is the prefix of the other. Clients commit requests either as a response to an *RING*, or a *PANIC* message. There are three possible cases:

- Both committed requests are a direct response to *RING* messages. By Lemma A.7, all correct replicas in Σ^{req}_{last} (resp. $\Sigma^{req'}_{last}$) executed request req (resp. req') with history h_{req} (resp. $h_{req'}$). Let r^{req} be the first correct replica in Σ^{req}_{last}, and let $r^{req'}$ be the first correct replica in $\Sigma^{req'}_{last}$. There are two distinct cases:
  - these replicas are the same ($r^{req} = r^{req'}$). A contradiction with Lemma A.1.
  - one preceeds the other, in ring order which starts from the sequencer. Without loss of generality, we assume $r^{req} < r^{req'}$. By Lemma A.5, r^{req} executed all requests $r^{req'}$ has had executed, at the same position. A contradiction.
- Both committed requests are a direct response to *OBR* messages. From Step R$^+$4b.3a.1, a client commits a request, if there are $f+1$ matching *GET_A_GRIP* messages. By Step R$^+$4b.3a, a replica executes a request and sends a *GET_A_GRIP* message if there are at least $2f+1$ correct signatures. Thus, each client commits a request, after receiving a message executed by at least $f+1$ correct replica. These two sets (carried in *GET_A_GRIP* messages) of correct replicas intersect on one correct replica, which executed both requests. A contradiction by Lemma A.1.
- One committed request is a direct response to a *RING* message, while the other is a direct response to an *OBR* message. Without loss of generality, let assume client c committed req as a direct response to the *RING* message, while client c' committed req' as a direct response to the *OBR* message. By Lemma A.7, all correct replicas in Σ^{req}_{last} executed req. Let r^{req} be the first correct replica in Σ^{req}_{last}. By Lemma A.5, all correct replicas in the range $\{r_{req.entry} \dots r^{req}\}$ executed request, and there are at least $f+1$ correct replicas in that range (as r^{req} belongs to the last $f+1$ replica in the ring orders starting from $req.entry$). Similarly to the previous case, client c' commits the request after receiving $f+1$ matching *GET_A_GRIP* messages. Every replica which sent the *GET_A_GRIP* message executed the request, after receiving an *OBR* message with at least $2f+1$ signature. Thus, the set of correct replicas which executed req, and set of replicas which executed req' intersect on at least one correct replica. A contradiction by Lemma A.1.

Abort Order. Assume, by contradiction, that there is a committed request req_C (by some benign client) with a commit history h_{req_C} and an aborted request req_A (by some benign client) with commit history h_{req_A}, such that h_{req_C} is not a prefix of h_{req_A}. There are two different cases:

- req_C was committed without client sending a *PANIC* message. By Lemma A.7 and the assumption of at most f faulty replicas, all correct replicas (at least one) from $\Sigma^{req_C}_{last}$ execute req_C and their state upon executing req_C is h_{req_C}. Let $r_j \in \Sigma^{req_C}_{last}$ be a correct replica with the highest (w.r.t. the ring order which starts at $req_C.entry$) index ind among all replicas in $\Sigma^{req_C}_{last}$. By Lemma A.6, all correct (at least $f+1$) replicas r_k ($req_C.entry \leq k < j$) execute all the requests in h_{req_C} at the same positions these requests have in h_{req_C}.
- req_C was committed during handling of the *PANIC* message sent by the client. By Lemma A.10, and Step R$^+$4b.3a, all correct replicas (at least $2f+1$ replicas) execute req_C.

In addition, a correct replica executes all the requests in h_{req_C} before sending any $ABORT$ message (Step R$^+$4b.3b.1); indeed, before sending any $ABORT$ message, a correct replica must stop further execution of requests. Therefore, for every local history LH_j that a correct replica sends in an $ABORT$ message, h_{req_C} is a prefix of LH_j.

Finally, by Step R$^+$4b.3b.2, a client that aborts a request waits for $2f+1$ $ABORT$ messages including at least $f+1$ from correct replicas. By construction of the abort history every commit history, including h_{req_C} is a prefix of every abort history, including h_{req_A}, a contradiction.

■

Non-Triviality. Non-Triviality relies on the fact that replica's timer triggered in Step R$^+$4b.1 is set such that it does not expire in case when the set of replicas, including the client, is synchronous.

Assume by contradiction that there is a correct replica r that stops and denote the first such time by t_{STOP}. Replica r has sent the *OBR* message m at $t = t_{STOP} - ((2f+1)+1)\Delta$. Since no client panics by t_{PANIC} all replicas execute all requests they receive by t_{PANIC}. Then, it is not difficult to see, since there are no link failures, that: (i) by $t+\Delta$ the sequencer receives m and takes Step R$^+$4b.2, and (ii) by time $t+(f+1+1)\Delta < t_{STOP}$ all correct replicas take Step R$^+$4b.2 for m, and (iii) by time $t+((2f+1)+1)\Delta < t_{STOP}$ all correct replicas take Step R$^+$4b.3a. Since the sequencer is correct then all correct replicas execute all requests received before t_{STOP} in the same order (established by the sequence numbers assigned by the sequencer). Hence, by $t+((2f+1)+1)\Delta = t_{STOP}$, r receives a message with at least $2f+1$ signatures (Step R$^+$4b.3a), commits request req (associated with m) and does not abort. A contradiction.

In addition, a correct replica r_i executes Step R$^+$4b.3b and stops appending new requests, only if r_i fails to commit an OBR request for a *RING* message signed by some client. Since such an OBR request cannot raise a verification failure, r_i can fail to commit such request only in case asynchrony in the set of replicas, as per Lemma A.10 if the sequencer is correct, a malicious replica cannot prevent correct replicas from receiving the *OBR* message.

■