Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas

Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke and Neeraj Suri
Computer Science Department, Technische Universität Darmstadt
{marco, pbokor, dan, majuntke, suri}@cs.tu-darmstadt.de

Abstract

Byzantine-Fault-Tolerant (BFT) state machine replication is an appealing technique to tolerate arbitrary failures. However, Byzantine agreement incurs a fundamental trade-off between being fast *(i.e. optimal latency) and achieving optimal resilience (i.e.* $2f + b + 1$ *replicas, where* f *is the bound on failures and* b *the bound on Byzantine failures [10]). Achieving fast Byzantine replication despite* f *failures requires at least* $f + b - 2$ *additional replicas [11, 7, 9]. In this paper we show, maybe surprisingly, that fast Byzantine agreement despite* f *failures is practically attainable using only* $b - 1$ *additional replicas, which is independent of the number of crashes tolerated. This makes our approach particularly appealing for systems that must tolerate many crashes (large* f*) and few Byzantine faults (small* b*). The core principle underlying our approach is to have the correct replicas agree on a quorum of responsive replicas before agreeing on requests. This is key to circumventing the resilience lower bound of fast Byzantine agreement [7].*

Submission: Regular paper
Declaration: Cleared through the authors affiliations

1 Introduction

Byzantine Fault Tolerant (BFT) state machine replication [12] has the potential to become a generic solution for reliable distributed computing. BFT replication can be used to make any deterministic server application tolerant to worst-case failures in eventually synchronous systems. However, the potential for generality can be fully exploited only if the performance overhead and replication costs of BFT are minimized. This motivates much ongoing work aimed at making BFT replication more efficient, from the PBFT protocol [2], to *quorum-based* protocols [1, 6], to *fast* BFT protocols exhibiting the optimal number of agreement steps [11, 7, 9]. Table 1 compares the properties of relevant primary-based BFT replication protocols and shows how practical BFT replication [2] has evolved to more advanced solutions based on speculation like Zyzzyva [9].

Speculative BFT protocols [9] and other fast Byzantine agreement protocols such as FAB [11] and DGV [7] improve the performance of BFT replication by trading optimal resilience for optimal performance (in terms of latency and throughput) in presence of *unresponsive* replicas. Replicas can be unresponsive if they are faulty or simply slow relative to other replicas due to heavier workload or poorer network connection. Fast BFT protocols can deliver a reply to the client with optimal latency (three communication steps if the client is not a replica). In order to attain optimal latency in presence of f unresponsive replicas while tolerating $b = f$ Byzantine faults, these protocols need up to $2f$ additional replicas compared to the minimum of $3f + 1$.

Lower bounds [11, 7] show that if only $3f+1$ replicas are used, no protocol can be fast in the presence of even a single unresponsive replica. The Zyzzyva protocol, for example, exhibits optimal resilience [10] as it requires $3f + 1$ replicas to tolerate f Byzantine faults. However, Zyzzyva requires all replicas to respond to clients fast enough in order to reach fast agreement and leverage speculation. If some replica is slower or faulty, clients need to face a dilemma: either they wait for the missing responses, which may never arrive or may be indefinitely slow, or require replicas to execute a slower explicit agreement. This dilemma is eliminated in the Zyzzyva5 protocol by raising replication costs from $3f+1$ to $5f+1$, thus sacrificing minimum replication costs in favor of better performance. Note that quorum-based protocols such as Q/U [1] and HQ [6] do not offer better performance than Zyzzyva [9, 16].

Contributions In this paper we propose Scrooge, a new BFT replication protocol which improves on the resilience of fast Byzantine agreement in presence of unresponsive replicas. Scrooge requires only $N = 2f + 2b$ replicas to tolerate $f > 0$ faults, out of which $b > 0$ are Byzantine, and it is fast despite f unresponsive replicas. This is $f - 1$ less replicas than any existing fast Byzantine replication algorithm [11, 7].

■ It is important to note that Scrooge does not contradict the resilience lower bounds of fast byzantine agree-

ment [11, 7]. Scrooge attains fast Byzantine agreement only under the additional condition that a set of $N - f$ responsive replicas is known in advance. However, this additional requirement can be implemented and hence has little practical impact. Scrooge, in fact, ensures that under the same assumptions required for speculation (i.e. primary is fault-free, clients are honest and communication is timely) the set of responsive replicas *eventually* is identified. Thus, fast agreement eventually is provided for all requests.

■ All algorithms in Table 1 preserve safety in worst-case scenarios but are designed to achieve high performance only in more common scenarios with fault-free or unresponsive replicas and clients. Achieving liveness in presence of worst-case attacks requires using different techniques, such as using specific network topologies, which are mostly orthogonal to our work and which go beyond the scope of this and the other cited papers [4]. However we explicitly consider the use of signatures for client requests, as indicated in [4], because this impacts the design of our protocol.

■ We designed Scrooge to use few replicas when b is small. In fact, Scrooge requires only $b - 1$ replicas more than the optimal number of $2f+b+1$ [10]. Unlike any other fast Byzantine agreement protocol, the replication overhead incurred by Scrooge does not depend on f. This allows Scrooge to scale with the number of unresponsive replicas tolerated. Moreover, when a single Byzantine fault needs to be tolerated, Scrooge achieves optimal resilience ($2f + 2$) and requires only one additional replica compared to protocols tolerating only crashes.

■ We experimentally and analytically evaluated Scrooge. Scrooge performs as well as state-of-the-art fast BFT protocols like Zyzzyva if all replicas are responsive. In scenarios with at least one unresponsive replica we found that:

- The peak throughput advantage of Scrooge is more than 1.3 over Zyzzyva. Scrooge also has lower latency with high load.
- Scrooge reduces latency with low load by at least 20% and up to 98% compared to Zyzzyva.
- Scrooge performs as well as Zyzzyva5, which uses $f+1$ more replicas than Scrooge (with $f = b$).
- As the number of tolerated faults increases, the overhead of Scrooge degrades more slowly than in other protocols using equal or lower redundancy.

1.1 First technique: Replier quorums

Scrooge uses two novel techniques, *replier quorums* and *message histories*, to reduce replication costs. The first technique consists of having replicas agree on a set of replicas, termed *replier quorum*, whose members are the only ones responsible for sending replies to clients in normal runs. A distinguished replica, called the primary, sends messages to the other replicas that dictate the order of execution of requests. Scrooge uses speculation so replicas directly reply to the client without reaching agreement on the execution order first (Fig. 1.a). This allows clients to immediately deliver a reply if all the repliers are responsive and correct. If a replier becomes unresponsive or starts behaving incorrectly, this is indicated by clients to the replicas, which then execute a *reconfiguration* to a new replier quorum excluding the suspected replica.

	Replication costs (min. $2f + b + 1$ [10])	Fast($\star$) w. no unresponsive replica	Fast($\star$) w. f unresponsive replicas
PBFT [2]	$3f + 1$	NO	NO
Zyzzyva [9]	$3f + 1$	YES	NO
Zyzzyva5 [9]	$5f + 1$	YES	YES
DGV [7]	$3f + 2b - 1_{(\triangleright)}$	YES	YES
Scrooge	$2f + 2b$	YES	YES

Table 1: Comparison of primary-based BFT replication protocols that tolerate f failures, including b Byzantine ones. The first three protocols assume $f = b$. ($\star$) A protocol is *fast* if it has minimal best case latency to solve consensus [11, 7]. If the primary is faulty or the clients are Byzantine none of these protocols is fast. Upon backup failure events, Scrooge is fast after a bounded time whereas Zyzzyva5 is always fast. ($\triangleright$) Cost for $f > 1$ in order to be fast with f unresponsive replicas. For $f = 1$ the corresponding cost is $2f + 2b + 1$ replicas.

During reconfigurations, explicit agreement is performed by the replicas (Fig. 1.b). The agreed-upon value contains two types of information: the execution order of client requests *and* the new replier quorum. Agreeing on the order of requests ensures that all client requests can complete even in presence of faulty or unresponsive repliers. Agreeing on the new replier quorum allows future requests to be efficiently completed using speculation. Coupling these agreements reduces the overhead incurred by reconfiguration. The goal of the first explicit agreement in Fig. 1.b is just completing the ongoing request from Client i. When the request of Client j is received, the primary has a chance to propose a new replier quorum and let all replicas explicitly agree on it. Speculation is reestablished as soon as this agreement is reached.

Scrooge requires clients to participate in the selection of repliers. Giving more responsibility to clients is common in many BFT replication protocols, such as Q/U, HQ and Zyzzyva. This is reasonable as clients are ultimately entrusted not to corrupt the state of the replicated state machine with their requests. Scrooge protects the system from Byzantine clients and ensures that they can not make replica states diverge. However, Byzantine clients can reduce the performance of the system by forcing it to perform frequent reconfigurations and to use the communication pattern of PBFT (like in the request of Client j in Fig. 1.b), which

anyway allows achieving good performance, instead of using speculation (like in Fig. 1.a). This kind of client attacks can be easily addressed by simple heuristics, as for example by bounding the number of accusations a client can send in a given unit of time. This reliance on the client to indicate suspected replicas results from the use of speculation. Replicas can not observe if other replicas prevent fast agreement by not sending correct speculative replies.

Reconfigurations are avoided in existing speculative protocols such as Zyzzyva5 by using more replicas than Scrooge. Replier quorums allow reducing the replication costs to $4f + 1$ replicas when $f = b$.

1.2 Second technique: Message histories

The second technique leverages the Message Authentication Codes (MACs) used in BFT replication protocols to implement authenticated channels and to detect forged and corrupted messages. The sender of a message generates an authenticator, which is a vector of MACs with one entry for each other receiver, and attaches it to the message before sending it. In current primary-based protocols such as [2, 9] replicas store the history of operations dictated by the primary but discard the authenticator after the authenticity of the message has been verified. In Scrooge, replicas store the entire messages (also the authenticators), which are therefore called *message histories*. Message histories further reduce the replication cost from $4f + 1$ to $4f$ (again with $f = b$).

Paper structure. In Section 2 we introduce the system and fault models. The Scrooge protocol is described in Section 3. Section 4 describes the view change algorithm of Scrooge, which leverages our two novel techniques. The experimental comparison of Scrooge with state-of-the-art fast BFT state machine replication protocols is reported in Section 5. Related work is discussed in Section 6.

2 System and Fault Model

The system is composed of a finite set of clients and replicas. We assume that at most f replicas can be faulty, out of which at most b can be Byzantine with $0 < b \leq f$ while the others can only crash. The system consists of at least $N = 2f + 2b$ replicas. Any number of clients can be Byzantine. Clients and replicas are connected via an asynchronous network without known upper bounds on the communication delays. The network can drop, corrupt or duplicate messages or deliver them out of order. We say that the system is in a timely period when all messages sent among correct nodes are delivered within a bounded delay.

We assume the availability of computationally secure cryptographic primitives. Namely, we assume the availability of symmetric key cryptography to calculate MACs, and public key cryptography to sign messages. If message m is sent by process i to process j and is authenticated using simple MACs, this is denoted as $\langle m \rangle_{\mu_{i,j}}$. In case m is sent to all replicas by process i, an authenticator consisting of a vector of MACs with one entry per replica is sent with m and denoted as $\langle m \rangle_{\mu_i}$. If m is signed by i using its private key we denote it as $\langle m \rangle_{\sigma_i}$. We also assume the availability of a collision-resistant hash function H to compute message digests ensuring that it is impossible, given $H(m)$, to find a message $m' \neq m$ such that $H(m) = H(m')$.

Name	Description	Type
v	current view	timestamp
RQ	replier quorum	set of pids
n	current seq. number	timestamp
mh	message history	array of $\langle$*req.*, RQ, *auth.*$\rangle$
h	history digests	array of digests
aw	agreed watermark	timestamp
cw	commit watermark	timestamp
SL	suspect list	set of f pids
v'	new view	timestamp
ih	initial history	array of $\langle m, RQ,$ *auth.*$\rangle$
E	view establishment certificate	set of $N - f$ signed EST-VIEW messages

Table 2: Global Variables of a Replica

3 The Scrooge Protocol

Scrooge replicates deterministic services, modeled as state machines, over multiple servers. Clients use Scrooge to interact with the replicated servers as if they were interacting with a single reliable server.

Throughout the protocol description we reason about its correctness. Complete correctness proofs can be found in our technical report [13]. Beyond the classic safety and liveness properties of BFT replication, we prove that in Scrooge clients eventually complete all their requests from speculative replies if the usual conditions for speculation are satisfied (i.e. the primary is fault-free, the clients are non-Byzantine and the system is timely).

We ease the discussion by presenting a simplified version of Scrooge which assumes that replicas process unbounded histories. A complete description of the full Scrooge protocol with garbage collection, together with full correctness proofs, can be found in [13].

3.1 Normal Execution

In normal executions where the system is timely, the primary is fault-free and the replier quorum is agreed by all replicas and contains only fault-free replicas, Scrooge behaves as illustrated in Fig. 1.a and Alg. 1.[1] Table 2 summarizes the local variables used by the replicas. We informally describe the predicates and the helper procedures used in the pseudocode and refer to [13] for a more formal definition. Only MACs are used for normal runs and reconfigurations.

[1] Upon receiving a message, processes discard them if they are not well-formed, that is, signatures, MACs, message digests or certificates are not consistent with their definitions. We ignore such non well-formed messages in the pseudocode.

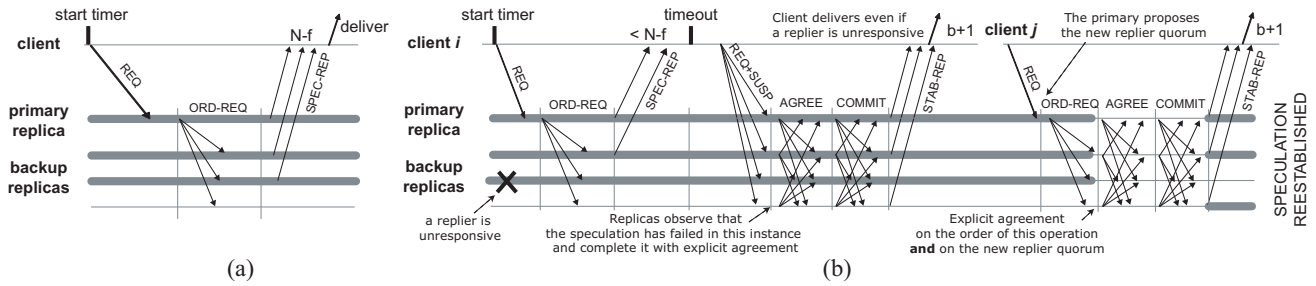

Figure 1: Communication patterns: (a) with speculation, during normal periods; (b) with explicit agreement, during transient reconfiguration periods where two client requests are processed. Repliers are indicated with a thicker line.

Algorithm 1: Scrooge - Normal Execution

```
1.1  upon client invokes operation o
1.2      t ← t + 1;  SL ← ⊥;
1.3      send m = ⟨REQ, o, t, c, SL⟩_σc to the primary;
1.4      start timer;
1.5
1.6  upon primary p(v) receives request m from client m.c or a replica
1.7      if not IN-HISTORY(m, mh) then
1.8          n ← n + 1;  d ← H(m);  RQ_p ← replicas ∉ SL;
1.9          send (⟨ORD-REQ, v, n, d, RQ_p⟩_μp, m) to all replicas;
1.10     else if not COMMITTED(m, mh, cw) then
1.11         update(m.SL);
1.12         agree(m);
1.13     else reply-cache(m.c);
1.14
1.15 upon replica i receives ordered request orq from primary p(v)
1.16     if i = p(v) or (orq.v = v and orq.n = n + 1 and
         p(v) ∈ orq.RQ_p) and not IN-HISTORY(orq.m, mh) then
1.17         n ← n + 1;  h[n] ← H(h[n − 1], mh[n]);
1.18         mh[n] ← ⟨orq.m, orq.RQ_p, orq.μ_p⟩;
1.19         r ← execute(orq.m.o);
1.20         if SPEC-RUN(i, orq.m, orq.RQ_p, RQ) then
1.21             if i ∈ RQ then
1.22                 send ⟨SPEC-REP, v, n, h[n], RQ, orq.m.c,
                     orq.m.t, r, i⟩_μp,c to client orq.m.c;
1.23         else
1.24             agree(orq.m);
1.25             if RQ ≠ orq.RQ_p then RQ ← ⊥;
1.26         if AGREEMENT-STARTED(i, n, v) then agree(orq.m);
1.27
1.28 upon client receives speculative reply sp from replica sp.i
1.29     if ∃RQ : received N − f matching speculative replies sp to m with
         sp.RQ = RQ from all replicas in RQ then
1.30         deliver (o, t, sp.r);   stop timer ;
1.31
```

Scrooge runs proceed through a sequence of views. In each view v, one replica, which is called the primary and whose ID is $p(v) = v \bmod N$, is given the role of assigning a total execution order to each request before executing it. The other replicas, called backups, execute requests in the order indicated by the primary.

Clients start the protocol for an operation o with local timestamp t by sending a signed *request* message REQ to the primary. Clients then start a timer and wait for speculative replies (Lines 1.1 – 1.4). When the primary receives a request for the first time (Lines 1.6 – 1.9) it assigns it a sequence number and sends an *order request* message ORD-REQ to mandate the same assignment to all backups. The primary also stores the request in its message history together with the current replier quorum RQ_p and the authenticator μ_p of the ORD-REQ message.

When a replica receives order requests from the primary of the current view (Lines 1.15 – 1.19), it checks that its view number is the current one, that it contains the next sequence number not yet associated with a request in the message history (predicate IN-HISTORY), and that the primary has included itself in the replier quorum. If all these checks are positive, the request is executed and the fields of the ORD-REQ message are added to the message history.

Speculative runs where the pattern of Fig. 1.a is executed are the common-case runs (Lines 1.20 – 1.22). A replica checks the predicate SPEC-RUN to distinguish speculative runs. The predicate is true unless (i) a client could not complete the request out of speculative replies and has resent its request to all replicas, including backups, or (ii) the primary has proposed a new replier quorum which has not yet been agreed upon. In speculative runs, replicas send a *speculative reply* to the client if it is a replier. Beyond the reply r and the view number v and the sequence number n associated to the client request, speculative replies contain the digest of the current history $h[n]$ and the replier quorum RQ. The former allows clients to verify that the senders of speculative replies have a consistent history; the latter to identify the replicas in the current replier quorum. If a client receives $N - f$ matching speculative replies from RQ, it delivers the reply (Lines 1.28 – 1.30).

3.2 Reconfiguration

If a replica in the replier quorum fails, the client can not complete requests out of speculative replies. The replier quorum is then reconfigured by identifying and eliminating faulty repliers to re-establish the communication pattern of Fig. 1.a. Replicas execute a full three-phase agreement similar to PBFT [2] (see [13] for the full pseudocode). An example of reconfiguration involving two client requests is given in Fig. 1.b.

3.2.1 Completion of client requests

When clients cannot deliver speculative replies before the timer expires, they double the timer, indicate the IDs of the repliers which have failed to respond and require replicas to explicitly agree on a common message history. Similar to Client i in Fig. 1.b, they do this by simply resending their requests m, together with the set SL of suspect replicas, to all replicas.

When the primary receives a request which is already in its message history, it checks the predicate COMMITTED, which is true when a three-phase agreement on the order of the request has been already completed. If not, it adds the suspect list provided by the client into the list of the f most-recently suspected servers SL and starts agreement (Lines 1.10 – 1.12). Backup similarly start agreement because receiving the client request invalidates SPEC-RUN. However, they need to receive the corresponding ordered request from the primary first (Lines 1.23 – 1.26). A replica i also starts an agreement phase whenever another replica previously sent it an agreement message (Line 1.26).

Replicas then execute the remaining two phases of agreement, agree and commit, to converge to a consistent history and send stable replies to the client. In each phase replicas send an *agree* or a *commit* message and wait for $N - f - 1$ matching messages from the other replicas before completing the phase. The agree and commit watermarks aw and cw mark the end of the history prefix which has been respectively agreed and committed. Like in PBFT, all correct replicas completing the agreement phase for sequence number n' have the same message history prefix up to n'. When correct replicas complete the commit phase for n', they know that a sufficient number of correct replicas have completed agreement on the history prefix up to n' to ensure that the prefix will be recovered during view change. Replicas thus sends *stable reply* messages to the client. Stable replies are different from speculative replies because they indicate that the history prefix up to the replied request can be recovered. Clients can deliver after receiving a stable reply from at least one correct replica, that is, after receiving matching stable replies from *any* set of $b + 1$ replicas. Replicas cache the replies to committed requests to respond to clients re-sending their requests (Line 1.13).

3.2.2 Agreement on a new replier quorum

The classic three-phases agreement is also executed for all subsequent requests until a new replier quorum is agreed, as in case of the request of Client j in Fig. 1.b. This time agreement is needed to reconfigure to the new replier quorum RQ_p calculated from the new suspect list SL in Line 1.8. The primary proposes RQ_p along with the next request which is ordered. Proposing a new replier quorum invalidates the SPEC-RUN predicate for all backups and lets them start agreement (Lines 1.23 – 1.25). Replicas register ongoing reconfigurations by setting RQ to $\perp$ until a reconfiguration is completed. They then start the successive two phases of agreement. Explicit agreement lets replicas converge not only on a common history but also on a new replier quorum. When a replica commits, it sets RQ to the new replier quorum proposed by the primary so SPEC-RUN holds again for future requests and speculation is reestablished. The commit on the new replier quorum ensures that it will be recovered if view changes take place. When a replier quorum is updated for sequence number n, correct replicas also send speculative replies with the new replier quorum in order not to be suspected by other clients waiting for speculative replies for sequence numbers greater than n.

4 Scrooge View Change

If correct backups receive requests from the clients and see that the system is not able to commit them before the timer expires, they start a view change to replace the current primary. There are two key differences making view change in Scrooge harder than in existing protocols. In contrast to PBFT, we can only expect replicas to have explicitly agreed on a *prefix* of the request history completed by clients. Also, different from existing fast protocols allowing speculation in presence of unresponsive replicas, Scrooge uses a lower number of replicas. We developed a novel view change protocol (see Alg. 2) to achieve these challenging goals. As customary in BFT protocols, replicas use signed messages during view change.

4.1 Communication Pattern

View change to a new view v' tries to build an initial history ih for v', which is then adopted as new message history when v' is started. When a replica initiates view change from the current view v to view v', it stops processing requests, starts a timer, and sends a *view change* message VIEW-CHANGE to all replicas (see Fig. 2 point a and Lines 2.1 – 2.5). A view change can also be initiated when a replica receives $b + 1$ view change message for a newer view (Lines 2.15 – 2.16).

A view change message contains the new view v' that the replica wants to establish, the old view v, its message history mh, the view establishment certificate E and the agreement watermark aw. The message history mh contains as prefix the initial history ih_v of v, which was stored at the end of the view change to view v. By induction on the correctness of the view change subprotocol for a given view, ih_v contains every operation completed by any client in the views prior to v. The view establishment certificate E contains the EST-VIEW messages received at the end of the view change to view v and proves the correctness of ih_v. The remaining suffix of mh contains the ORD-REQ messages received by i from the primary of view v. These requests need to be recovered by the view change if they have been observed by any client.

Algorithm 2: Scrooge - View change

```
2.1  procedure view-change(nv)
2.2      stop executing request processing;
2.3      v' ← nv;
2.4      send ⟨VIEW-CHANGE, v', v, mh, aw, E, i⟩_σi to all replicas;
2.5      start timer;
2.6
2.7  upon replica i receives view change message vc from replica vc.i
2.8      if vc.v' > v and not yet received a view change message vc for view
         nv = vc.v' from vc.i then
2.9          k ← n' + 1 : ∀ev ∈ vc.E, cv.n = n';
2.10         while mh[k] ≠ ⊥ do
2.11             res[k] ← verify(vc.v, k, vc.mh[k]);
2.12             k ← k + 1;
2.13         d ← H(vc); j ← vc.i; v_j ← vc.v;
2.14         send ⟨CHECK, j, v_j, d, res, i⟩_σi to p(vc.v');
2.15         if received b + 1 vc msgs with vc.v' > v' then
2.16             view-change(vc.v');
2.17         if i = p(v') and vc.v' = v' and recover-prim() then
2.18             send ⟨NEW-VIEW, v', VC, CH, i⟩_μi to all replicas;
2.19
2.20 upon replica i receives a check message ch
2.21     if i = p(v') and ch.v_j = v' and recover-prim() then
2.22         send ⟨NEW-VIEW, v', VC, CH, i⟩_μi to all replicas;
2.23
2.24 upon replica i receives a new view message nv
2.25     if not yet received nv with nv.v' = v' from p(v') and
         recover(nv.VC, nv.CH) then
2.26         h ← H(ih);
2.27         n ← length(ih);
2.28         send ⟨EST-VIEW, v', n, h, i⟩_σi to all replicas;
2.29
2.30 upon replica i receives an establish view message ev
2.31     if received set E_v' of N − f − 1 ev msgs: ev.v' = v' and
         ev.h = H(ih) and ev.n = length(ih) then
2.32         mh ← ih; v ← v'; E ← E_v';
2.33         aw, cw ← max{k : mh[k] ≠ ⊥}; RQ ← mh[cw].RQ;
2.34         start executing request processing;
2.35
```

A novelty of Scrooge is that each replica which receives the view change message from i checks if the messages in the history mh has been actually sent by the primary of view v (see Fig. 2 point b). Let $vc.v$ be the value of the current view field v contained in a view change message vc sent by replica i to replica j. Scrooge executes one additional step during view change to validate that all history elements in vc, except those in the initial history of view $vc.v$, have been built from original order request messages from the primary of view $vc.v$ (Lines 2.7 – 2.14). When j receives vc, it first verifies that the new view field $vc.v'$ is higher than the current view v of j and that i has not already sent to j a view change message for the same view. Next, j checks if the elements in the message history of i are "authentic". For each element with sequence number k, j calls the *verify* function (see Alg. 3) which first rebuilds the order request message sent by the primary of view $vc.v$ to i for sequence number k, and then verifies the authenticator of the message. Message histories make the first operation possible because they contain enough information to rebuild the original order request messages, including the message authenticator $\mu_{p(vc.v)}$ used by the primary of view $vc.v$. Replica j verifies the authenticator by calculating the MAC of the rebuilt order request message and by returning *true* if and only if this MAC is equal with the entry of j in $\mu_{p(vc.v)}$. The results of the verification of each element in the message history of vc is stored in a vector res, which is sent to the primary of the new view v' in a CHECK message together with additional information to associate the check message to vc.

Different from existing algorithms, the new primary only recovers from *stable* view change messages that are consistently checked by at least $b + 1$ replicas (Fig. 2 point c). If these messages claim that the message history is authentic we call the history *verified*. This is the way the VERIFIED predicate is defined. The purpose of the additional check step will become clearer at the end of this section, when we discuss the details of recovery. For the moment, we just note that all view change messages eventually become stable in timely periods and that the goal of this step is ensuring that if the primary of the old view v is non-Byzantine and i stores correct ORD-REQ messages in its history, then: (P1) the message history becomes verified because it receives positive CHECK messages from all correct replicas, which are at least $b+1$, and (P2) no forged, inconsistent history can receive a positive CHECK message by any correct correct replica and thus become verified.

The primary of a new view v' calls the function *recover-prim()* (see Alg. 3) to try to recover the initial history ih whenever it receives a view change message (Lines 2.17 – 2.18) or a check message (Lines 2.20 – 2.22) for v'. We call this the *recovery* function. As said before, recovery examines only stable VIEW-CHANGE message for the new view. The procedure returns *true* only if it is able to successfully recover all operations completed by any client in all views prior to v'. In this case, the resulting history forms the initial history of v' and is stored into ih. We argue about the correctness of the recovery function in the next subsection and continue illustrating the communication pattern.

If history ih is recovered, the primary sends a *new view* message to all other replicas with the sets of view change and check messages VC and CH used for the recovery (see Fig. 2 point d). When a backup receives a new view message for the view it is trying to establish (Lines 2.24 – 2.28) it executes the same deterministic *recover* function as the primary does on the same set of view change and check messages to build the same initial history. If the backup recovers an initial history ih for a new view v', it sends an *establish view* message to all other replicas in order to agree

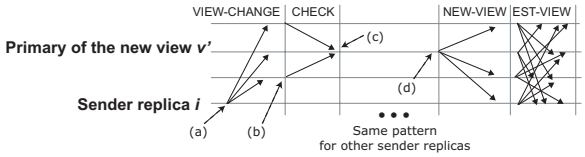

Figure 2: Scrooge view change subprotocol.

Algorithm 3: Scrooge - View change procedures

```
3.1  function verify(v, n, e)
3.2      d ← H(e.m); or ← (ORD-REQ, v, n, d, e.RQ);
3.3      μ ← calculate-MAC(or, p);
3.4      if μ = e.μ_p[i] then return true;
3.5      else return false;
3.6
3.7  function recover(VC, CH)
3.8      recovered ← false;
3.9      VC_s ← VC \ {vc ∈ VC : ¬STABLE(vc, CH) ∨ vc.v' ≠ v'};
3.10     if |VC_s| ≥ N − f then
3.11         mv ← max{v̄ : ∃vc ∈ VC_s with vc.v = v̄};
3.12         vc_mv ← vc ∈ VC_s with vc.v = mv;
3.13         n_mv ← n̄ : ∀ev ∈ vc.E, ev.n_v = n̄;
3.14         ih ← {vc_mv.mh[k] : (k ≤ n_mv)};
3.15         RQ_mv ← vc_mv.mh[n_mv].RQ;
3.16         k ← n_mv + 1; loop ← true; recovered ← true;
3.17         while loop do
3.18             A ← {e : AGREED-CAND(e, k, mv, VC_s, ih)};
3.19             O ← {e : ORDERED-CAND(e, k, mv, VC_s, RQ_{k−1},
                 ih)};
3.20             if WAIT-AGR(A, k, mv, VC_s) or
                 WAIT-ORD(A, O, k, mv, VC_s) then
3.21                 loop, recovered ← false;
3.22             else
3.23                 if ∃e ∈ A then
3.24                     ih[k] ← e;
3.25                 else if ∃e ∈ O : VERIFIED(e, VC_s, CH) then
3.26                     ih[k] ← e;
3.27                 else if ∃e ∈ O then
3.28                     ih[k] ← e;
3.29                 else loop ← false;
3.30                 RQ_k ← ih[k].RQ;
3.31             k ← k + 1;
3.32     return recovered;
3.33
3.34 function recover-prim()
3.35     VC ← set of received view change messages for view v';
3.36     CH ← set of received check messages for view v';
3.37     return recover(VC, CH);
3.38
```

on ih. If it later receives $N-f-1$ establish view messages for v' consistent with ih, it forms a view establishment certificate for ih, sets v' as its current view and ih as its agreed history prefix, and updates the watermarks (Lines 2.30 – 2.34). The replica then starts processing messages in the new view.

If the replica timer expires before the new view is established, a view change to a successive new view $v'+1$ is started, the timer is doubled and all messages related to the view change to v' are discarded.

4.2 The recover function

The *recover* function (see Alg. 3) is a critical component because it guarantees that safety is preserved and that each history prefix observed by any correct client in previous views is also a prefix of the initial history ih of the next view. In order to allow the expert reader to verify all the nuances of the algorithm, and in particular of recovery, we list the predicates used in the pseudocode in Table 3.

Before starting recovery, a replica i makes sure that it has received a set VC_s of at least $N-f$ *stable* view change messages for the new view v'. A view change message vc

IN-HISTORY$(m, mh) \triangleq \exists k : mh[k].m.c = m.c \wedge mh[k].m.t \geq m.t$

COMMITTED$(m, mh, cw) \triangleq \exists k \leq cw : mh[k].m.c = m.c \wedge mh[k].m.t \geq m.t$

SPEC-RUN$(i, m, RQ_p, RQ) \triangleq RQ \neq \perp \wedge RQ_p = RQ \wedge$ i is backup and has never received a message with timestamp $\geq m.t$ from $m.c$

AGREEMENT-STARTED$(i, n, v) \triangleq$ i has received an agree message ag with $ag.n = n$ and $ag.v = v$ in view v

STABLE$(vc, CH) \triangleq \exists bool : \forall k : vc.mh[k] \neq \perp,$ $\exists (b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i \wedge$ $ch.d = \text{digest}(vc) \wedge ch.res[k] = bool$

AGREED-CAND$(e, k, v, VC, ih) \triangleq$ not IN-HISTORY$(e, ih) \wedge$ $\big(\exists\ (b+1)\ vc' \in VC : vc'.v = v \wedge vc'.mh[k] = e\big) \wedge$ $\big(\ \exists(|VC| - f - b)\ vc \in VC :$ $e = vc.mh[k] \wedge vc.v = v \wedge vc.aw \geq k\ \big)$

ORDERED-CAND$(e, k, v, RQ, VC, ih) \triangleq$ not IN-HISTORY$(e, ih) \wedge$ $\exists(|VC| - f - b)\ vc \in VC :$ $e = vc.mh[k] \wedge vc.v = v \wedge vc.i \in RQ \wedge vc.aw < k$

WAIT-AGR$(A, k, v, VC) \triangleq \exists e \in A :$ $\big(\nexists\ (b+1)\ vc \in VC : vc.v = v \wedge vc.mh[k] = e)\big) \wedge$ $\big(\nexists\ (f+b+1)\ vc' \in VC :$ $(vc'.v \neq v) \vee (vc'.v = v \wedge vc'.mh[k] \neq e\big)$

WAIT-ORD$(A, O, k, v, VC) \triangleq |A \cup O| > 1 \wedge |VC| \leq N - f \wedge$ $(\exists\ vc \in VC : vc.i = p(v) \wedge vc.v = v)$

VERIFIED$(e, VC, CH) \triangleq \exists k,\ vc \in VC : e = vc.mh[k] \wedge$ $(\exists (b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i$ $\wedge\ ch.d = \text{digest}(vc) \wedge ch.res[k] = true)$

Table 3: Predicates

is stable if each element in the corresponding message history $vc.mh$ is consistently verified by at least $b+1$ check messages received by i (Lines 3.9 – 3.10). In timely periods each view change message vc sent by correct replicas eventually become stable as all $N-f \geq f+2b > 2b$ correct replicas send CHECK messages containing binary vectors res for $vc.mh$.

Recovery starts by selecting an initial prefix for the initial history ih (Lines 3.11 – 3.15). The highest current view mv included in a view change message in VC_s is either the last view $lv < v'$ where some client has completed a request, or a successive view where all requests completed in lv have been recovered. This is because at least $N-f-b \geq f+b$ correct replicas must have established lv and at least $b > 0$ of them have sent a message included in VC_s. Scrooge first recovers the initial history ih of mv from any view change message containing a message history for mv. View change messages include a view establishment certificate E composed of $N-f$ signed messages all containing the same length n_{mv} of the initial history of mv and the same corresponding history digest. The certificate ensures that the initial history ih recovered from the view change message vc_{mv} is the correct initial history for mv and is not forged by a Byzantine replica. Together with the initial history also the initial recovery quorum RQ_{mv} is recovered.

The next step is recovering the history elements observed by clients during view mv for sequence numbers $k > n_{mv}$

(Lines 3.16 – 3.31). If a request has been completed by a client from $b + 1$ stable replies, at least one correct replica has committed the entire history prefix up to that request. A replica commits a history element only if the number of replicas which have agreed on it is sufficient to recover the element like in PBFT [2] (Lines 3.23 – 3.24). Our definition of *agreed candidate* and the wait condition used to decide whether a potentially agreed candidate must be recovered are equivalent to those used in the view change of PBFT and are reflected by predicates AGREED-CAND and WAIT-AGR. Therefore, we do not further discuss the recovery of agreed-upon requests and focus on the recovery of histories completed through speculative replies.

4.2.1 Why are replier quorums useful?

If a reply is delivered by clients in a *fast* manner, that is, out of speculative replies (Lines 1.28 – 1.30), recovering it requires a higher redundancy than the minimum. Scrooge reduces these additional costs. By recovering agreed history elements, a replica also recovers the replier quorum which has been updated when the element has been committed. Recovering the replier quorum RQ_n committed for sequence number n allows to clearly identify the set of repliers for sequence numbers greater than n and thus to reduce the number of required replicas to $2f + 2b + 1$. To see that, consider a system having $N = 2f + 2b + 1$ replicas where replier quorums consist of $N - f$ replicas. Assume that a client completes a request in a view v for sequence number $n' > n$ after receiving matching speculative replies from all repliers, at least $N - f - b$ of which are correct, and assume that RQ_n is the last recovered replier quorum for sequence numbers smaller than n'.

If the primary fails, the history prefix up to n' must be recovered to ensure safety. To this end, all replicas share their history, but only the histories of repliers in the replier quorum need to be considered. During view change up to f of the $N - f - b$ correct repliers might be slow and might fail to send a stable VIEW-CHANGE message. Due to the asynchrony of the system, the primary can not indefinitely wait for these messages because it can not distinguish if the replicas are faulty or simply slow. Despite this, the new primary can always receive view change messages from at least $N - 2f - b = b + 1$ correct repliers reporting the history prefix observed by the client. As the primary knows the identity of the repliers and as only b Byzantine repliers can report incorrect histories, the observed prefix can be recovered by selecting a history reported in the VIEW-CHANGE message of at least $b + 1$ repliers.

4.2.2 Why are message histories useful?

Scrooge further reduces the replication costs to $N = 2f+2b$ replicas by using message histories and the check messages. Assume that a client has delivered a reply to a request m after receiving matching speculative replies from all repliers for a sequence number n'. During view change, as we use one replica less than the previous case, the history observed by the client is reported in the VIEW-CHANGE message of at least $N - 2f - b = b$ repliers. Let $|VC_s| \geq N - f$ be the number of stable view change messages received by the primary of the new view. We call a history element reported by $|VC_s| - f - b$ repliers an *ordered candidate*. The set of observed candidates is defined by the predicate ORDERED-CAND. It follows from this definition that two different ordered candidates may be reported for sequence number n' and view v by two sets Q and Q' of $|VC_s| - f - b = b$ repliers each, where Q contains correct repliers and Q' the Byzantine ones. The problem is distinguishing the candidate containing m from other candidates.

If two sets of b replicas claim to have two inconsistent histories for the same view v *and* the old primary p of view v is in one of these sets, then either p is Byzantine and has sent inconsistent order requests to the backups, or b backups are Byzantine and are reporting a forged history. Therefore, at least one Byzantine replier is contained in one of these two sets and that it is live to wait for the view change message from one additional correct replier as indicated by the predicate WAIT-ORD. After the additional VIEW-CHANGE message has been received and has become stable, $|VC_s| > N - f$. As only the correct history is reported by at least $|VC_s| - f - b > b$ repliers, it is recovered as the only remaining observed candidate (Lines 3.27 – 3.28).

If there are two different candidates reported by b replicas each and the primary is *none* of these sets we distinguish two cases. If p is not Byzantine, but potentially faulty, it might be impossible to wait until only one ordered candidate remains. In this case the predicate WAIT-ORD is false and a verified candidate is recovered if present (Lines 3.25 – 3.26). Message histories and the novel check phase allow to identify in these cases the history prefix observed by the client. In fact, recovery uses stable view change messages whose history elements are verified by $b+1$ check messages in CH with consistent positive outcomes (Lines 3.9 – 3.10). Clients only deliver a speculative reply if all the repliers, including the non-Byzantine primary, have the same message history of ORD-REQ messages. This and the properties (P1) and (P2) of the check phase ensure that the history element observed by the client is verified and recovered.

The second case is when the old primary p is Byzantine. This implies that at most $b - 1$ Byzantine repliers are included in the two sets reporting the two different ordered candidates. Two correct repliers have thus received inconsistent histories from the primary. This inconsistency is detected by the client by checking the history digest of the SPEC-REP messages. Therefore the client does not deliver the reply, a contradiction.

4.2.3 Why is Validity ensured?

We have argued above that after view change, the largest history prefix h observed by any correct client in previous views is recovered and included into the initial history ih of the new view. Unlike other protocols, Scrooge allows a request to be included into ih even if it is only reported by b Byzantine-faulty replicas. As client requests are signed and Byzantine replicas can not forge the signature of correct clients, no request in ih is fabricated. This corresponds to the customary Validity condition satisfied by other BFT replication protocols, e.g. [8].

5 Evaluation and Comparison

We conduct a comparative evaluation of Scrooge with other existing protocols: the standard PBFT protocol and two state-of-the-art fast protocols with publicly-available implementation, Zyzzyva and Zyzzyva5. The goal of the evaluation is to show that, during normal executions, Scrooge does not introduce significant additional overheads in the critical path compared to other speculative protocols such as Zyzzyva and Zyzzyva5. The results prove that Scrooge has the same performance as Zyzzyva in fault-free runs. Scrooge also improves over the performance of Zyzzyva in presence of unresponsive replicas, reaching the same performance as Zyzzyva5 but with less replicas.

Scrooge adds two types of overhead in the critical path. First, it uses larger history elements which include authenticators. This increases the overhead of executing the history digests included in the speculative replies. Second, speculative replies must include a bitmap representing the current replier quorum. The purpose of the evaluation is to show that these additional overheads are negligible.

A thorough comparison between quorum-based and speculative algorithms can be found in [16]. Just to give a point of reference, however, we scaled the performance figures of Q/U of [1] to our setting.

5.1 Optimizations

Scrooge uses similar optimizations as PBFT and Zyzzyva to improve the performance of the protocol. The main difference between Zyzzyva and Scrooge is the read-only optimization, which lets client clients try to send read-only requests directly to the replicas, which reply to the request without having the primary order them. If this does not succeed, the client sends the read as a regular request [2, 9]. In Scrooge, the optimization succeeds if clients receive $N - f$ consistent replies from replicas in the same replier quorum. In Zyzzyva, all replicas need to send consistent replies for the read optimization to succeed.

The Zyzzyva library uses a *commit optimization* to optimize the performance with unresponsive replicas. If a client cannot receive speculative replies from all replicas for a request, it explicitly notifies replicas to stop using speculation

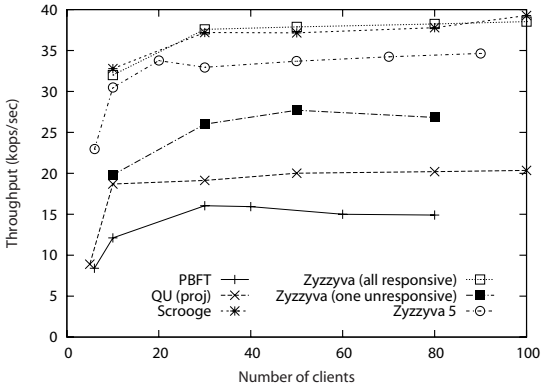

Figure 3: Throughput for 0/0 microbenchmark without batching and with $f = 1$.

for successive requests and to use instead an all-to-all agreement round that is similar to the prepare phase of PBFT. Without commit optimization, the performance of Zyzzyva with unresponsive replicas degrades [9].

Batching improves the performance of BFT algorithms under high load by letting replicas execute the protocol on groups of client requests [2]. However, batching makes it more difficult to compare algorithms as large batches make the performance of all algorithms converge [16], making an objective comparison more difficult.

PBFT, Zyzzyva and Zyzzyva5 use MACs for client requests but they are vulnerable to client attacks [4]. Scrooge tolerates such attacks by using signed requests. A fair experimental comparison would require running all algorithms with signed requests. However, the use of signatures for client requests impacts all protocols in a similar manner, making their performance figures closer. Therefore, we conduct this comparative evaluation assuming that all algorithms use MACs.

5.2 Evaluation setup

Our setting tolerates a single fault ($f = b = 1$). PBFT, Zyzzyva and Scrooge use four replicas while Zyzzyva5 uses six. All machines in the experiments have Intel Core2DUO 6400 2.1GHz processors, 4 GB of memory and Intel E1000 network cards. Since servers are single-threaded processes, the server machines perform as if they had a single 2.1GHz processor available. We use Fedora Linux 8 with kernel version 2.6.23 and a Gigabit switched star network. We use MD5 to compute MACs and the AdHash library for incremental hashes as in [2, 9]. Measurements are initiated when performance is stable and the system has executed the first 10,000 operations, and are stopped when the successive 10,000 operations are completed. In order to test the performance of the protocols in isolation, we use the same X/Y micro-benchmark used by the authors of PBFT [2], where X and Y are the size (in KB) of client requests and replica

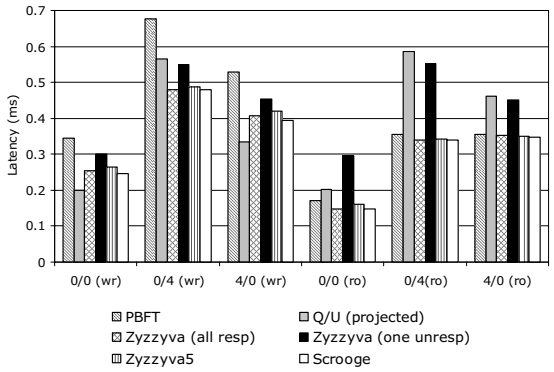

Figure 4: Latency for different benchmarks with a single client and no batching.

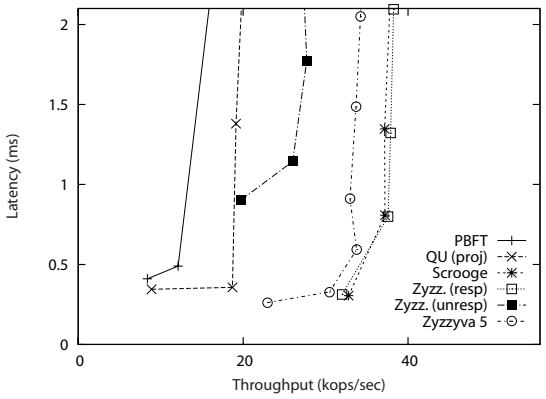

Figure 5: Latency-throughput curves for 0/0 microbenchmark without batching and with $f = 1$.

replies respectively.

We tested all protocols in scenarios where all replicas are responsive and where one replica is unresponsive because it is initially crashed.

5.3 Throughput

We first examine the throughput of Scrooge. Fig. 3 shows the throughput achieved by the 0/0 micro-benchmark without batching. Scrooge is the protocol which achieves the highest throughput with the lowest, and in this case minimal, number of replicas. Zyzzyva5 displays similar trends but a slightly lower peak throughput. This is probably due to the use of a larger number of replicas, which forces the primary to calculate a higher number of MACs (40% more than Scrooge) to authenticate order request messages. Zyzzyva can perform as well as Scrooge only in runs with all responsive replicas because it cannot otherwise use speculation. In runs with one unresponsive replica, the peak throughput improvement of Scrooge over Zyzzyva is more than one third. PBFT does has lower peak throughput because it calculates at least twice as many MACs as Scrooge and has quadratic message complexity.

If we consider read-only requests with one unresponsive replica the difference becomes even more evident because Zyzzyva is not able to use the read optimization, as previously discussed. Even if we use batches of size 10, thus reducing the relative difference among protocols, Zyzzyva achieves 52 kops/s peak throughput in presence of read-only workloads, whereas Scrooge achieves a peak of 85 kops/s.

5.4 Latency

The latency of different protocols using different micro-benchmarks is shown in Fig. 4. Scrooge performs in line with Zyzzyva5 with all micro-benchmarks. PBFT has approximately 40% higher latency than Scrooge for write requests and similar latency as Scrooge for read-only requests. Zyzzyva suffers a significant performance degradation in runs with unresponsive replicas. In case of write requests the difference with Scrooge ranges between 14% for the 0/4 case to 22% for the 0/0 case. The difference becomes much higher for read-only operations because unresponsive replicas disable the read-only optimization. The time a client needs to wait when it tries to use the read optimization without success depends on the timer settings of the client and is hard to evaluate. Fig. 4 only considers for Zyzzyva the optimistic latency given by processing read requests upfront as normal writes. Even in this scenario, the latency of Zyzzyva compared to Scrooge is 29% higher in the 0/0 case and up to 98% higher for the 4/0 case.

Fig. 5 illustrates how latency scales with the throughput when batching is not used. Scrooge is the protocol achieving the best latency at lowest, and in this case minimal, cost. Scrooge and Zyzzyva5 have almost equal measurement results. Zyzzyva displays higher latency in runs with unresponsive replicas starting with an initial load of 10 clients (~ 0.9 kops/sec).

5.5 Fault scalability

A fault scalable replication protocol keeps costs low when the number of replicas, and thus of tolerated faults, grows [1]. Scrooge is the most fault-scalable primary-based protocol in presence of unresponsive replicas. In Scrooge a primary computes $2 + (4f - 1)/s$ MACs operations per request if $b = f$ and s is the size of a batch. This is also the number of messages sent and received by the primary. Zyzzyva has a slightly lower overhead in fault-free runs, $2 + 3f/s$. Scrooge is more scalable than PBFT ($2 + 8f/s$) and Zyzzyva5 and Zyzzyva with one unresponsive replica and with commit optimization ($2+5f/s$). In Q/U, however, the bottleneck replica makes only 2 MACs operations per request. Note that renouncing to the commit optimization in Zyzzyva results in linear complexity but in significantly lower performance with unresponsive replicas [9].

Scrooge uses $1 + 3f + (4f - 1)/s$ messages per request, approximately the same as Zyzzyva in fault-free runs ($2+3f+3f/s$), Zyzzyva5 ($2+4f+5f/s$) and, if batching is not used, Q/U ($2+8f$). Both PBFT and Zyzzyva with unresponsive replicas and commit optimization have quadratic message complexity.

6 Related Work

We have already provided a comparison of Scrooge with PBFT [2], Zyzzyva [9] and DGV [7] throughout the paper.

In [8] a framework is proposed where different protocols can be combined to react to different systems conditions. The authors present two new protocols which improve the latency or the throughput of BFT replication in fault-free runs where specific preconditions are met (e.g. clients do not submit requests concurrently). In presence of unresponsive replicas, these protocols need to switch to a backup protocol such as PBFT.

Protocols like Q/U [1] and HQ [6] let clients directly interact with the replicas to establish an execution order. This reduces latency in some case but is more expensive in terms of of MACs operations [9, 16].

Preferred quorums is an optimization used by clients in some quorum-based BFT replication protocol to reduce the cryptographic overhead or to keep persistent data of previous operations of the client [6, 1]. Preferred quorums are not agreed-upon using reconfigurations and are not used during view change. This technique is thus fundamentally different from replier quorums because using (or not using) it has no effect on the replication cost of the protocol.

Some papers, such as [5, 14, 3], propose reducing the redundancy costs of BFT replication by assuming a stronger system model where trusted components, and sometimes synchronous networks, are used.

In a previous workshop paper [15] we discussed the motivations of Scrooge and only announced the results of this paper. The Scrooge protocol is presented here for the first time together with its evaluation.

7 Conclusions

Current protocols for BFT state machine replication require making a tradeoff between optimal performance and replication costs. Scrooge mitigates this tradeoff by two novel techniques: replier quorums and message histories. Evaluations show that, compared with Scrooge, PBFT is less performant, Zyzzyva matches its performance only in fault-free runs, and Zyzzyva5 has similar performance but higher replication costs. For the first time, in systems where tolerating any number of crashes and one Byzantine failure is sufficient, Scrooge reaches fast agreement with unresponsive replicas using a minimal number of replicas.

Acknowledgements

The authors are grateful to Ramakrishna Kotla, Lorenzo Alvisi and Mike Dahlin for sharing the Zyzzyva source code, and to Miguel Castro, Rodrigo Rodrigues, James Cowling and Barbara Liskov for sharing the PBFT source code. We also thank Flavio Junqueira for his comments on a draft of the paper.

References

[1] M. Abdel-El-Malek, G. Ganger, G. Goodson, M. Reiter and J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," *SOSP*, pp. 59–74, 2005.

[2] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and Proactive Recovery," *ACM TOCS*, 20(4), pp. 398–461, 2002.

[3] B.G. Chun, P. Maniatis, S. Shenker and J. Kubiatowicz "Attested append-only memory: making adversaries stick to their word," *SOSP*, pp. 189–204, 2007.

[4] A. Clement, M. Marchetti, E. Wong, L. Alvisi and M. Dahlin, "Making Byzantine Fault Tolerant System Tolerate Byzantine Faults," *NSDI*, 2008.

[5] M. Correia, N.F. Neves and P. Verissimo " How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems," *SRDS*, pp. 174–183, 2004.

[6] J. Cowling, D. Myers, B. Liskov, R. Rodrigues and L. Shrira, "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance," *OSDI*, pp. 177–190, 2006.

[7] P. Dutta, R. Guerraoui and M. Vukolic, "Best-case complexity of asynchronous Byzantine consensus," *EPFL T.R.*, 2004.

[8] R. Guerraoui, V. Quéma and M. Vukolić, "The next 700 BFT protocols," *EPFL LPD-REPORT-2008-08*.

[9] R. Kotla, L. Alvisi, M. Dahlin, A. Clement and E. Wong, "Zyzzyva: Speculative Byzantine Fault Tolerance," *SOSP*, pp. 45–58, 2007.

[10] L. Lamport, "Lower Bounds for Asynchronous Consensus," *FuDiCo workshop*, pp. 22-23, 2003.

[11] J-P. Martin and L. Alvisi, "Fast Byzantine Consensus," *IEEE TDSC*, 3(3), pp. 202–215, 2006.

[12] F. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Comp. Surv.*, 22(4), pp. 299–319, 1990.

[13] M. Serafini, P. Bokor, D. Dobre, M. Majuntke and N. Suri, "Scrooge: Reducing the Costs of Fast Byzantine Replication in Presence of Unresponsive Replicas," *TR-TUD-DEEDS-07-02-2009*, 2009. *www.deeds.informatik.tu-darmstadt.de/marco/Scrooge.pdf*

[14] M. Serafini and N. Suri, "The Fail-Heterogeneous Architectural Model," *SRDS*, pp. 103–113, 2007.

[15] M. Serafini and N. Suri, "Reducing the Costs of Large-Scale BFT Replication", *LADIS*, 2008.

[16] A. Singh, T. Das, P. Maniatis, P. Druschel and T. Roscoe, "BFT Protocols Under Fire," *NSDI*, pp. 189–204, 2008.

A Correctness of the Scrooge protocol

In this Section, we prove the correctness of the simplified Scrooge protocol. In the next two sections we describe the full version of Scrooge by extending the simplified version, and we prove that the introduced modifications preserve correctness. As customary, we first prove that the protocol never violates some invariant properties (safety) and that the protocol eventually achieves some useful results (liveness).

A.1 Replica state and definitions

We consider systems composed by $N \geq 2f+2b$ replicas. At most $f > 0$ replicas can be faulty and at most b can be Byzantine, with $0 < b \leq f$, while the others only crash.

We reason in terms of the *message history*, or simply history, stored by replicas. A history is an array indexed by a unique *sequence number* n. Each history element is a triple including the following fields:

- a client request m
- a replier quorum RQ
- a primary authenticator μ_p

Replicas go through a sequence of views, and accept messages of the agreement protocol for a view $v > 0$ only if after the view change to v is completed. In this case the view is called is *established*. We say that a replica is *in view* v if v is its last established view. If a correct replica i in view v participates to a view change to a view $v' > v$, it can build a *tentative history* for v', which is denoted as *t-hist(v, i)*. Tentative histories are indicated by the primary of the new view v'. When a correct replica i establishes view v', it agrees with the other replicas on the tentative history (Lines 2.30 – 2.34), which then becomes an *established history* and is denoted as *e-hist(v, i)*. Each view v has a predefined primary $p(v)$. In the pseudocode, both tentative and established histories are denoted by ih during view change.

A *history prefix in view* v *for sequence number* n *and correct replica* i is denoted as *prefix(n, v, i)* and is defined as the subset of the message history elements stored in view v by i with sequence number $n' \leq n$. Given two histories h and h', h *is a prefix of* h' iff for each element of h there is also one identical element of h' associated with the same sequence number. Furthermore, h is a *request prefix of* h' iff for each history element of h there is an element in h' associated with the same sequence number, client request and replier quorum.

An *agreed history prefix in view* v *for sequence number* n *and correct replica* i is any history prefix in view v for n and i where $n \leq aw$ and aw is the agreement watermark of i at the end of view v. It is denoted as *a-prefix(n, v, i)*. Committed history prefixes *c-prefix(n, v, i)* are defined similarly.

Name	Description	Type	Init
v	current view	timestamp	0
RQ	replier quorum	set of pids	[0,N-f-1]
n	current seq. number	timestamp	0
mh	message history	array of $\langle$*req.,RQ,auth.*$\rangle$	$\perp$
h	history digests	array of digests	$\perp$
aw	agreed watermark	timestamp	0
cw	commit watermark	timestamp	0
SL	suspect list	set of f pids	[N-f,N-1]
n_{SL}	seq. number of SL	timestamp	0
v'	new view	timestamp	0
ih	initial history	array of $\langle m, RQ,$ *auth.*$\rangle$	$\perp$
E	view establishment certificate	set of $N-f$ signed EST-VIEW messages	default value

Table 4: Global Variables of a Replica

View establishment certificates are attached to each view change message. A view establishment certificate E received from a replica i is *valid for a view change message* vc if $E = vc.E$ and all its $N - f$ signed establish view messages ev from different replicas contain the same view number $ev.v = vc.v$, the same sequence number $ev.n \leq vc.aw$ and the same correct digest $ev.h$ of the history prefix $\{vc.mh[0], \ldots, vc.mh[ev.n]\}$.

A replier quorum $RQ' \neq \perp$ is said to be *valid for correct replica* i *in view* v *at sequence number* n if, given the highest sequence number $n_c \leq n$ such that *c-prefix(n_c, v, i)*, all the history elements with sequence number in $[n_c, n]$ contain RQ'. We denote this as ***RQ-valid(RQ', n, v, i)***. A replier quorum $RQ' \neq \perp$ is said to be *current for correct replica* i *in view* v *at request* n if replica i in view v has set RQ to RQ' when it handles the order request for sequence number n in Lines 1.15 – 1.26. We denote this as ***RQ-current(RQ', n, v, i)***. The difference between the two predicates is that the first refers to replier quorums stored in the history logged by a replica, while the second refers to the current replier quorum of the replica when an order request message is processed.

A correct client c completes an operation o after receiving replies from a quorum of replicas in view v which executed the request with the same sequence number n and history prefix h. We denote this as *complete(o, n, v, h, c)*.

A.2 Agreement and Helper procedures

Before discussing the correctness of the algorithm we provide the pseudocode of the reconfiguration phase (Alg. 4) and of the helper procedures (Alg. 5) together with the list of variables of the algorithm (Tab. 4).

A.3 Proof sketch

In this section we provide proof sketches for the safety and liveness properties of the protocol. For simplicity, we assume here that $f = b$.

Algorithm 4: Scrooge - Explicit agreement

```
4.1  procedure agree(m)
4.2      if ∃k : mh[k].m = m and never sent agree message for sequence
         number k in view v then
4.3          send ⟨AGREE, v, k, h[k], i⟩_μi to all replicas;
4.4          start timer if not already running;
4.5
4.6  upon client timeout
4.7      SL ← ⊥;
4.8      if ∃RQ : received matching speculative replies sp to m with
         sp.RQ = RQ from a set S ⊂ RQ of N − 2f replicas then
4.9          SL ← RQ \ S;
4.10     stop waiting for sp messages; timer ← timer · 2;
4.11     repeat
4.12         send m = ⟨REQ, o, t, c, SL⟩_σc to all replicas;
4.13     until client receives b + 1 matching stable replies st to m ;
4.14     deliver (o, t, st.r);
4.15
4.16 upon backup i receives request m from client m.c
4.17     if not IN-HISTORY(m, mh) then
4.18         send m to primary p(v);
4.19         start timer if not already running;
4.20     else if not COMMITTED(m, mh, cw) then agree(m);
4.21     else reply-cache(m.c);
4.22
4.23 upon replica i receives an agree message ag from replica ag.i
4.24     if ag.v = v and ag.h = h[ag.n] then
4.25         agree(mh[ag.n].m);
4.26         if received N − f − 1 matching agree messages for ag.n from
             other replicas then
4.27             send ⟨COMMIT, v_i, n, i⟩_μi to all replicas;
4.28             aw ← ag.n;
4.29
4.30 upon replica i receives a commit message cm from replica cm.i
4.31     if cm.v = v and cm.n ≤ aw and received N − f − 1 matching
         commit messages for cm.n from other replicas then
4.32         c ← mh[cm.n].m.c; t ← mh[cm.n].m.t;
4.33         r ← stored reply for mh[cm.n];
4.34         send ⟨STAB-REP, v, n′, c, t, r, i⟩_μi,c to client c;
4.35         if cw ≤ cm.n then
4.36             cw ← cm.n; RQ_cw ← mh[cw].RQ;
4.37             if ∀k ∈ [cw, n] : mh[k].RQ = RQ_cw then
4.38                 RQ ← RQ_cw;
4.39         if never sent agree message for sequence number n′ > cw and
             view v then stop timer;
4.40         send-missing-spec-rep(cw, RQ_cw);
4.41
4.42 upon replica timer expires
4.43     timer ← timer · 2;
4.44     view-change(v′ + 1) ;
4.45
```

Algorithm 5: Scrooge - Helper procedures

```
5.1  procedure update (SL′)
5.2      if n > n_SL and |SL′| ≤ f then
5.3          n_SL ← n;
5.4          if p(v) ∈ SL′ then SL′ ← SL′ \ {p(v)};
5.5          remove the |SL′| oldest elements from SL;
5.6          add elements of SL′ into SL;
5.7
5.8  procedure agree(m)
5.9      if ∃k : mh[k].m.c = m.c and mh[k].m.t = m.t and never sent
         agree message for sequence number k in view v then
5.10         send ⟨AGREE, v, k, h[k], i⟩_μi to all replicas;
5.11         start timer if not already running;
5.12
5.13 procedure reply-cache(c)
5.14     n′ ← sequence number of last committed operation from c;
5.15     r ← stored reply for mh[n′];
5.16     send ⟨STAB-REP, v, n′, c, mh[n′].t, rc[n′], i⟩_μi,c to client c;
5.17
5.18 procedure send-missing-spec-rep(k, RQ)
5.19     if i ∈ RQ then
5.20         while mh[k].RQ = RQ and never sent speculative reply for
             sequence number k in view v do
5.21             m ← mh[k].m; r ← stored reply for mh[k];
5.22             send ⟨SPEC-REP, v, k, h[k], RQ m.c, m.t, r, i⟩_μi to
                 client m.c; k ← k + 1;
5.23
```

A.3.1 Safety

The safety property of BFT replication protocols is that clients have the abstraction of interacting with a non-replicated server executing all requests according to a total order [12]. Therefore, if two clients issue two different requests and complete them, these must have been consistently ordered by the replicas which have generated the delivered replies. We now argue that this property holds within a view and across views.

Within a view: A client can complete requests in a view in two cases: either after receiving $3f$ matching speculative replies or after receiving $f + 1$ stable replies. If two clients complete a request, they receive speculative replies from two sets of replicas which intersect in one correct replica i. Since correct replicas execute requests in the order dictated by sequence numbers, replica i has established a local order between the two requests. A client completes a request from speculative replies only if $3f$ repliers have sent a consistent history digest. This implies that these replicas have the same history as i, so the requests are consistently ordered. A similar reasoning can be done if one of the client, or both, deliver after receiving stable replies. In fact, stable replies are agreed by replicas after receiving a set of $3f$ agreement messages with matching histories and any two such set intersect in a correct replica i.

Across views: If a client completes a request m after receiving replies for view v and sequence number n and a view change to view $v + 1$ occurs afterwards, m must be associated with n in the new view as well. This ensures that if a second client completes a request in $v + 1$, and by induction in any successive view too, the two requests will be consistently ordered. We now consider how m is recovered during view change.

If m has completed from $f + 1$ stable replies, at least one correct replica has received in view v matching commit messages for n from at least $2f$ correct replicas. These replicas have a common agreed history prefix including all elements with sequence numbers $n' \leq n$. During the view change protocol, at least f of these correct replicas will send a view change message to the new primary. Request m is thus included in one agreed candidate. However, at most f other faulty replicas might report a different agreed candidate. In order not to be discarded, agreed candidates must be contained in the history of at least $f + 1$ replicas, one of which is correct. If this holds for both candidates, it im-

plies that the primary has sent two different ordered request messages for n and is thus one of the f replicas reporting a different candidate. The view change message from the primary is discarded in this case and a view change message from a correct replica, which can only report m as agreed candidate for n, is awaited. After that, only the agreed candidate containing m remains and is selected by the recovery function.

If m has completed after the client has received $3f$ speculative replies from a replier quorum RQ, at least $2f$ correct repliers i had updated their current quorum RQ_i to RQ when they had committed on an agreement watermark $n_a^i < n$. The history element for n_a^i, and thus the replier quorum RQ contained in it, can thus be recovered. Before delivering, the client has also checked that all the $2f$ correct repliers i have a common history consisting of elements with sequence numbers $n' \leq n$. Also, as the request has been speculatively replied by all correct replicas in RQ, the SPEC-RUN predicate was true when the repliers received the order request message for n. This implies that RQ_i was not set to $\perp$ by any of the correct repliers. The replier quorum RQ is thus contained in all history elements stored with sequence number n' such that $n_a^i \leq n' \leq n$ by each correct replier. By induction on n', RQ_{n-1} is set equal to RQ when a agreement is reached for n. During view change, the primary will receive at least f view change messages from elements in RQ_{n-1} associating m to n. The request m is thus associated with an ordered candidate, which is selected for sequence number n by the recovery function and included into the initial history of v' as discussed in Section 4.2. If a different request m' is reported as agreed candidate by f faulty replicas and is included in the histories of $f + 1$ view change messages, this indicates that the old primary of view v was faulty and has sent inconsistent ordered requests. Its message is thus discarded and another message from a correct replica is awaited. After this is received, m' is not an agreed candidate any longer as correct replicas can only agree on m.

A.3.2 Liveness

Liveness is guaranteed when the system is in timely periods and thus the same view can be established by all correct replicas. If a client c cannot complete its request m from speculative replies, it resends m to all replicas. If the primary is correct, the SPEC-RUN condition ensures that the agreement and commit phases are executed by each correct replica once one correct replica receives both m from c and the corresponding order request message from the primary. All correct replicas initiate and complete the agreement and commit phases on the entire history prior to m and send to $f + 1$ matching stable replies to c. These are sufficient to complete the request. If this does not eventually happen, the primary is faulty and at least $f + 1$ correct replicas accuse it. This is sufficient to let all replicas initiate a view change.

The protocol cannot block during view change in timely periods. A new correct primary can always wait for $3f$ well-formed view change messages from correct replicas. Each of them eventually becomes stable as the outcome vectors o contain binary boolean values and the new primary receives outcome vectors from $3f$ correct replicas for each of these view change messages. Additional messages are waited if there are multiple candidates for a sequence number and one of them is included in the view change message sent by the primary of the last established view v. This implies that at least one faulty replica has sent a view change message and is thus possible to wait for one additional view change message. Also, if an agreed candidate e is sent by correct replicas, at least $2f$ correct replicas have it in their local history. If an incorrect agreed candidate is reported by a faulty replica, all $3f$ correct replicas send view change messages reporting that no agreement on it was reached. In both cases progress is ensured.

A.4 Scrooge safety

The safety property provided by BFT replication protocols is that different correct clients always observe consistent histories, as reflected in the following *safety property*.

Property 1 *For each pair of correct clients c_1 and c_2 and for each pair of operations o_1 and o_2 completed by client c_1 and c_2 respectively, let n_1 and n_2 be the sequence numbers associated with o_1 and o_2 respectively and h_1 and h_2 the history prefixes stored in any view by any correct replica for n_1 and n_2 respectively. If $n_1 \leq n_2$, then h_1 is a request prefix of h_2.*

The purpose of this section is to prove that Property 1 is an invariant.

We first prove some consistency properties within a view v, and then we show how consistency is preserved across views.

Lemma 1 If h = *e-hist(v, i)* and h' = *e-hist(v, j)*, then $h = h'$.

Proof: If $v = 0$ all replicas have the same established initial history, which is empty.

If $v > 0$, replicas consider their initial history for view v as established in Lines 2.30 – 2.34 after having received valid establish view messages with matching history digests from a quorum of $N - f$ replicas, including at least $N - f - b \geq f + b$ correct replicas. If two correct replicas i and j have established the same view v, there are at least $b > 0$ correct replicas k in the intersection of the quorums of i and j. Since a correct replica accepts only one new view message per view in Lines 2.24 – 2.28, it never sends two different establish view message. The initial histories for i and j are thus the same.

Lemma 2 If h = *prefix(n, v, i)* and h' = *prefix(n', v, i)* and $n \leq n'$ then h is a prefix of h'.

Proof: A correct replica i in view v adds an entry to its history only upon receiving order request messages from the current primary $p(v)$ (Lines 1.15 – 1.26). The entry for sequence number n is added only if n is the smallest sequence number not yet associated with an entry in the history of i. This implies that (a) only one entry can be associated with a given sequence number in a history in view v, (b) there are no gaps and (c) if h is the history prefix for sequence number n, requests for higher sequence numbers n' added in view v are appended to h.

Lemma 3 If *RQ-current(Q, n, v, i)* then *RQ-valid(Q, n, v, i)*.

Proof: Let n_c be the highest sequence number smaller than n such that *c-prefix(n_c, v, i)* and assume by contradiction that *RQ-valid(Q, $n-1$, v, i)* does not hold. This implies that some history element with sequence number in $[n_c, n-1]$ contains a replier quorum $S \neq Q$. Let $n_S \geq n_c$ be the highest sequence number of such an element. From *RQ-current(Q, n, v, i)* it follows that $RQ = Q$ for i when the order request with sequence number n is processed by i in view v (Lines 1.15 – 1.26). Order requests are processed following their sequence numbers and the replier quorum RQ is set to $\perp$ if the predicate SPEC-RUN does not hold. RQ is set to a value $Q \neq \perp$ in view v only when a new commit watermark is reached in v and all history elements from the commit watermark up to the current sequence number are associated with Q in the message history of i (Lines 4.30 – 4.40). Therefore, *RQ-current(Q, n, v, i)* implies that (a) there exists a sequence number $n_Q < n$ such that a commit on n_Q is reached in view v before an order request with sequence number $k < n$ is processed and Q is associated to all history elements in $[n_Q, k-1]$, and (b) RQ is not set to $\perp$ when order request with sequence numbers in $[k, n]$ are processed, so SPEC-RUN holds and $RQ_p = Q$ for all the corresponding history elements. This implies that $n_Q > n_S \geq n_c$, which contradicts the definition of n_c.

Lemma 4 If a correct replica i in view v sends a speculative reply for a request associated with sequence number n in its history, then there exists a replier quorum Q such that *RQ-valid(Q, n, v, i)* and $i \in Q$.

Proof: If i sends the speculative reply in Lines 1.15 – 1.26 it follows from SPEC-RUN that it received an order request message for n from the primary $p(v)$ containing a replier quorum $RQ \neq \perp$ such that *RQ-current(RQ, n, v, i)* holds and $i \in Q$. The result thus follows from Lemma 3.

If i sends the speculative reply after a commit in Line 4.40, the result follows from the fact that the procedure *send-missing-spec-rep* checks that $i \in Q$ and that Q is associated to the history element with the highest committed sequence number $n_c \leq n$ and to all history elements with sequence numbers in $[n_c, n]$. This ensures that each commit with sequence number in $[n_c, n]$ is associated with Q, as required by *RQ-valid(Q, n, v, i)*.

Lemma 5 If h = *a-prefix(n, v, i)* then either h is a prefix of *e-hist(v, i)* or there exist $N - f - b$ correct replicas j such that h = *prefix(n, v, j)*.

Proof: A correct replica i can update its agreed watermark to $aw \geq n$ in two cases: when it establishes the view v in Lines 2.30 – 2.34 or when it completes an agreement phase for sequence number aw in Lines 4.23 – 4.28. In the first case, h is a prefix of *e-hist(v, i)* as from Lines 2.30 – 2.34. In the second case, replica i has received equal agree messages containing a history digest for h and sequence number n from a quorum Q of $N - f$ replicas in view v. At least $N - f - b$ replicas in Q are correct and have thus sent matching agree messages only if h = *prefix(n, v, j)*.

Lemma 6 If a_i = *a-prefix(n_i, v, i)* and a_j = *a-prefix(n_j, v, j)* and $n_i \leq n_j$, then a_i is a prefix of a_j.

Proof: From a_i = *a-prefix(n_i, v, i)* and Lemma 5 either a_i is a prefix of *e-hist(v, i)* or there exist at least $N - f - b$ correct replicas k such that a_i = *prefix(n_i, v, k)*.

In the first case, it follows that if any correct replica j has a history prefix a_j in view v, then *e-hist(v, j)* is a prefix of a_j (from Lemma 1) and *e-hist(v, j)* = *e-hist(v, i)* (from Lemma 2). It thus follows that a_i is a prefix of a_j.

In the second case, it follows from Lemma 5 that there exists a set of at least $N - f - b \geq f + b$ correct replicas k such that a_i = *prefix(n_i, v, k)*. As $n_j \geq n_i$, replica j also sets its agreement watermark in Lines 4.23 – 4.28 after receiving agree messages from at least $b > 0$ of these correct replicas k reporting that $a_j = a_k$ = *prefix(n_j, v, k)*. Since k is correct, from Lemma 2 and $n_j \geq n_i$ it follows that *prefix(n_i, v, k)* is a prefix of *prefix(n_j, v, k)*, and thus a_i is a prefix of a_j.

Lemma 7 If h = *a-prefix(n, v, i)* and there exist at least $f + 1$ correct replicas j and a history prefix h' = *prefix(n', v, j)* for $n' \geq n$, then h is a prefix of h'.

Proof: From h = *a-prefix(n, v, i)* and Lemma 5 either h is a prefix of *e-hist(v, i)* or there exist $N - f - b$ correct replicas k such that h = *prefix(n, v, k)*.

In the first case, it follows that if any correct replica j has a history prefix h' in view v, then *e-hist(v, j)* is a prefix of h' (from Lemma 1) and *e-hist(v, j)* = *e-hist(v, i)* (from Lemma 2). It thus follows that h is a prefix of h'.

In the second case, a set S of at least $N - f - b$ correct replicas k have h = *prefix(n, v, k)*. From Lemma 2 it follows that if any of these replicas has an history prefix h'' =

prefix(n′, v, k) for $n' \geq n$, then h is a prefix of h''. Since each set of $f+1$ correct replicas j intersect with one correct replica in S, their common history prefix h' is equal to h''.

We can now show how the protocol preserves consistency across views. The following lemmas are the core lemmas to prove the safety of the protocol.

Lemma 8 If ih = *t-hist(v + 1, i)* and there exist a sequence number n and $N - f$ replicas j such that *a-prefix(n_j, v, j)* for $n_j \geq n$ if j is correct, then all j have the same history prefix hp = *prefix(n, v, j)* and hp is a request prefix of ih.

Proof: We consider the case where i is the primary of view $v + 1$. The case of the other backup replicas is similar since the same decision procedure *recover* used by the primary to recover ih is used by the backups.

In Lines 2.7 – 2.18 of the view change to view $v + 1$, the primary of the new view $v + 1$ receives view change messages vc with message histories $vc.mh$ including h' as a prefix and with $vc.aw \geq n$ from at least $N - 2f - b$ of the at least $N - f - b$ correct replicas j. Since v is the highest established view smaller than $v + 1$ and since it is contained in $N - 2f - b \geq b > 0$ view change messages in VC, v is selected as the highest previous established view mv in Lines 3.11 – 3.15. From Lemmas 1 and 2, *e-hist(v, i)* is a prefix of hp. Also, *e-hist(v, i)* is a prefix of ih because $mv = v$ implies that the initial history ih is set to *e-hist(v, i)* in Lines 3.11 – 3.15. We now prove that the suffix of hp which is not included in *e-hist(v, i)* is also included in ih in Lines 3.16 – 3.31.

From the hypothesis, there exist at most f correct replicas l such that $h' \neq$ *a-prefix(n, v, l)*. For each history element e with sequence number $n' \leq n$ in the suffix of hp, AGREED-CAND(e, n', v, VC) holds. In fact, apart at most b Byzantine replicas and f correct replicas, all other correct replicas j send view change messages vc to the new primary of v' such that $vc.v = mv = v$, $e = vc.mh[n']$ and $vc.aw \geq n$. This is because for all replicas j, hp = *a-prefix(n, v, j)* as $n_j \geq n$ for all j and from Lemma 6. We show that e is the only possible agreed candidate which is selected for n' in ih in Lines 3.23 – 3.24.

Assume by contradiction that a candidate $g \neq e$ is selected for n'. As e is an agreed candidate, g must be an agreed candidate which is selected in Lines 3.23 – 3.24 and predicates WAIT-AGR(A, n', v, VC) and WAIT-ORD(A, O, n', v, VC) are false. As g is an agreed candidate, $b + 1$ replicas, including at least a correct one, have received from the primary of view v a different order request for n' than the replicas which agreed on e. The old primary $p(v)$ of view v is thus Byzantine. From Lemma 6, no correct replica can send a view change message with g in its agreed history prefix for n'. By definition of AGREED-CAND, g is an agreed candidate only if all $|VC| - f - b \geq b$ Byzantine replicas, including the primary, have sent a view change message with g in the agreed history prefix for n' and v and all these messages are in VC. As WAIT-ORD is false and there are multiple different candidates, the new primary of view $v + 1$ waits until $|VC| > N - f$. This implies that g is not selected as an agreed candidate as g is included in the agreed prefix of view change messages in VC sent by at most b replicas but $|VC| - f - b > N - 2f - b \geq b$.

Lemma 9 If ih = *t-hist(v + 1, i)* and there exist a history prefix hp and a replier quorum RQ such that $|RQ| = N - f$ and $Q \subseteq RQ$ is the subset of all correct replicas in RQ and for each $j \in Q$, hp = *prefix(n, v, j)* and *RQ-valid(RQ, n, v, j)*, then hp is a request prefix of ih.

Proof: We consider the case where i is the primary of view $v + 1$. The case of the other backup replicas is similar since the same *recover* function used by the primary to recover ih is used the the backups.

Let l be the replica having the the smallest agreement watermark aw_l among the replicas in Q. As Q contains $N - f - b$ correct replicas, Lemma 8 implies that the history prefix h' = *prefix(aw_l, v, l)* is a prefix of the initial history ih. If $n \leq aw_l$, then it follows from Lemma 2 that hp is a prefix of h' and we are done. If $n > aw_l$ then *RQ-valid(RQ, n, v, l)* implies by definition that RQ is contained in the history element s of h' for aw_l, which is the highest agreement watermark $\leq n$ of replica l. It thus follows from Lemma 8 that RQ is contained in the candidate which is selected for sequence number aw_l and subsequently used to identify the observed candidates for sequence number $aw_l + 1$. We now prove by induction that for all sequence numbers n' such that $aw_l < n' \leq n$, the client request included in hp for n' is selected for the initial history ih.

The inductive hypothesis implies that $RQ_{n'-1} = RQ$. As $Q \subseteq RQ$ contains at least $N - f - b$ correct replicas, the new primary of view $v + 1$ receives view change messages from at least b of them. All these replicas report the same history element for n', which is thus a candidate e. The replier quorum RQ is included in e since for all $j \in Q$ it holds that *RQ-valid(RQ, n, v, j)*. Assume by contradiction that a candidate $g \neq e$ is selected for n'. The candidate g must be selected in one of the three cases of Lines 3.23 – 3.28. We show that in each of these cases we reach a contradiction.

If g is selected in Lines 3.23 – 3.24, this implies that g is an agreed candidate, AGREED-CAND(g, n', v, VC) holds and the predicates WAIT-AGR(A, n', v, VC) and WAIT-ORD(A, O, n', v, VC) are false. As AGREED-CAND holds, g is included in the local history of $b + 1$ replicas, including a correct one. This implies that g, as well as e, has been associated by the old primary $p(v)$ to

sequence number n. The old primary $p(v)$ has thus sent inconsistent order request messages for n' and is thus Byzantine. From Lemma 7, as Q contains $N - f - b \geq f + b > f$ replicas, only Byzantine replicas can claim to have agreed on g for n' and view v. It follows from AGREED-CAND that all $|VC| - f - b \geq b$ Byzantine replicas, including the primary, have included g in the agreed history prefixes of their view change messages, and all these view change messages are included in VC. If g is selected then WAIT-ORD is false and $|VC| > N - f$ as there are two different candidates. In order to be selected as an agreed candidate, g there must be at least one correct replica which has agreed on g. This contradicts Lemma 7 as Q contains more than f correct replicas.

If g is an observed candidate selected in Lines 3.25 – 3.28, either g satisfies VERIFIED(g, VC, CH) because at least one correct replica was able to verify that the corresponding order request message was generated from the old primary $p(v)$, or e does not satisfy VERIFIED(e, VC, CH). This in turns implies that $p(v)$ is Byzantine. In fact, if $p(v)$ were correct then $p(v) \in Q$, since ***RQ-valid(****RQ, n′, v, j)* holds for some correct replica j which always checks that the primary of a view is a member of the replier quorums in that view. All replicas $j \in Q$ would have the same history prefix hp = ***prefix(****n, v, j)* as $p(v)$, including the same authenticator which was generated by $p(v)$ for the order request message corresponding to e. The candidate e, which would be the only one generated by primary $p(v)$, would thus be the only verified candidate, a contradiction.

By hypothesis, only faulty replicas in $Q = RQ_{n'-1}$ can associate g to n'. In order to be a candidate, g must be associated to n' in the view change messages sent to the new primary of v by all the $|VC| - f - b \geq b$ Byzantine replicas, including at the old primary $p(v)$, and all these messages must be included in VC. As there are multiple candidates and WAIT-ORD must be false for a candidate to be selected, $|VC| > N - f$. From the definition of ORDERED-CAND, g must be contained in the local history of at least $|VC| - f - b > b$ replicas in RQ_{n-1}. As there are at most b Byzantine replicas, g is associated to n' in the message history of at least one correct replica in RQ_{n-1}, which is also included in Q by definition. This contradicts the fact that $e \neq g$ corresponds to a common history element for n' and for all replicas $j \in Q$.

Lemma 10 If complete(o, n, v, h, c) and h' = ***t-hist(****v + 1, i)* then h is a request prefix of h'.

Proof: A client completes the request m in Lines 1.28 – 1.30 or in Lines 4.13 – 4.14.

If the client completes o in Lines 4.13 – 4.14, it has received $b + 1$ stable replies from correct replicas j whose committed prefixes include h as a prefix and whose commit watermarks are $n_j \geq n$. This implies that at least one correct replica has received in Lines 4.30 – 4.40 consistent agree messages for its agreed history prefix from at least $N - f$ replicas. From Lemma 8 it follows that for each correct replica i, h is a prefix of h'.

If the client completes o in Lines 1.28 – 1.304, it has received speculative replies from a set RQ of $3f$ replicas j claiming to have the same history prefix h = ***prefix (****n, v, j)* and to be members of the same replier quorum RQ such that $p(v) \in RQ$. Let Q be the subset of correct replicas in RQ. From Lemma 4, for each $j \in Q$, ***RQ-valid(****RQ, n, v, j)* and $j \in RQ$. Therefore, $Q \subseteq RQ$. From Lemma 9, it follows that for each correct replica i, h is a prefix of h'.

Lemma 11 If complete(o, n, v, h, c) and h' = ***t-hist(****v′, i)* and $v < v'$ then h is a request prefix of h'.

Proof: Assume by contradiction that h is not a prefix of h'. If no correct replica had established a view v'' such that $v < v'' < v'$, then all correct replicas would send for view v' the same view change messages as the ones sent for view $v + 1$ except from the new view field. A contradiction would thus follow from Lemma 10. Therefore, the primary of view v' must have received view change messages from some replicas j having a valid view establishment certificate for an established view v_j with $v < v_j < v'$ and for a corresponding established history h_j = ***e-hist(****v_j, j)* = ***t-hist(****v_j, j)*. Let k be, among the replicas j, the replica which sends the view change message with the highest established view v_k. This implies that h_k is selected as initial history ih_{v_k} by the recover function. From ***complete(****o, n, v, h, c)* and $v < v_k$ it follows that if h is not a prefix of h', then h is not a prefix of h_k = ***t-hist(****v_j, j)*. This argument for v' can be inductively be applied to v_k. By induction on the largest established view $v'' < v'$ reported to the new primary of view v'', h is not a prefix of ***t-hist(****v″, i)*. Let v_i be the smallest view $v'' > v$ established by any correct replica i. All correct replicas send for view v_i the same view change messages as the ones sent for view $v + 1$ except from the new view field, but h is not a prefix of ***t-hist(****v_i, i)*. This contradicts Lemma 10.

Lemma 12 If complete(o, n, v, h, c) and complete(o', n', v, h', c') and $n \leq n'$ then h is a request prefix of h'.

Proof: Two clients c and c' can complete a request either in Lines 1.28 – 1.30 or in Lines 4.13 – 4.14. If client c completes a request after receiving $b + 1$ stable replies in Lines 4.13 – 4.14, then h = ***a-prefix(****n, v, i)* for at least $N - f - b$ correct replicas i. If client c' completes a request after receiving $b + 1$ stable replies in Lines 4.13 – 4.14, then h' = ***a-prefix(****n′, v, j)* for at least $N - f - b$ correct replicas j. From Lemma 6 and $n \leq n'$ it follows that h is a prefix of h'. If client c' delivers from $3f$ speculative replies in Lines 1.28 – 1.30, then h' = ***prefix(****n′, v, j)* for at least

$N - f - b$ correct replicas j. From Lemma 7, it follows that h is a prefix of h.

If client c completes after receiving $N - f$ speculative replies in Lines 1.28 – 1.30, then h = *prefix(n, v, i)* for a set Q of at least $N - f - b$ correct replicas i. In order to deliver either Lines 1.28 – 1.30, client c' must receive one reply from at least one correct replica $i \in Q$, and the result follows from Lemma 2. If c' completes a request after receiving $b + 1$ stable replies in Lines 4.13 – 4.14, then h' = *a-prefix(n', v, j)* for at least $N - f - b$ correct replicas j, including at least one replica in Q. From Lemma 2, this implies that h is a prefix of h'.

Lemma 13 If complete(o, n, v, h, c) and complete(o', n', v', h', c') and $n \leq n'$ and $v < v'$, then h is a request prefix of h'.

Proof: If client c' completes a request in view v', this implies that it receives speculative or stable replies from at least one correct replica i in view v' and that h' = *prefix(n', v', i)*. Since this replica has established v', there exists an established history h'' = *e-hist(v', i)* = *t-hist(v', i)* = *prefix(n'', v', i)*. From *complete(o, n, v, h, c)*, $v < v'$ and Lemma 11, h is a prefix of h''. It follows that h = *prefix(n, v', i)* and thus that h, h' and h'' are all prefixes of i in view v'. As $n \leq n'$, h is a prefix of h' from Lemma 2.

Lemma 14 If complete(o, n, v, h, c) and complete(o', n', v', h', c') and $n \leq n'$ and $v > v'$, then h is a request prefix of h'.

Proof: If client c completes a request in view v, this implies that it receives speculative or stable replies from at least one correct replica i in view v such that h = *prefix(n, v, i)*. Since this replica has established v, there exists an established history h'' = *e-hist(v, i)* = *t-hist(v, i)* = *prefix(n'', v, i)*. From *complete(o', n', v', h', c')*, $v' < v$ and Lemma 11, h' is a prefix of h''. It follows that h' = *prefix(n, v, i)* and thus that h, h' and h'' are all prefixes of i in view v. As $n \leq n'$, h is a prefix of h' from Lemma 2.

Theorem 1 Property 1 holds.

Proof: Two clients can complete requests by receiving enough replies from replicas in the same view. If they complete their operations in the same view, the result follows from Lemma 12. Else, it follows from Lemmas 13 and 14.

A.5 Scrooge liveness

The liveness property of Scrooge is the following:

Property 2 *If a correct client issues a request, it eventually completes it.*

Additionally, Scrooge ensures the following property:

Property 3 *If the system is in a timely period, v is the current established view for all correct replicas, the primary of v is correct and faulty clients only crash, eventually all correct clients complete their requests from speculative replies.*

In the proofs, we assume that the system eventually enters a *timely period* where no timeout is fired and all sent messages are received. We first show that Property 2 holds.

Lemma 15 If the system is in a timely period and there exists a view v' such that the primary of v' is correct and all correct replicas initiate a view change to v', then all correct replicas eventually establish v'.

The view change protocol can block under this hypothesis because the *recover* function never completes correctly or because a new view can not be established. We now show that the protocol does not block in either case.

For recovery, the new primary will eventually receive $N - f$ well-formed view change and check messages from correct replicas. These also eventually satisfy the predicate STABLE as the vector res of each check message in CH contain binary values (see Lines 2.20 – 2.22) and each correct replica eventually sends its own outcome vector for each of these view change messages. As the system is composed of at least $N - f \geq f + 2b > 2b$ correct replicas, at least one of the two outcomes collects $b + 1$ check messages. Therefore, for recovery to block, either the predicate WAIT-ORD or the predicate WAIT-AGREED must still hold after $N - f$ view change and check messages are received from correct replicas and $|VC| \geq N - f$ (Lines 3.20 – 3.21).

For WAIT-ORD, let $mv < v'$ be the highest view $vc.v$ reported by a view change message $vc \in VC$ (Lines 3.11 – 3.15). If the old primary of $p(mv)$ were correct, each history element stored by a correct replica in view mv would be consistent with those of the primary of that view. If $p(mv)$ were correct and only correct replicas would have sent view change messages to the new primary of view v, there would not be inconsistent candidates. As multiple inconsistent candidates are present and one of them is sent by $p(mv)$, it follows that at least one Byzantine replica, either a backup which reports a forged element of the old primary, has sent a view change message which is included in $|VC|$. This implies tat $|VC| > N - f$ so WAIT-ORD does not hold.

For WAIT-AGREED to hold, the primary of view v' must have received an agreed candidate e for sequence number n and view mv. According to the predicate definition, this implies that (i) at most b correct replicas have associated e in their local history for n and mv is their last established view, and (ii) at most $f + b$ correct replicas have a view $v'' \neq mv$ as last established view or do not associate e to their agreed history prefix for sequence number n and view

mv. From (i) and from the fact that mv is the highest established view contained in any view change message received by the primary of the new view v, it follows that at least $f + b$ correct replicas have not yet established mv or have established mv but have not included e in their local history for n. Therefore, from Lemma 5, no correct replica in view mv can include e in their agreed history prefix for sequence number n. This implies that all $N - f$ correct replicas either have a view $v'' \neq mv$ as their last established view, or do not associate e to their agreed history prefix for sequence number n and view mv. This contradicts (ii). Therefore, WAIT-AGREED does not hold and the protocol does not block due to the recover function.

After recovery is concluded, the new correct primary sends new view messages to all correct replicas, which then compute the same decision procedure as the primary in Lines 2.24 – 2.28 and send consistent establish view messages. Lines 2.30 – 2.34 can thus be completed and the new view is established.

Lemma 16 If the system is in a timely period, v is the current view established by all correct replicas and a correct replica sends an agreement message for sequence number n and view v, then all other correct replicas eventually do the same.

Proof: If a correct replica executes Lines 5.9 – 5.11 for n, then every other replica will receive an agreement message for n. If the replica i has already received an order request for n, it starts agreement in Lines 4.23 – 4.28. Else, the agreement message makes the predicate AGREEMENT-STARTED(i, n, v) hold. Agreement is started by replica i when the replica receives the order request for sequence number n (Line 1.26).

Lemma 17 If the primary of a view v is correct, the system is in a timely period and v is the current view established by all correct replicas, then a view change is never initiated by a correct replica and all requests from correct clients are completed.

Proof: A correct replica which does not suspect the primary and never executes Lines 4.42 – 4.44 initiates view change only if at least another correct replica accuses the primary (Lines 2.15 – 2.16). A correct replica accuses the primary of the current view v in Lines 4.42 – 4.44 only if it starts an agreement phase for a sequence number n in Lines 5.9 – 5.11 and the timer expires. From Lemma 16, if a correct replica starts agreement each other correct replica do the same. If the primary is correct and the system is timely, the agreement phase is concluded by each correct replica (Lines 4.23 – 4.28). This implies that the commit phase is also concluded before the timer at any correct replica expires (Lines 4.30 – 4.40) and consistent stable replies are sent by all the $N - f$ correct replicas. A correct client can thus complete all its requests in Lines 4.13 – 4.14 by resending them to all replicas.

Theorem 2 Property 2 holds.

Proof: We consider the system behavior when the the system eventually enters a timely period. If the correct client cannot complete a request from speculative replies in view v in Lines 1.28 – 1.30, it contacts all correct replicas until it can deliver a reply in Lines 4.13 – 4.14. When correct backups receive from the client a request in Lines 4.16 – 4.21, they start a timer and accuse the primary if the reconfiguration phase is not completed when this expires (Lines 4.42 – 4.44). If the primary is correct, all correct replicas obtain an order request, start agreement in Lines 1.10 – 1.12 or 4.16 – 4.21 or 1.26, and complete the agreement and the commit phases. Since there are at least $N - f$ correct replicas in the system, the client can receive $b + 1$ consistent stable replies and complete its requests in Lines 4.13 – 4.14. If the primary is faulty and less than $b+1$ correct replicas conclude the commit phase and send a stable reply, at least $f + b$ replicas timeout and send a view change message to all other replicas. From Lines 2.15 – 2.16, all correct replicas also start a view change to remove the faulty primary. This is iterated until either the client completes the request in a view, or all replicas execute a view change to a view v' with a correct primary. From Lemma 15, the view change to v' is completed by all correct replicas. From Lemma 17, no correct replica initiate a further view change and all requests from correct clients are completed.

Scrooge also ensures the following additional liveness property to re-establish speculation after a failure event.

Theorem 3 Property 3 holds.

Proof: We consider the system behavior when the the system eventually enters a timely period. We first prove that SPEC-RUN eventually holds for each correct replica i. The primary proposes a replier quorum $RQ_p = Q$ along with an order request for sequence number n. If SPEC-RUN does not hold for a correct replica i, the replica send an agreement message for n in Lines 1.23 – 1.26. From Lemma 16, all other correct replicas do the same. If the primary never changes its replier quorum again, this implies that no client re-sends a request suspecting a replica in Q because all requests are completed from speculative replies, q.e.d. Let us thus assume by contradiction that n' is the lowest sequence number after n where the primary associate a replier quorum $RQ_p \neq Q$ to an ordered sequest. As the system is timely, all correct replicas commit on sequence number n and send speculative replies for all sequence numbers in (n, n'). Replicas do this in Lines 1.15 – 1.22 if they receive the order request for the sequence number after the

commit is reached, or in Line 4.40 otherwise. The primary updates its suspect list in Lines 1.10 – 1.12 and proposes a different replier quorum $RQ_p = S$ for n' only if the client of a request with sequence number in (n, n') has suspected some replica in Q as faulty and has indicated this while resending the request (see Lines 5.1 – 5.6). The client resends a request only if it has not received a speculative reply from at least one replica in Q, which is thus faulty. If $f = 1$, the new replier quorum S does not contain the faulty replica. S is eventually committed as, as explained previously, all replicas send speculative replies for all sequence number greater than n', a contradiction. If $f > 1$, the result follows by simple induction on the number of faults detected by clients.

B Integrating garbage collection

In this subsection we describe how garbage collection is integrated intro Scrooge. Readers who are familiar with PBFT [2] will notice that Scrooge uses the very same mechanisms.

B.1 Garbage collection

In BFT replication protocols, the checkpoint subprotocol is used to curb the size of the message history. With checkpoiting, replicas only store history elements with sequence numbers in the range $[l + 1, l + L]$ where l is called lower watermark, L is the size of the history log, and $l + L$ is called higher watermark.

Given a constant checkpoint interval K, a tentative checkpoint of the application state is built after requests with sequence number n such that $(n \bmod K) = 0$ are executed. The checkpoint subprotocol indicates that a tentative checkpoint can be retrieved by any correct replica because it has been established by enough (i.e. $b + 1$) correct replicas. When this happens, the checkpoint is considered as *stable*, all history elements prior to n are garbage-collected, and the lower watermark l is set to n. The protocol steps are the following:

Step GC.1: Replica executes the n^{th} request and n mod $K = 0$

Replica i initiates an agreement phase by executing the procedure *agree* for the request.

Step GC.2: Replica commits the n^{th} request and n mod $K = 0$

Replica i builds a tentative checkpoint composed by the application state and the replier quorum associated with sequence number n in the history, and sends a checkpoint message $\langle$CHECKPOINT, n, $i\rangle_{\mu_i}$ to all other replicas.

Step GC.3: Replica receives a checkpoint message

If a replica i receives $f + b + 1$ checkpoint messages corresponding to one of its tentative checkpoints for sequence number n, it considers it as *stable*, it sets the lower watermark l to n and garbage-collects the history elements and tentative checkpoints for sequence numbers $n' < n$.

B.2 Modifications to normal executions

Replicas simply need to check that their history log does not overflow.

Lines 1.6 – 1.13: Primary receives a request

The primary assigns sequence number n to the request only if n is not greater than its higher watermark $l + L$.

Lines 1.15 – 1.26: Primary receives an order request

A replica accepts an order request message or only if its sequence number $or.n$ is not greater than its higher watermark $l + L$.

B.3 Modifications to view change

With checkpointing it is not necessary to recover the entire history of previous requests but only a small subset of it. Replicas include information about their current checkpoint in their view change messages. An initial checkpoint for the new view is selected by the recovery function based on this additional information. The specific modifications are the following:

Lines 2.1 – 2.5: Initiating view change

The format of the view change message is modified to $\langle$VIEW-CHANGE, v', v, mh, C, E $i\rangle_{\sigma_i}$, where C is a set containing, for each checkpoint stored by the replica, a tuple $\langle n, d, RQ\rangle$ where n is the sequence number where the checkpoint was taken, d is the digest of the application state, and RQ is the repliers quorum associated to n. Also, the message history mh only includes the messages which are currently in the log and have not yet been garbage-collected.

Recover function: Recovering the observed history

The primary selects the checkpoint tuple $\langle n', d, RQ_{n'}\rangle$ with the highest sequence number n' which is contained in the view change messages of at least $b + 1$ replicas and such that the view change messages of at least $f + b + 1$ replicas report checkpoints for sequence numbers $n \leq n'$. This is called *initial checkpoint*. The history is then recovered as in the previous case but only for sequence numbers in the range $[n' + 1, n' + L]$, where L is the size of the history log. As in the previous case, the last established view mv is still identified using view establishment certificates, but the entire initial history ih_v is not recovered. Therefore, the instructions after Line 3.11 and until 3.15 are removed.

$RQ_{n'}$ is used to identify observed candidates for sequence number $n' + 1$.

Lines 2.24 – 2.28: Backup receives a new view message

Backup replicas also perform the same steps as the primary to recover the initial history for the new view.

B.4 Correctness

We prove that checkpointing preserves both safety and liveness. For safety we need to prove the following.

Lemma 18 If an initial checkpoint $\langle n', d, RQ_{n'} \rangle$ is selected and L is the size of the history log, then d and $RQ_{n'}$ are respectively the only digest of the application checkpoint and the only replier quorum associated with sequence number n' by any correct replica in any view.

Proof: A initial checkpoint is selected only if it has been sent by $b + 1$ replicas, including a correct one. This correct replica has thus completed the commit phase for sequence number n'. It follows from an argument similar to those of Lemmas 8 and 11 that the agreed history prefix for n' is recovered in any view by any correct replica.

Lemma 19 If an initial checkpoint $\langle n', d, RQ_{n'} \rangle$ is selected and L is the size of the history log, then no request with sequence number greater than $n' + L$ has completed.

Proof: Let us assume by contradiction that a request r is completed with sequence number n greater than $n' + L$. If r is completed in Lines 1.28 – 1.30 or 4.13 – 4.14, at least $N - f$ replicas have accepted an order request message with sequence number n. From Lines 1.15 – 1.26 it follows that n is not greater than their higher watermark. This implies that the lower watermark of these $N - f$ replicas is strictly greater than n' and, from Step GC.3, that their checkpoint for n' has been garbage-collected. At most f correct replicas and b Byzantine replicas can thus report a checkpoint for n' in their view change messages. This checkpoint can not be chosen as initial checkpoint by the recovery function as it is included in the view change messages from at most $f + b$ replicas.

Liveness is also ensured as follows.

Lemma 20 A correct replica can always recover one provably correct checkpoint.

Proof: Consider a period where the system is timely and let $c = \langle n, d, RQ_n \rangle$ be the stable checkpoint with the highest sequence number among those established by any correct replica at any given moment t. We prove that there are at least $b + 1$ correct replicas storing c as tentative or stable checkpoint c. This ensures that, by receiving $b + 1$ consistent checkpoints from these replicas, any other correct replica can prove that the checkpoint is correct. Assume by contradiction that at most b correct replicas store c. This implies that a set Q of at least $f + b$ correct replicas only store checkpoints for either smaller or larger sequence numbers than n. As a correct replica has set c as stable checkpoint, at least $f + b + 1$ replicas have once stored c as tentative checkpoint (Step GC.3). It is thus impossible that all the $f + b$ correct replicas in Q only store checkpoints for sequence numbers smaller than n. At least one of them, say j, must have only stored checkpoints for sequence numbers larger than n. This implies that the tentative checkpoint c has been garbage-collected by j because a higher stable checkpoint has been reached. Therefore, c is not the stable checkpoint with the highest sequence number among those established by any correct replica at time t, a contradiction