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

Marco Serafini, Péter Bokor, Dan Dobre, Matthias Majuntke and Neeraj Suri
Dept. of CS, TU Darmstadt, Germany
{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 [9]). Achieving fast Byzantine replication despite* f *failures requires at least* $f + b - 2$ *additional replicas [10, 6, 8]. In this paper we show, perhaps 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 of our approach is to have 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 [6].*

1 Introduction

Byzantine Fault Tolerant (BFT) state machine replication [11] 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, 5], to *fast* BFT protocols exhibiting the optimal number of agreement steps [10, 6, 8]. 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 [8].

Speculative BFT protocols [8] and other fast Byzantine agreement protocols such as FAB [10] and DGV [6] 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 [10, 6] 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 [9] 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 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 [5] do not offer better performance than Zyzzyva [8, 15].

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 [10, 6].

It is important to note that Scrooge does not contradict the resilience lower bounds of fast byzantine agreement [10, 6]. Scrooge attains fast Byzantine agreement only

*Research supported in part by EC Indexsys, Inco-Trust, Think-Trust and DFG GRK 1362 (TUD GKMM).

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 is *eventually* identified. Thus, fast agreement is eventually 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 [4], such as using specific network topologies, which are mostly orthogonal to our work and go beyond the scope of this and the other papers cited in Table 1. 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 $2f + b + 1$ [9]. 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 analytically and experimentally 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 increases more slowly with Scrooge than with PBFT, Zyzzyva or Zyzzyva5.

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

	Replication costs (min. $2f + b + 1$ [9])	Fast($\star$) with no unresponsive replica	Fast($\star$) with f unresponsive replicas
PBFT [2]	$3f + 1$	NO	NO
Zyzzyva [8]	$3f + 1$	YES	NO
Zyzzyva5 [8]	$5f + 1$	YES	YES
DGV [6]	$3f + 2b - 1$($\triangleright$)	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 [10, 6]. 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 finite time whereas Zyzzyva5 is always fast. ($\triangleright$) Cost to be fast with $f > 1$ unresponsive replicas. For $f = 1$ the corresponding cost is $2f + 2b + 1$ 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.

During reconfigurations, explicit agreement is performed by the replicas (Fig. 1.b). This is similar to PBFT, but 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 re-established 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, for example by bounding the number of accusations a client can send in a given unit of time. This reliance on clients 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

Scrooge 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. In current primary-based protocols such as [2, 8], replicas store the history of operations dictated by the primary but discard the authenticator of the messages from the primary after their authenticity has been verified. Scrooge lets replicas store the entire content of these messages, including the authenticator, in their *message histories*. This further reduces the replication cost from $4f + 1$ to $4f$ (again with $f = b$).

2 System and Fault Model

The system is composed of a finite set of clients and replicas. 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 has $N \geq 2f + 2b$ replicas. Any number of clients can be Byzantine. Clients and replicas are connected via an unreliable asynchronous network. The network has *timely* periods when all messages sent among correct nodes are delivered within a bounded delay.

We assume the availability of computationally secure 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 this is 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 ensuring that for any value m is impossible, given $H(m)$, to find a value $m' \neq m$ such that $H(m) = H(m')$.

3 The Scrooge Protocol

Scrooge replicates deterministic applications, 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. Beyond the classic safety and liveness properties necessary for BFT replication, in Scrooge clients eventually complete all their requests from speculative replies if the basic conditions for speculation are satisfied (i.e. the primary is fault-free, the clients are non-Byzantine and the system is timely) [14].

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

For easier understanding, we present 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 [12].

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. Replicas use only MACs for normal runs and reconfigurations.

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 pri-

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

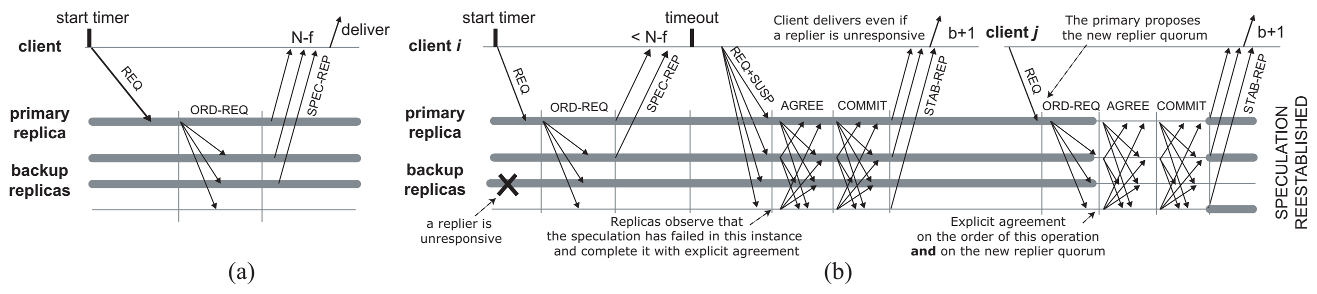
start timer
client
N-f
deliver
REQ
SPEC-REP
primary replica
ORD-REQ
backup replicas
(a)
start timer
timeout
Client delivers even if a replier is unresponsive
b+1
client i
client j
The primary proposes the new replier quorum
b+1
< N-f
REQ
SPEC-REP
REQ+SUSP
AGREE
COMMIT
STAB-REP
REQ
ORD-REQ
AGREE
COMMIT
STAB-REP
ORD-REQ
primary replica
backup replicas
SPECULATION REESTABLISHED
a replier is unresponsive
Replicas observe that the speculation has failed in this instance and complete it with explicit agreement
Explicit agreement on the order of this operation and on the new replier quorum
(b)

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      $t \leftarrow t+1;\ SL \leftarrow \perp$;
1.3      send $m = \langle$REQ, $o$, $t$, $c$, $SL\rangle_{\sigma_c}$ to the primary;
1.4      start timer;
1.5
1.6  upon primary $p(v)$ receives REQ message $m$ from client $m.c$ or a replica
1.7      if not IN-HISTORY($m$, $mh$) then
1.8          $n \leftarrow n+1;\ d \leftarrow H(m);\ RQ_p \leftarrow$ replicas $\notin SL$;
1.9          send ($\langle$ORD-REQ, $v$, $n$, $d$, $RQ_p\rangle_{\mu_p}$, $m$) 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 ORD-REQ message $orq$ from primary $p(v)$
1.16     if $i = p(v)$ or ($orq.v = v$ and $orq.n = n+1$ and
         $p(v) \in orq.RQ_p$) and not IN-HISTORY($orq.m$, $mh$) then
1.17         $n \leftarrow n+1;\ h[n] \leftarrow H(h[n-1], mh[n])$;
1.18         $mh[n] \leftarrow \langle orq.m, orq.RQ_p, orq.\mu_p\rangle$;
1.19         $r \leftarrow$ execute($orq.m.o$);
1.20         if SPEC-RUN($i$, $orq.m$, $orq.RQ_p$, $RQ$) then
1.21             if $i \in RQ$ then
1.22                 send $\langle$SPEC-REP, $v$, $n$, $h[n]$, $RQ$, $orq.m.c$,
                 $orq.m.t$, $r$, $i\rangle_{\mu_{p,c}}$ to client $orq.m.c$;
1.23         else
1.24             agree($orq.m$);
1.25             if $RQ \neq orq.RQ_p$ then $RQ \leftarrow \perp$;
1.26         if AGREEMENT-STARTED($i$, $n$, $v$) then agree($orq.m$);
1.27
1.28 upon client receives SPEC-REP message $sp$ from replica $sp.i$
1.29     if $|sp.RQ| = N-f$ and client received speculative replies matching
         $sp$ from all replicas in $sp.RQ$ then
1.30         deliver ($o$, $t$, $sp.r$);   stop timer ;
1.31
```

mary 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, 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 matching speculative replies from all the $N-f$ replicas in 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 eliminating faulty repliers to re-establish the communication pattern of Fig. 1.a. Replicas start a full three-phase agreement similar to PBFT by calling the *agree* procedure, which takes the client request as argument (see [12] for the full pseudocode). An example of reconfiguration over two client requests is in Fig. 1.b.

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, 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 with the predicate COMMITTED if a three-phase agreement on the order of the request has already been completed. If not, the primary adds the suspect list provided by the client to its list of the f most-recently suspected servers SL and starts agreement (Lines 1.10 – 1.12). The backups 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 – 1.25). 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. Similar to 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 send *stable reply* messages to the client. Stable replies differ from speculative replies as 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).

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. The primary computes a new replier quorum RQ_p from the suspect list SL in Line 1.8. It then 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 on RQ_p 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 that 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.

4 Scrooge View Change

If backups receive requests from the clients and see that the system is not able to commit them, they start a view change to replace the current primary. 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, replicas now use signed messages.

Algorithm 2: Scrooge - View change

```
2.1  procedure view-change($nv$)
2.2      stop executing request processing;
2.3      $v' \leftarrow nv$;
2.4      send $\langle$VIEW-CHANGE, $v'$, $v$, $mh$, $aw$, $E$, $i\rangle_{\sigma_i}$ 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          $k \leftarrow n' + 1 : \forall ev \in vc.E, cv.n = n'$;
2.10         while $mh[k] \neq \perp$ do
2.11             $res[k] \leftarrow$ verify($vc.v$, $k$, $vc.mh[k]$);
2.12             $k \leftarrow k + 1$;
2.13         $d \leftarrow H(vc)$; $j \leftarrow vc.i$; $v_j \leftarrow vc.v$;
2.14         send $\langle$CHECK, $j$, $v_j$, $d$, $res$, $i\rangle_{\sigma_i}$ 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 $\langle$NEW-VIEW, $v'$, $VC$, $CH$, $i\rangle_{\mu_i}$ 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 $\langle$NEW-VIEW, $v'$, $VC$, $CH$, $i\rangle_{\mu_i}$ 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         $h \leftarrow H(ih)$;
2.27         $n \leftarrow$ length($ih$);
2.28         send $\langle$EST-VIEW, $v'$, $n$, $h$, $i\rangle_{\sigma_i}$ to all replicas;
2.29
2.30 upon replica $i$ receives an EST-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         $mh \leftarrow ih$; $v \leftarrow v'$; $E \leftarrow E_{v'}$;
2.33         $aw, cw \leftarrow \max\{k : mh[k] \neq \perp\}$; $RQ \leftarrow mh[cw].RQ$;
2.34         start executing request processing;
2.35
```

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(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 the current 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.

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(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 sufficient 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(c)). If these messages claim that the message history is authentic we call the history *verified*. 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 replica and thus become verified.

The primary of a new view v' calls the *recover* function

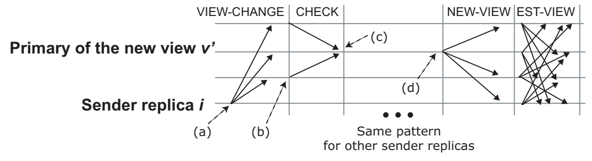

Figure 2: Scrooge view change subprotocol.

(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'. 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 in ih. We now continue illustrating the communication pattern and argue about the correctness of the recover function in the next subsection.

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(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 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, Table 3 lists the predicates used in the pseudocode.

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 is stable if each element in the corresponding message history $vc.mh$ is consistently verified by at least $b + 1$ check messages received by the new primary (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

Algorithm 3: Scrooge - View change procedures

```
3.1  **function** verify($v$, $n$, $e$)
3.2      $d \leftarrow H(e.m)$; $or \leftarrow$ (ORD-REQ, $v$, $n$, $d$, $e.RQ$);
3.3      $\mu \leftarrow$ calculate-MAC($or$, $p$);
3.4      **if** $\mu = e.\mu_p[i]$ **then** **return** true;
3.5      **else return** false;
3.6
3.7  **function** recover($VC$, $CH$)
3.8      $recovered \leftarrow false$;
3.9      $VC_s \leftarrow VC \setminus \{vc \in VC : \neg\text{STABLE}(vc, CH) \vee vc.v' \neq v'\}$;
3.10     **if** $|VC_s| \geq N - f$ **then**
3.11         $mv \leftarrow \max\{\overline{v} : \exists vc \in VC_s \text{ with } vc.v = \overline{v}\}$;
3.12         $vc_{mv} \leftarrow vc \in VC_s$ with $vc.v = mv$;
3.13         $n_{mv} \leftarrow \overline{n} : \forall ev \in vc.E, ev.n_v = \overline{n}$;
3.14         $ih \leftarrow \{vc_{mv}.mh[k] : (k \leq n_{mv})\}$;
3.15         $RQ_{mv} \leftarrow vc_{mv}.mh[n_{mv}].RQ$;
3.16         $k \leftarrow n_{mv} + 1$; $loop \leftarrow true$; $recovered \leftarrow true$;
3.17         **while** $loop$ **do**
3.18             $A \leftarrow \{e :$ AGREED-CAND$(e, k, mv, VC_s, ih)\}$;
3.19             $O \leftarrow \{e :$ 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                 $loop, recovered \leftarrow false$;
3.22             **else**
3.23                 **if** $\exists e \in A$ **then**
3.24                     $ih[k] \leftarrow e$;
3.25                 **else if** $\exists e \in O$ : VERIFIED$(e, VC_s, CH)$ **then**
3.26                     $ih[k] \leftarrow e$;
3.27                 **else if** $\exists e \in O$ **then**
3.28                     $ih[k] \leftarrow e$;
3.29                 **else** $loop \leftarrow false$;
3.30                 $RQ_k \leftarrow ih[k].RQ$;
3.31             $k \leftarrow k + 1$;
3.32     **return** $recovered$;
3.33
3.34 **function** recover-prim()
3.35     $VC \leftarrow$ set of received view change messages for view $v'$;
3.36     $CH \leftarrow$ set of received check messages for view $v'$;
3.37     **return** recover($VC$, $CH$);
3.38
```

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

NEXT(mh, k) $\triangleq (\forall k' < k, mh[k'] \neq \bot) \wedge mh[k] = \bot$

SPEC-RUN(i, m, RQ_p, RQ) $\triangleq RQ \neq \bot \wedge RQ_p = RQ \wedge$
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) $\triangleq \exists bool : \forall k : vc.mh[k] \neq \bot,$
$\exists (b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i \wedge$
$ch.d =$ 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) $\triangleq \exists k,\ vc \in VC : e = vc.mh[k] \wedge$
$(\exists(b+1)\ ch \in CH : ch.v_j = vc.v \wedge ch.j = vc.i$
$\wedge\ ch.d =$ digest(vc) $\wedge\ ch.res[k] = true)$

Table 3: Predicates needed for view change

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 (Lines 3.23 – 3.24). Committed histories are recovered like in PBFT (see predicates AGREED-CAND and WAIT-AGR). Therefore, our discussion will focus on recovering histories completed by clients through speculative replies.

Why are replier quorums useful? If a reply is delivered by clients in a *fast* manner, i.e., out of speculative replies (Lines 1.28 – 1.30), then 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.

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 ordered 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 it is thus 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 ordered 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.

Validity Unlike other protocols, Scrooge allows a request to be included into ih even if it is only reported by b Byzantine replicas. As client requests are signed, no request in ih is fabricated on behalf of correct clients, as commonly required for Validity by BFT replication protocols, e.g. [7].

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. We also show that Scrooge 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 calculating the history digests included in the speculative replies. Second, speculative replies must include a bitmap representing the current replier quorum. Our evaluation shows that these overheads are negligible.

We refer to [15] for a comparison between quorum- and primary-based algorithms. As a reference, however, we scale the performance figures of Q/U [1] to our setting.

Optimizations Scrooge uses optimizations similar to PBFT and Zyzzyva to improve the performance of the protocol. The main difference between Zyzzyva and Scrooge is the read-only optimization. This lets clients send read-only requests directly to the replicas, which immediately 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, 8]. 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. Also, the Zyzzyva library uses a *commit optimization* to avoid excessive performance degradation with unresponsive replicas. If clients cannot receive speculative replies from all replicas, the protocol stops using speculation for successive

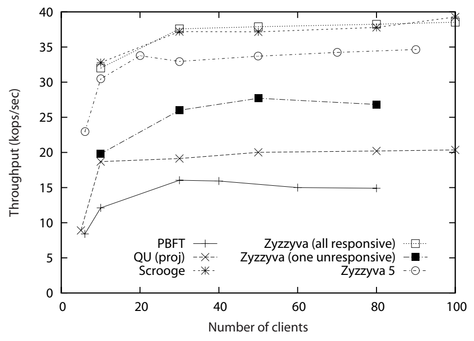

Figure 3: Throughput for 0/0 microbenchmark without batching and with $f = 1$.

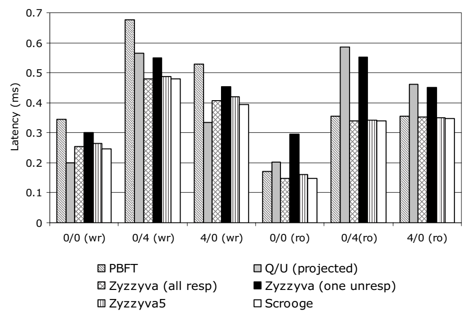

Figure 4: Latency for different benchmarks with a single client and no batching.

requests and use one all-to-all agreement round instead [8].

Batching improves the performance of BFT algorithms under high load by letting replicas execute the protocol on groups of client requests [2]. Using batching similarly impacts all evaluated algorithms, making it more difficult to compare their performance under high load [15].

PBFT, Zyzzyva and Zyzzyva5 use MACs for client requests but this makes them vulnerable to client attacks [4]. Scrooge tolerates such attacks by using signed client requests. For fairness and consistency with previously published results, our comparison lets all algorithms use MACs.

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, and are connected through a Gigabit switched star network. All servers are single-threaded processes. Nodes run Fedora Linux 8 with kernel version 2.6.23. We use MD5 to compute MACs and the AdHash library for incremental hashes as in [2, 8]. For performance stability, measurements are initiated after the execution of the first 10,000 operations, and are stopped after the successive 10,000 operations. 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 replies respectively. We consider scenarios where all replicas are responsive and where one replica is initially crashed.

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 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, Zyzzyva achieves 52 kops/s peak throughput in presence of read-only workloads, whereas Scrooge achieves a peak of 85 kops/s.

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 (~ 0.9 kops/sec) in runs with unresponsive replicas and 10 clients.

Fault scalability A fault scalable replication protocol keeps costs low when the number of replicas, and thus

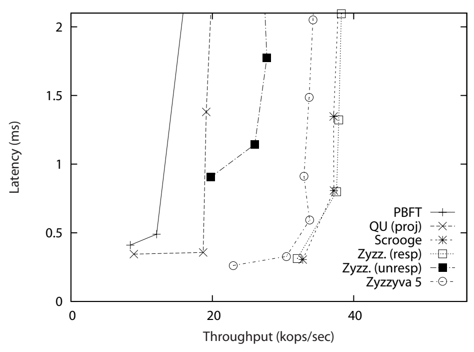

Figure 5: Latency-throughput curves for 0/0 microbenchmark without batching and with $f = 1$.

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 $(2 + 8f/s)$, Zyzzyva5 $(2 + 5f/s)$ and Zyzzyva with one unresponsive replica $(2 + 5f/s)$. In Q/U the bottleneck replica makes only 2 MACs operations per request.

Scrooge uses $1 + 3f + (4f - 1)/s$ messages per request, similar to Zyzzyva in fault-free runs $(2+3f+3f/s)$, Zyzzyva5 $(2 + 4f + 5f/s)$ and, with $s = 1$, Q/U $(2 + 8f)$. With unresponsive replicas, PBFT and Zyzzyva with commit optimization have quadratic complexity. Without commit optimization, Zyzzyva has lower message complexity but also significantly lower performance [8].

6 Related Work

We have provided a comparison of Scrooge with PBFT [2], Zyzzyva [8] and DGV [6] throughout this paper.

In [7] a framework is proposed where different BFT protocols can be combined to react to different systems conditions and requirements. Two protocols for fault-free runs, one optimized for latency in runs with no concurrency and another with high throughput but high latency, are introduced. In presence of unresponsive replicas, these protocols switch to a backup protocol such as PBFT.

Protocols like Q/U [1] and HQ [5] 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 number of calculated MACs [8, 15].

Preferred quorums is an optimization used by clients in some quorum-based BFT replication protocol, mainly to reduce cryptographic overhead [5, 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.

The redundancy costs of BFT replication in asynchronous networks with eventual timely periods can also be reduced by using trusted components as shown in [13, 3].

7 Conclusions

BFT state machine replication requires making a tradeoff between optimal performance and replication costs. Scrooge mitigates this tradeoff through two novel techniques: replier quorums and message histories. 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. In systems where tolerating any number of crashes but only one Byzantine failure is sufficient, Scrooge is the best choice as it is always fast and uses a minimal number of replicas.

Acknowledgements The authors are grateful to R. Kotla, L. Alvisi and M. Dahlin for sharing Zyzzyva, to M. Castro, R. Rodriguez, J. Cowling and B. Liskov for sharing PBFT, and to F. Junqueira for his comments on a draft of this paper.

References

[1] M. Abdel-El-Malek *et al.*, "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 *et al.*, "Attested append-only memory: making adversaries stick to their word," *SOSP*, pp. 189–204, 2007.

[4] A. Clement *et al.*, "Making Byzantine Fault Tolerant System Tolerate Byzantine Faults," *NSDI*, pp. 153–168, 2008.

[5] J. Cowling *et al.*, "HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance," *OSDI*, pp. 177–190, 2006.

[6] P. Dutta *et al.*, "Best-case complexity of asynchronous Byzantine consensus," *EPFL T.R. 200499*, 2004.

[7] R. Guerraoui *et al.*, "The next 700 BFT protocols," *Eurosys*, 2010.

[8] R. Kotla *et al.*, "Zyzzyva: Speculative Byzantine Fault Tolerance," *SOSP*, pp. 45–58, 2007.

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

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

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

[12] M. Serafini *et al.*, "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*

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

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

[15] A. Singh *et al.*, "BFT Protocols Under Fire," *NSDI*, pp. 189–204, 2008.