HP: Hybrid Paxos for WANs*

Technical Report TR-TUD-DEEDS-04-03-2009

Dan Dobre, Matthias Majuntke, Marco Serafini and Neeraj Suri
{dan,majuntke,marco,suri}@cs.tu-darmstadt.de
Phone: +496151165657, Hochschulstr. 10, 64289 Darmstadt, Germany

February 18, 2010

Abstract

Implementing a fault-tolerant state machine boils down to reaching consensus on a sequence of commands. In wide area networks (WANs), where network delays are typically large and unpredictable, choosing the best consensus protocol is difficult. During normal operation, Classic Paxos (CP) requires three message delays, whereas Fast Paxos (FP) requires only two. However, when collisions occur, due to interfering commands issued concurrently, FP requires four extra message delays. In addition, FP uses larger quorums than CP. Therefore, CP can outperform FP in many situations.

We present Hybrid Paxos (HP), a consensus protocol that combines the features of FP and CP. HP implements generalized consensus, where collisions are caused only by interfering commands. In the absence of collisions HP requires two message delays, and only one extra message delay otherwise. Our evaluation shows that when collisions are rare, the latency of HP reaches the theoretical minimum. When collisions are frequent, HP behaves like CP.

Keywords: Consensus, WAN Replication, Distributed Systems, Fault-tolerance, Quorum Systems

*Research funded in part by EC Inspire, IBM Faculty Award, Microsoft Research, Think-Trust and DFG GRK 1362 (TUD GKmM).

1 Introduction

WAN replication offers protection against catastrophic failures of a single site and can be used to enhance the resilience of critical services. Implementing a deterministic service in a fault-tolerant manner boils down to reaching consensus on a sequence of system commands.

In the standard state-machine approach, a sequence of instances of a consensus protocol are used to choose the sequence of client commands, where the ith instance chooses the ith command. In this paper we consider generalized consensus [17], where a single instance of consensus is used to choose an increasing history of commands. A history is an equivalence class of command sequences, where two command sequences are equivalent iff executing them produces the same state and output. The underlying observation is that often commands *commute*, so it does not matter in which order they are executed.

The consensus problem is stated in terms of *proposers* that propose commands, *acceptors* that choose an increasing command history and *learners* that learn what history has been currently chosen. In a client/server system, clients might play the roles of proposer and learner and servers might play the role(s) of acceptor (and learner). A leader is elected among the acceptors to coordinate their actions.

In a WAN environment, where network delays are large and unpredictable, the *latency* of a consensus protocol matters. Latency is defined as the number of message delays between when a client proposes a command and when that command is learned by a learner.

Consensus protocols which attain the optimal latency [19] are the well known Classic Paxos (CP) [16] and the more recent Fast Paxos (FP) [17]. Their message patterns are illustrated in Figure 1. In normal operation, CP requires three message delays. The communication pattern during normal operation is Client → Leader → Acceptors → Learners. FP saves one message delay by having the clients send their proposals directly to the acceptors, bypassing the leader. This works fine if the acceptors receive the same sequence of interfering commands. However, when commands are proposed concurrently, commands may be accepted in interfering orders, resulting in no command being chosen. In order to guarantee progress, FP then runs a collision recovery procedure, which adds four message delays. Thus, if collisions are frequent, FP has a significantly higher latency and a lower throughput than CP.

We found that even in the absence of collisions, depending on the the layout of clients and servers, CP can outperform FP (for many clients). This comes from the fact that in order to be fast, FP needs larger quorums than CP, called *fast quorums* [19].

When clients have direct access to a local replica, the recently developed consensus protocol Mencius [20] has been shown to outperform CP. However often, clients and servers are not co-located. When clients are using a remote service replicated for disaster tolerance, none of the mentioned protocols has the final say.

Paper Contributions In this paper we present a generalized consensus protocol called Hybrid Paxos (HP). HP is essentially CP with an additional "fast mode" that enables fast learning in the absence of collisions. In presence of collisions, HP requires three message delays as CP does. These latencies are optimal [19] and they are attained using a linear number of messages and the optimal number of $2f+1$ servers, where f is the bound on crash-failures. Compared to Mencius, HP uses weaker synchrony assumptions, resulting in higher availability in WANs.

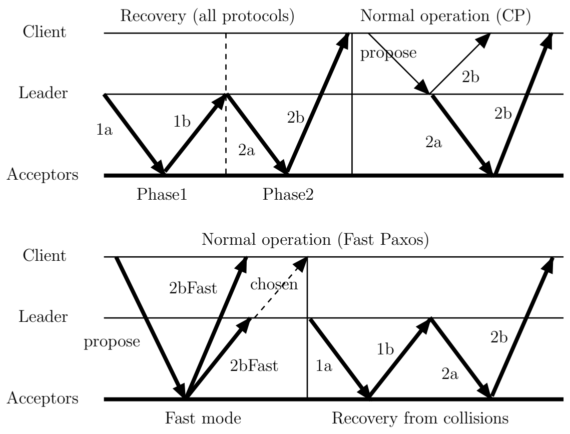


Figure 1: Paxos message patterns

We show for the first time that generalized consensus can be used in practice to build efficient replicated services. The key to efficiency is that fast learning must not impact the bottleneck, which in CP is the leader. Additional messages in HP are exchanged only between clients (which are both proposers and learners) and acceptors. Thus, HP exploits the relative underutilization of the acceptors and offers a better latency up to 70% of the peak throughput of CP.

In addition, fast learning is enabled only if spare capacity is available. This is done by adaptively switching it on and off based on the load. Our evaluation using Emulab [25] shows that the latency of HP reaches the theoretical minimum. In the presence of collisions and with increasing load, HP behaves like CP.

1.1 Motivation

There is no clear winner with Fast and Classic Paxos We argue that the quorum size matters by showing that even in the absence of collisions, CP can outperform FP.

We have sampled delays among Planetlab [3] nodes and used them to simulate the normal operation (best case) of CP and FP in four different WAN settings (Table 1). The client distribution is as follows: 56% are located in the US, 38% in Europe, and 6% in Asia. All topologies use 11 servers. FP requires a fast quorum (9 servers), while CP only requires a (majority) quorum. The simulation results in Figure 2 suggest that: (a) in many settings, some clients are better off using FP and others prefer CP and (b) the crash of a single server can turn a setting beneficial for FP into one beneficial only for CP. Thus, there exist practical settings where neither of the two protocols always outperforms the other.

Table 1: WAN server layout (11 servers)

Topology	**Europe**	**World**	**CLUS-5(4)**
Leader Site	Hungary	Japan	Switzerland
Backups	Europe	Global	Europe
Clusters	No	No	1 with 5 (4) nodes, 3 with 2 nodes
Quorums	6 servers for CP, 9 servers for FP		

In the *Europe* setting, servers are located at 11 different sites in Europe. For most clients, the distance between them and the servers is close to uniform. Thus, the FP pattern leads to good results: 28% of the clients observe that FP is at least 10% faster than CP, and 10% of the clients even observe a 20% improvement. However, 12% of the clients find that CP is 10% better, as they do not have good connectivity with three additional servers required by FP. This supports observation (a).

The *World* setting models a world-wide setting in which acceptors are spread over the US, Europe and Asia, and the leader is located in Asia. In this scenario the advantage of the FP pattern is even clearer: 75% of the clients observe a 10% improvement over CP, and for 20% of the clients the improvement is $> 35\%$. The reason is the additional message delay to reach the leader, which is large for most of the clients. However, there are some clients which prefer CP: 6% find it to be 20% more efficient than FP, supporting observation (a). This is essentially the fraction of Asian clients which can quickly reach the leader.

The *CLUS* topology considers the case when servers are clustered at four different sites in Europe, providing cheap distaster tolerance. The only distinction between the *CLUS-5* and *CLUS-4* is that in the latter, one node in the largest cluster is crashed (Table 1). Before the crash, a fast quorum (9 servers) can be reached by contacting three sites. Note that for 13% of the clients, FP outperforms CP by at least 10%. Two sites are sufficient for CP to sample a quorum (6 servers). However, after the crash FP accesses *all* sites. This results in a shift of the performance profile, with CP dominating FP for 50% of the clients, supporting observation (b).

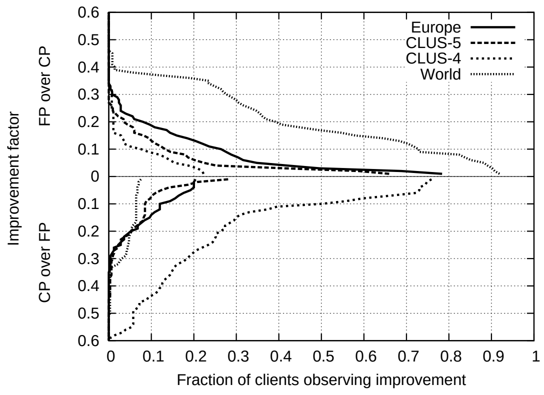


Figure 2: Improvement factor of FP over CP and vice versa

2 System Model and Definitions

We consider a distributed system consisting of n servers and any number of clients. For simplicity, we assume reliable FIFO channels. Further we consider a crash-stop model in which clients and servers fail only by crashing and nonfaulty servers never crash. [1] We assume that at most a minority of servers fails, which is necessary to solve consensus [7]. The system is *asynchronous*, with no bounds on message delay or processor speed. However, each server has access to a failure detector Ω, that eventually outputs at all servers the same nonfaulty server [6].

Mathematical Preliminaries In the generalized consensus problem [17] a single consensus instance

[1]The algorithm can be easily extended to a model in which crashed nodes may recover [1] and links are fair-lossy [2]. This however, lies outside the scope of the paper.

is performed on a monotonically increasing and partially ordered set of commands, called *command history*. A command history, or history for brevity, is defined as an equivalence class of command sequences. Two command sequences are equivalent iff they can be transformed into one another by permuting its elements such that the order of every pair of interfering commands is preserved. Two commands are *interfering* if it matters in which order they are executed. Else, they are called *commutable*.

Histories are constructed by appending a command sequence σ to the initially empty history $\perp$ using the special append operator $\bullet$. The resulting history is $\perp \bullet \sigma$. Histories $\perp \bullet \sigma$ and $\perp \bullet \tau$ are equal iff σ and τ are equivalent sequences.

The prefix relation $\sqsubseteq$ is defined as a partial order on the set of histories. For two histories h and h', $h \sqsubseteq h'$ iff there is a command sequence σ such that $h \bullet \sigma = h'$. We say that h is a prefix of h' (or equivalently that h' is an extension of h). A history h is isomorphic to a directed graph $G(h)$ whose nodes are the commands. There is an edge between any two interfering commands c_i and c_j from c_i to c_j in $G(h)$ iff $i < j$ in h. For two histories h and h', it holds that $h \sqsubseteq h'$ iff the graph $G(h)$ is a prefix of the graph $G(h')$. $G(h) = G(h')$ iff $h = h'$.

A lower bound of a set H of histories is a history that is a prefix of every element in H. The greatest lower bound (*glb*) of H is a lower bound of H that is an extension of every lower bound of H. We write the *glb* of H as $\bigsqcap H$ and we let $h \sqcap h'$ equal $\sqcap\{h, h'\}$ for any two histories h and h'. The least upper bound (*lub*) is defined in the analogous manner. We write *lub* of H as $\bigsqcup H$ and and we let $h \sqcup h'$ equal $\sqcup\{h, h'\}$. Intuitively, the *glb* (resp. *lub*) of a set of histories is the largest common prefix (resp. the smallest common extension).

We define two histories h and h' to be *compatible* iff they have a common upper bound,i.e., there is some history g with $h \sqsubseteq g$ and $h' \sqsubseteq g$. A set of histories H is compatible iff every pair of histories in H are compatible.

Consensus Properties Generalized consensus ensures that if any two command histories are learned, then they are compatible (*Consistency*). To rule out choosing a default value, it must hold that if history h is chosen then there exists a proposed command sequence σ, such that $h = \perp \bullet \sigma$ (*Nontriviality*). Finally, if a learner process learns history h, then h was chosen (*Conservatism*). Liveness requires that if command c is proposed, then eventually some history containing c is learned (*Progress*).

3 Generalized Consensus and Paxos

In this section we review Paxos and describe it as consensus on a growing command history [17]. When needed, we differentiate between FP and CP. This description serves as a basis for HP which is introduced in Section 4.

In the client/server system that we consider, clients are both proposers and learners. The servers are acceptors and cooperate to choose a single command history. Acceptors query the Ω failure detector that elects a leader among them. Safety is guaranteed even if no leader or multiple leaders are elected, but a unique leader is required to ensure progress. Paxos operates on a set of round numbers. The round numbers are partitioned among the potential leaders such that each leader has its disjoint set of round numbers.

As mentioned in the introduction, Paxos assumes predefined sets of acceptors called *quorums*. The requirement for CP is that any two quorums intersect. FP requires larger quorums, called *fast quorums*, and the requirement is that the intersection of any fast quorum FQ and any quorum is larger than $n - |FQ|$.

Following the Paxos protocol description [16], we divide the acceptor and leader actions in Phase 1 and Phase 2 actions. Phase 1 actions are executed when a new round is started (e.g. after a leader crash). Phase 2 actions (1) complete the choosing all the histories that failed to be chosen in an earlier round and (2) they are repeatedly executed during normal operation.

We now describe the algorithms' actions below (see also Fig. 1 as an illustration). Note that the focus lies on consensus, and therefore the execution of commands is omitted from the description.

Phase1: Start a new round

(1a) Leader l picks a new round number r from its set of round numbers and sends a $\langle$"$1a$"$, r\rangle$ message to all acceptors.

(1b) When acceptor a receives a $\langle$"$1a$"$, r\rangle$ message from leader l, if it has not yet received a message with a higher round number, then it replies with a $\langle$"$1b$"$, r, \ldots\rangle$[2] message. We say that a has moved to round r and considers r as its current round from now on. Moreover, a stops accepting proposals from clients.

If a has already received a message with round number $r' > r$ then it sends a message to the leader, indicating that it is ignoring the $\langle$"$1a$"$, r\rangle$ message. (Upon receiving such a message the leader performs step (1a) with a round number $> r'$ if it still believes to be the leader.)

Phase2: Complete earlier rounds

(2a) If leader l has received $\langle$"$1b$"$, r, \ldots\rangle$ messages from a quorum of acceptors, then it sends a $\langle$"$2a$"$, r, h\rangle$ message to the acceptors where h is the history that has been determined from the received "$1b$" messages. Further, the leader adopts h as the history currently chosen. The rule of picking h depends on the type of protocol and is described later (see Sections 3.1 and 4.2.1).

(2b) If acceptor a receives a $\langle$"$2a$"$, r, h\rangle$ message in its current round r (i.e. it has not yet received any message with a higher round number), it stores h as the accepted history and sends a $\langle$"$2b$"$, r, h\rangle$ message to every learner. Next, a starts accepting proposals from clients.

(Learn) If a learner receives identical $\langle$"$2b$"$, r, h \bullet c\rangle$ messages from a quorum, then it learns that history $h \bullet c$ is chosen.

[2]Note that "$\ldots$" will be replaced by protocol specific information later.

Normal operation CP

(Propose c**)** Client cl sends a $\langle$"*propose*"$, c\rangle$ message to the leader.

(2aCP) When the leader receives a $\langle$"*propose*"$, c\rangle$ message from client cl, it appends command c to h and sends a $\langle$"$2a$"$, r, h \bullet c\rangle$ message to the acceptors.

(2bCP) If acceptor a receives a $\langle$"$2a$"$, r, h \bullet c\rangle$ message from the leader in its current round r (i.e. it has not yet received any message with a higher round number), then it accepts $h \bullet c$ and sends a $\langle$"$2b$"$, r, h \bullet c\rangle$ message to all learners. Learning is done as described in the (Learn) step.

Normal operation FP

(ProposeFP c**)** Client cl sends $\langle$"*propose*"$, c\rangle$ messages to the acceptors.

(2bFast) If acceptor a receives a $\langle$"*propose*"$, c\rangle$ message from client cl, then a appends c to its command history h and sends $\langle$"$2bFast$"$, r, h \bullet c\rangle$ messages to the learners and to the leader.

(Collision Handling) If the leader receives identical $\langle$"$2bFast$"$, r, h \bullet c\rangle$ messages from a fast quorum, it indicates to the learners that $h \bullet c$ is chosen by sending $\langle$"*chosen*"$, r, h \bullet c\rangle$ messages to the learners. Else, the leader initiates collision recovery, which entails starting a new round (Phase 1) and recovering from earlier rounds (Phase 2).

(Fast Learn) If a learner receives identical $\langle$"$2bFast$"$, r, h \bullet c\rangle$ messages from a fast quorum, or equivalently a $\langle$"*chosen*"$, r, h \bullet c\rangle$ message then it learns that $h \bullet c$ is chosen. "Slow" learning is done as in (Learn).

3.1 The rule of picking a history

We now explain the core of the Paxos protocol and why it satisfies Consistency. For this purpose, we now informally describe the rule of picking a history based on the $\langle$"$1b$"$, r, \ldots\rangle$ messages received by the leader in step (2a). A formal and complete treatment appears in an earlier work by Lamport [17].

Invariant Paxos maintains the following invariant for safety: if a history h is chosen in round r and a history h' is chosen in a higher numbered round, then $h \sqsubseteq h'$. Intuitively, Consistency follows from this invariant and the fact that once a quorum of acceptors has joined a higher numbered round, no history can be chosen in previous rounds anymore.

Pick classic In CP, learning implies that if a history h has been chosen, then a quorum of acceptors has accepted h. In step (1b), each acceptor reports the history it has accepted. By the quorum intersection property, at least one acceptor reports h. The picking rule is to select the *lub* among the reported histories. Is is not difficult to see that $h \sqsubseteq lub$.

Pick fast In FP, fast learning implies that if a history h has been chosen, then a fast quorum FQ has accepted h. In step (1b), each acceptor reports the history it has accepted. Let Q be the set of all reported histories collected by the leader in step (2a). By the intersection property of FQ with a quorum, Q contains at least $n - |FQ| + 1$ (possibly incompatible) extensions of h and at most $n - |FQ|$ histories which are not extensions of h. Hence, there is a majority subset $M \subseteq Q$ containing the extensions of h. The goal is now to find a history which is an extension of h using this knowledge. First, the *glb* is computed for every majority subset $M \subseteq Q$. As all majorities intersect, the *glb*s are pairwise compatible. Next, the leader picks the *lub* of these *glb*s. Note that one of the *glb*s is an extension of h, and therefore $h \sqsubseteq lub$.

4 The HP Protocol

As mentioned in the introduction, HP is essentially CP with an integrated "fast mode", which allows fast learning in the absence of collisions. Therefore, the progress property of HP is inherited from CP. Hence, throughout the section, we will focus on safety.

The roles played by clients and servers and their interaction are the same as in Paxos (see Section 3). Phase 1 and 2 do not change and so they are as depicted in Figure 1. Figure 3 illustrates the message pattern of HP during normal operation.

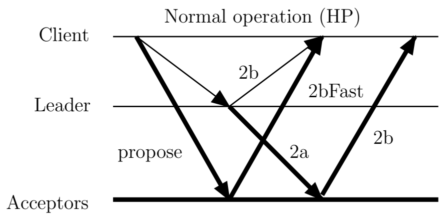


Figure 3: HP message pattern

4.1 Overview

We now briefly summarize the main differences between HP and Paxos.

First, fast learning is refined to accommodate that learning and fast learning are done in parallel, such that a learner can learn the quickest outcome. A naïve composition would fail, as two incompatible histories could be learned in the (Fast Learn) and (Learn) steps. We prevent this problem by replacing fast learning with *hybrid learning*. The idea of hybrid learning is that a learner waits for a fast quorum of identical "$2bFast$" and "$2b$" messages, of which at least one is of type "$2b$". Note that hybrid learning is fast because the leader is an acceptor (see Figure 3).

Secondly, the rule of picking a history in step (2a) is extended accordingly. In HP, each acceptor keeps two separate histories, a *classic* history updated by "$2a$" messages and a *fast* history updated by client proposals. Both histories are reported to the leader in step (1b). The leader applies the picking rules as described in Section 3.1 to each type separately. Resulting are two histories h and fh, where fh is determined by the fast histories. The final history is determined as the *lub* of h and the largest prefix of fh which is compatible with h.

4.2 The Protocol

We now describe the actions of the HP protocol during normal operation. The focus lies on highlighting the difference to CP. A complete description in pseudocode and proofs can be found in the Appendix.

Phase 1 and 2 actions as well as actions (2aCP), (2bCP) and (Learn) are actions of the HP protocol. Since they do not change, they are not listed below.

Normal Operation

(ProposeHP c) Client cl executes the actions (Propose c) and (ProposeFP c).

(2bFastHP) If acceptor a receives a $\langle$"*propose*"$, c\rangle$ message from client cl, then a appends c to the local fast history fh and sends $\langle$"$2bFast$"$, r, fh \bullet c\rangle$ messages to the learners. (Note that the difference to action (2bFast) is that c is appended to the fast history, and that no "$2bFast$" messages are sent to the leader).

(Hybrid Learn) If a learner receives identical $\langle$"$2bFast$"$, r, fh \bullet c\rangle$ messages from a fast quorum *and* one $\langle$"$2b$"$, r, h \bullet c\rangle$ message *and* $h = fh$ then it learns that $h \bullet c$ is chosen. "Slow" learning is done as in (Learn).

4.2.1 The rule of picking a history

We now describe the rule of picking a history in HP just as we did for Paxos. We will widely reuse the steps in Section 3.1 and refer to them when needed.

In HP, the $\langle$"$1b$"$, r, \ldots\rangle$ messages report two separate, accepted histories, the (classic) history and the fast history. The leader uses the reported (classic) histories to pick a history h as described in **Pick classic**. Next, the leader uses the reported fast histories to pick a history fh as described in **Pick fast**. Note that each of the histories satisfies the invariant. History h is an extension of any history learned in the (Learn) step and fh is an extension of any fast history learned in the (Hybrid Learn) step.

Pick hybrid We now explain the key difference between HP and Paxos. To be safe, ideally we would pick the *lub* of h and fh and initialize the acceptors with *lub* in phase (2a). However, if h and fh are incompatible, then their *lub* is undefined. Therefore, the idea is (1) to determine the largest prefix pfh of fh which is compatible with h and (2) to pick the *lub* of pfh and h. This would be safe only if we can guarantee that any history lh learned in step (Hybrid Learn) is a prefix of pfh.

We will now argue that this is the case. We know that lh is a prefix of fh. By the choice of pfh, all prefixes of fh which are compatible with h are also prefixes of pfh. Thus, it suffices to show that lh is compatible with h. Hybrid learning implies that some acceptor has accepted lh as (classic) history. Thus, lh is a prefix of some history sent by the leader in a "$2a$" message. Clearly, this holds for h too. Any two histories sent by the leader (of the same round) are prefixes of each other. So, if *max* is the largest of the two histories, then *max* is a common extension of lh and h. Thus, lh and h are compatible.

4.2.2 Implementation Considerations

Now that we have argued about the correctness (safety) of HP, in this section we describe how HP can be tweaked to be practical. We have identified a set of optimizations and listed them below.

O1 The leader does not have to send the entire history $h \bullet c$ in step (2aCP), it suffices to send c. When an acceptor receives c in a "$2a$" message, the FIFO property implies that it has already received h.

If the state machine is implemented at the servers, then there is no reason to send the entire history to the clients. All a client needs to learn is **(a)** the execution result of its last issued command and **(b)** that the history producing the result is chosen.

O2 The solution to **(a)** is to have the servers speculatively execute commands. Specifically, when the leader receives a proposal from a client, it immediately executes the command and includes the result in the "$2b$" message it sends back to the client. Speculation at the leader avoids rollbacks and history replays during normal operation.

O3 In order to attain **(b)** without sending the history, we replace the histories in the "$2b$" and "$2bFast$" messages with *history digests*. Two history digests are equal iff the corresponding histories are equal. Thus, in the (Learn) and (Hybrid Learn) step, clients check history digests for equality. Intuitively, a history digest function takes as arguments a history h and a command c contained in that history. It then computes the smallest prefix of h containing c and returns the digest thereof. We refer the reader to the Appendix for an incremental digest implementation based on hashing.

O4 If the classic and fast histories diverge during normal operation, the protocol as described above prevents hybrid learning. A simple solution would be to periodically start a new round. However, this imposes a considerable overhead. Therefore, the idea is to have each acceptor locally *align* the fast history fh to the classic history h as follows. Periodically, fh is replaced with the *lub* of h and the largest prefix of fh that is compatible with h. We know from Section 4.2.1 that this is safe.

O5 We have optimized HP to adapt to a changing workload. Specifically, HP uses a double threshold (T_{high}, T_{low}) such that when the load increases above T_{high} (resp. decreases below T_{low}) the fast mode is switched off (resp. switched on). Changes in the load are monitored by the leader, who is counting proposals per time unit. The leader simply tells the clients (in "$2b$" messages) to stop (respectively start) sending commands to the acceptors.

4.3 Discussion

In this section, we provide a comparison of HP and FP in terms of their performance during normal operation. We argue that the cost of collision recovery in FP outweights the gain obtained from fast learning. The original FP paper [18] says: "If collisions are very rare, then starting a new round might be best. If collisions are too frequent, then classic Paxos might be better than Fast Paxos."

The latency of HP and FP equals two message delays in the absence of collisions. In the presence of collisions, HP requires three message delays and FP requires *six*. Hence, if FP is recovering from collisions only 25% of the time, then there is no (average) gain from fast learning.

The message complexity of HP is $4n$ messages per request. FP requires $3n + 1$ messages in the absence of collisions and $(6 + l)n$ messages otherwise, where l is the number of learners. If we consider that $l = n \geqslant 3$, and FP is recovering from collisions only 12% of the time, then FP has a higher (average) message complexity. Even without collisions, in FP the leader collects "$2bFast$" messages and checks if collisions occured, thus quickly becoming the bottleneck.

Our experiments with HP have revealed that with increasing load, the collision rate is growing faster than the server capacity utilization rate. For instance, we have observed that the servers are still underutilized when the rate of hybrid learning drops under 50% (with 99% commutable commands). In this situation, FP would spend > 50% of the time recovering from collisions, thus performing poorly compared with HP.

5 Evaluation

This section explores the performance characteristics of HP and compares it with existing approaches. As argued in Section 4.3 above, we expect HP to outperform FP in most situations, and therefore we omit a direct comparison. We substantiate our claim by showing that HP's latency often attains the theoretical minimum. We compare HP with CP and show that it performs significantly better under low to medium load *and* equally well under high load. Where appropriate, we also compare HP with Mencius [20].

5.1 Experimental Settings

We have implemented a simple banking system in which multiple clients share a bank account. Clients can deposit or withdraw money. The state consists of the balance of the shared account, and clients can issue *withdraw* or *deposit* commands. Executing *withdraw* \$100 subtracts \$100 in a state with at least \$100 and produces \$100 as output. Executing *deposit* \$20 adds \$20 and produces OK as output. Note that any two *deposit* commands are commutable because executing them in either order has the same effect. However, when one of the two operations is a *withdraw*, the order matters.

A scenario is modeled in which clients frequently deposit small amounts of money and less frequently withdraw larger amounts. Where the rate of *withdraw* commands matters, we use "HP-x" to denote runs of the HP protocol, where on average, one out of x commands is a *withdraw*. We use "CP3" (respectively "CP4") to denote the specific CP protocol where a command can be learned by a client after three (respectively four) message delays; CP4 relates to CP without speculation.

We ran experiments in the Emulab testbed [25] and we implemented all protocols in Java using the Neko [24] framework. The protocols are evaluated in a system with five servers ($f = 2$) except for fault-scalability, where the number of servers is scaled up to 21 ($f = 10$).

Client and server nodes are connected by links with a one-way delay of $20ms$ and a bandwidth of 100 Mbps. The chosen delays are comparable to the "Europe" WAN setting analyzed in section 1.1. The chosen network bandwidth models modern high-end WAN links such Geant2 [12]. Server nodes are 600 Mhz PCs with 256 MB memory running Fedora 6.

5.2 Latency

Figure 4(a) shows the average latency of of HP under low and medium load as the rate of *withdraw* operations is varied between 0% and 100%. Note that the withdraw rate corresponds to the probability of collisions. For a *withdraw* rate of 0.5% and load offered by 100 clients, HP has a 32% lower latency than CP3. This is close to the theoretical minimum. For a *withdraw* rate of 100% and load offered by 10 clients (between 0.1 and 0.2 Kops), HP still features a latency of 20% below the optimum of CP3.

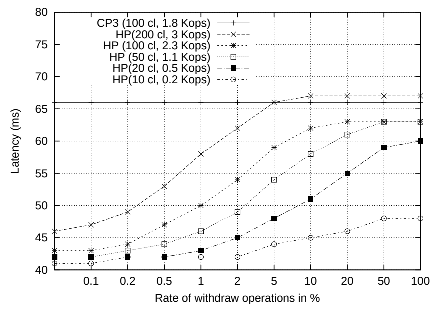
Figure 4(b) compares the latency and throughput of HP-500, CP3, CP4 with and without request batching (of 20 commands) as we vary the offered load. As illustrated, under low load, batching at the leader increases the latency of CP4 and CP3 but not that of HP because most commands ($> 90\%$) are chosen in the fast mode. On the other hand, batching increases peak throughput. In fact, with batching all protocols converge to the same peak throughput. Starting from a throughput of 6 Kops, the curves of HP and CP3 coincide because the fast mode is switched off.

Figures 4(c) and 4(d) illustrate the effectiveness of adaptive switching by means of a dynamic workload. The workload is organized as follows: 50% of the commands are sent under moderate load generated by 100 clients and 50% under high load generated by 1000 clients between $t = 68$ and $t = 98$. Figure 4(c) compares the average latency of HP with that of CP3 and CP4. We have measured the latency of HP with and without adaptive switching. The latter is referred to as nonadaptive. Figure 4(c) clearly shows that during the high load burst, the nonadaptive version of HP performs worst among all protocols. The explanation is the following. Batching offloads the leader and the acceptors become the bottleneck nodes because they process more messages. In contrast, the adaptive version shows a short spike after the load burst starts and a short tail after the burst ends. These can be attributed to conservative thresholds. Overall, the adaptive version of HP features the minimum latency of all protocols. Figure 4(d) compares the cumulative latency distribution of the four protocols under the same workload and confirms that adaptive HP performs best both under moderate and high load.

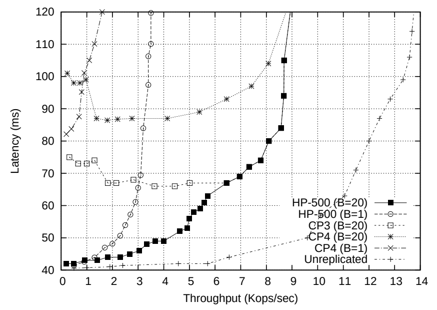
The experiments presented so far have been conducted in a setting in which the network is always timely. We now add a Pareto distribution to each link using the NetEm [14] utility. The one-way network delay now varies between $20ms$ and $60ms$. Pareto is a heavy tailed distribution, which models the fact that wide-area links are usually timely (e.g. 80% of the time) but can present high latency occasionally.

Figure 5(a) compares the latency and throughput of HP-500, CP3, CP4 with batching as we vary the offered load. The trends are the same as in a situation with no network variance. An important point is that all protocols have lower peak throughputs, including the unreplicated system. High variance results in packet reordering and packet retransmission at the transport protocol level (TCP), causing additional load in the bottleneck node. HP outperforms CP3 up to 60% of the peak throughput. Up to a throughput of 1 Kops, HP and the unreplicated system have comparable latencies. The performance profile of HP is somewhat surprising because with high network variance, the likelihood of collisions increases. E.g., under a link variance of $40ms$, if two interfering commands are sent within $40ms$ from each other, they might be accepted in different orders.

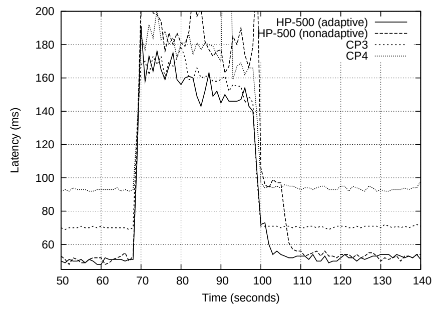
Figure 5(b) supports the above observation showing that under network variance, the latency of HP converges much faster to that of CP3. Nevertheless, under low load and a small fraction of withdraws, HP shows a latency improvement of up to 40% over CP3,



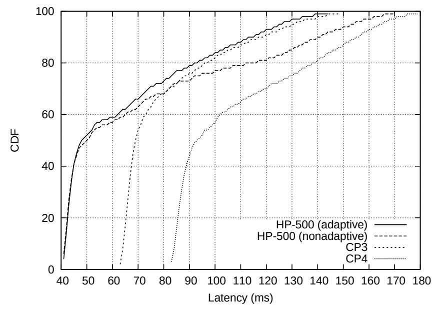
(a) Latency versus *withdraw* rate



(b) Latency versus throughput



(c) Average latency under a changing load (B = 20)



(d) Latency CDF under a changing load (B = 20)

Figure 4:

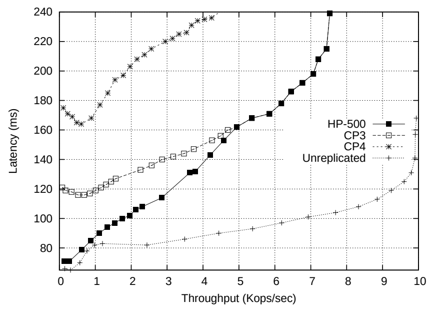
which is more than the theoretical maximum latency reduction of 1/3. An explanation could be that the longer it takes to run an instance of a protocol, the more likely it is to depend on a slow link in the critical path. In this particular case, this effect adds to the latency of CP3 and explains the measured latency difference.

Figure 5(c) compares the latency of HP-500, CP3 and Mencius [20] as more servers are added to the system. We are simulating a lightly loaded scenario with 20 clients. With Mencius, a server has to wait for all other servers to skip or to propose a command. For a fair comparison, all commands are commutable and thus Mencius can commit in only one message roundtrip after receiving a reply from all servers, which is optimal. Mencius' dependency on slow links grows as more servers are added and therefore its latency increases. In contrast, the latency of HP and CP3 remains roughly constant (CP3's latency even drops) because they wait for the fastest quorum. These results suggest that the latency of CP and HP strongly depends on how large is the fraction of nodes that form a quorum. We observe that the latency oscillates (and even drops) with this fraction.

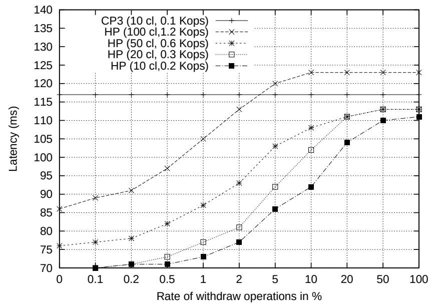
5.3 Throughput

We show now that the lower latency of HP does not come at the cost of lower throughput compared to CP.

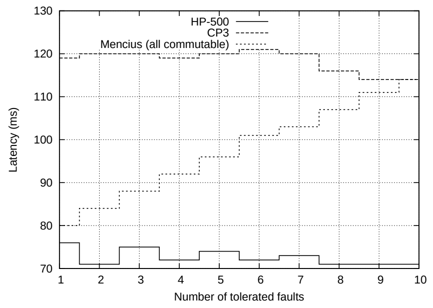
Figure 6(a) shows the throughput of HP-500, CP3 and CP4 with and without batching as the number of clients increases. All protocols scale equally well when batching is used; CP4 without batching scales poorly. Figure 6(b) compares the peak throughputs of HP (that equals CP3), CP4 and Mencius as the number of faulty servers tolerated increases. The throughput of Mencius is scaled down from 3GHz machines to ours (600MHz) using a factor of 1/4. The results show that HP outperforms all other protocols



(a) Latency versus throughput (B = 20)



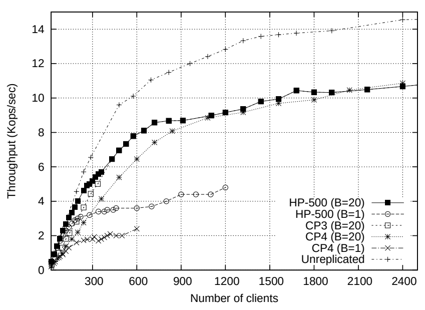
(b) Latency vs. withdraw rate



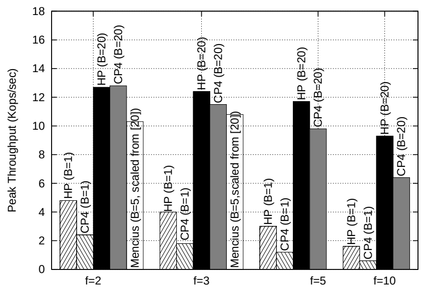
(c) Latency as f increases (20 clients)

Figure 5: Effect of network variance

except in the case of $f = 2$ with batching, when its peak throughput is comparable to CP4. The fault scalability of HP is superior to that of CP4 with and without batching. For $f = 10$ with batching, HP features 73% of the peak throughput for $f = 2$. In contrast, CP4's peak throughput drops down to 50%.



(a) Throughput as the number of clients increases.



(b) Throughput as f increases.

Figure 6:

6 Related Work

In the asynchronous model, the possibility of a single crash makes deterministic consensus impossible [11]. It has been shown that the FLP result can be circumvented by additional timing assumptions [10]. Chandra and Toueg [7] have introduced the concept of failure detectors, encapsulating timing assumptions, and Ω [6] has been shown to be the weakest failure detector for solving consensus. In his seminal Paxos paper [16], Lamport describes how to build a replicated state machine from consensus. Unlike many other consensus algorithms, Paxos is a perfect candidate for latency-critical applications because the read phase is done for infinitely many consensus instances together. Lamport's recent work on Fast Paxos [18] is based on the observation that the latency of CP can be further reduced if clients directly propose commands. At the heart of FP lies the idea of one-step consensus [4]. Pedone *et al.* [22, 23] have developed latency-efficient atomic broadcast algorithms based on the execution of a sequence of one-step consensus instances. The mentioned protocols suffer from collisions (which results in degraded latency) when multiple commands are sent at about the same time. In our prior work [9] we have developed consensus protocols tackling this problem and degrading gracefully in the presence of collisions. Guerraoui and Raynal [13] have developed a gracefully degrading consensus protocol that quickly chooses a value accepted by a fixed quorum. Charron-Bost and Schiper [8] present a consensus protocol with the minimum latency of FP and CP but only in failure-free runs.

Pedone and Schiper [21] have introduced the problem of generic broadcast, where one message delay can be saved by delivering messages in different but equivalent order. Recent work by Lamport on generalized consensus [17] borrows ideas from this work and extends FP to solve consensus on a growing set of partially ordered commands. Zielinski [26] proposes a protocol that combines FP and CP into a latency-optimal generic broadcast protocol. The protocol is not resilience optimal and incurs the expense of quadratic messages. Mencius [20] is a state machine replication protocol based on CP. The goal of Mencius is to reduce the load at the leader in order to prevent it from becoming a bottleneck. The main idea is to partition the set of consensus instances among several leaders. To reduce latency, Mencius assumes that each client has a local area connection with some leader. However, this cannot be guaranteed in a general system setting. Further, the system can only make progress if all leaders are correct. Therefore an eventually perfect failure detector [7] is assumed. These are all assumptions we do not make.

An earlier version of this work first introducing the concept of Hybrid Paxos is also discussed in some work by Junqueira *et al.* [15]. They study, by performing network simulations, when CP has a lower latency than FP.

Camargos *et al.*[5] developed a generalized consensus protocol that aims at improving the availability of CP by allowing multiple leaders to coexist. Under collisions, the protocol faces similar problems as FP and requires collision resolution. In contrast HP eliminates collision resolution, improving the latency and throughput of FP in stable runs.

7 Conclusion

We have developed Hybrid Paxos, a generalized consensus protocol that features minimal latency and maximum throughput in most situations. The core idea of HP is to add fast learning to CP. HP is to our knowledge the first generalized consensus protocol that attains the optimal latency of two message delays in the absence of collisions and three otherwise. Moreover, it has optimal latency, resilience and number of messages. We have shown that generalized consensus is a practical approach to replication in a WAN. Our experimental results demonstrate that HP can outperform state of the art protocols.

Acknowledgements We thank Flavio Junqueira for the inputs and discussions that helped to enhance the paper.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proc. of DISC*, pages 231–245, 1998.

[2] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proc. of WDAG*, pages 105–122, 1996.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Proc. of NSDI*, pages 253–266, 2004. *et al.*

[4] F. V. Brasileiro, F. Greve, A. Mostefaoui, and M. Raynal. Consensus in one communication step. In *Proc. of PACT*, pages 42–50, 2001.

[5] L. J. Camargos, R. M. Schmidt, and F. Pedone. Multicoordinated agreement protocols for higher availabilty. *NCA*, pages 76–84, 2008.

[6] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *Proc. of PODC*, pages 147–158, 1992.

[7] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *JACM*, (2):225–267, 1996.

[8] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. In *Proc. of PRDC*, 2006.

[9] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *Proc. of DSN*, pages 137–146, 2006.

[10] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *JACM*, (2):288–323, 1988.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *JACM*, (2):374–382, 1985.

[12] Geant2. Pan-european backbone network. Website. `http://www.geant2.net`.

[13] R. Guerraoui and M. Raynal. The information structure of indulgent consensus. *IEEE Trans. Comput.*, 53(4), 2004.

[14] S. Hemminger. Network emulation with netem. In *Proc. of LCA*, 2005.

[15] F. Junqueira, Y. Mao, and K. Marzullo. Classic paxos vs. fast paxos: caveat emptor. In *HotDep workshop*, Berkeley, CA, USA, 2007. USENIX Association.

[16] L. Lamport. The part-time parliament. *ACM Trans. Computer Systems*, (2):133–169, 1998.

[17] L. Lamport. Generalized consensus and paxos. In *MSR-TR-2005-33*, 2005.

[18] L. Lamport. Fast paxos. *Distrib. Comp.*, 19(2), 2006.

[19] L. Lamport. Lower bounds for asynchronous consensus. *Dist. Comp.*, 19(2), 2006.

[20] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machine for wans. In *OSDI*, 2008.

[21] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Dist. Comp.*, 15(2), 2002.

[22] F. Pedone and A. Schiper. Optimistic atomic broadcast: A pragmatic viewpoint. *Journal of Theoretical Computer Science*, 291(1):79–101, 2003.

[23] F. Pedone, A. Schiper, P. Urbán, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proc. of EDCC*, pages 44–61, 2002.

[24] P. Urbán, X. Defago, and A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Proc. of Information Networking*, pages 503–511, 2001.

[25] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of OSDI*, pages 255–270, 2003.

[26] P. Zielinski. Optimistic generic broadcast. In *Proc. of DISC*, pages 369–383, 2005.

A The `historyDigest` Function

The `historyDigest` function assumes a hash function H with the following property: for any two values x and y it holds that $H(x) = H(y)$ if and only if $x = y$.

Recall that, our banking example, described in section 5.1, provides two command types: *deposit* and *withdraw*. Commands of type *withdraw* are not commutable with any other command and *deposit* commands commute with each other. The description of HP in section 4 abstracts away the details of the command history data structure to simplify the presentation. The command history type maintains the following attributes: a *type* field to distinguish between a *classic* and *fast* history, a *cache* field which stores for each command the digest of the history up to that command and the field *digest*, a byte array, used to store the history digest. The *digest* field consists of an incremental hash value of the history (described below). The history type has the additional application specific field *Deposit*, which is an ordered queue containing *deposit* commands.

Since commands of type *withdraw* are not commutable with any other command, for every withdraw command w in a command history h we can distinguish between commands which are ordered before or after w in h. To compute incremental history digests for history h using `historyDigest` the idea is the following: the digest of the history up to the last *withdraw* command w appended to history h is maintained in $h.digest$. For any deposit command d appended after w and before the next *withdraw* command, the history up to d is completely determined by the history up to w. This is true because it subsumes all operations that must be ordered before d. Thus, the digest of d can be computed incrementally as $H(h.digest \circ d.id)$ (line 6.7), where $\circ$ is the concatenation operator and $d.id$ is the command identifier. When the next withdraw command w' is appended, $h.digest$ is updated. The new digest of h up to w' depends on $h.digest$ (which is the digest of h up to w) and all deposit commands which are appended to h between w and w'. Such commands $d_1, d_2, \ldots$ are collected in the ordered queue *Deposit* (Whenever a digest for a deposit command is computed, the command is added to *Deposit* in line 6.6). Therefore, $h.digest$ is updated with $H(h.digest \circ d_1.id \circ d_2.id \circ \ldots \circ w'.id)$ (line 6.10) and *Deposit* (lines 6.8–6.9) is cleared.

Additionally, when a digest for command c is computed, c and the corresponding digest are added to $h.cache$ (line 6.12). When the `historyDigest` function is called for a *classic* history h and an entry for c is contained in $h.cache$ (line 6.3), the corresponding digest is directly returned without any computation (line 6.4).

B Proof of Correctness

In this section we show that Algorithm 1, 2, and 3 solves the generalized consensus problem (see Section 2). We do this by showing that HP satisfies *Conservatism, Consistency, Progress* and *Nontriviality.* We first give some helpful definitions and prove a set of auxiliary lemmas.

Definition 1 (Accepted in round r)**.** *A command history h is accepted as classic history (respectively as fast history) in round r by acceptor j iff $chist_j \sqsupseteq h \wedge r_j = r \wedge \neg recover_j$ (respectively iff $fhist_j \sqsupseteq h \wedge r_j = r \wedge \neg recover_j$). A command history h is accepted in round r iff h is accepted as classic or fast history in round r by some acceptor.*

Definition 2 (Chosen in round r)**.** *A command history h is chosen in round r iff h is accepted in round r*

(Classic) *as classic history by a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors*

OR

(Fast) *(1) as fast history by a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors and (2) as classic history by some acceptor.*

Lemma 1. *If any two classic histories h and h' are accepted in round r, then h and h' are compatible.*

Proof. By the FIFO property of the channels, if any two histories h and h' are accepted in round r, then both are prefixes of the history proposed by the leader of round r. Thus, h and h' are compatible. ☐

Lemma 2. *If any two histories h and h' are chosen in round r, then h and h' are compatible.*

Proof. If h and h' are chosen in round r, then by Definition 2, both are accepted as classic histories in round r. By Lemma 1, h and h' are compatible. ☐

Lemma 3. *If history h is chosen in round r and h is accepted by a quorum Q, then for every acceptor $j \in Q$ holds: if j accepts h' at any later time in round r, then h' is an extension of h.*

Proof. We treat the two cases when h is accepted as classic history (case A) and h is accepted as fast history (case B) separately.

Case A: Note that the classic history accepted by any acceptor j in round r is a prefix of the history accepted by j at any later time in r (see line 2.18). This also applies to classic histories that are chosen.

Case B: According to Definition 2, if h is chosen, then h is the prefix of some classic history accepted in round r. By Lemma 1, it follows that h is compatible with *every* classic history accepted in round r. Hence, for every $j \in Q$, it holds that $h \sqsubseteq fhist_j$ and h is compatible with $chist_j$. Therefore, $\bigsqcup\{pref \sqsubseteq fhist_j : pref$ is compatible with $chist_j\}$ is an extension of h. Thus, once h is chosen, $fhist_j$ is updated only with extensions of h (see lines 2.14, 2.20, 2.26). ☐

Lemma 4. *If h is chosen in round r and h' is accepted in round $r' > r$ then h' is an extension of h.*

Proof. We show the lemma by induction on round number k, where $r < k \leqslant r'$. For any k, let g be the history accepted by any acceptor in round k. Note that g is necessarily an extension of the history rh

contained in the "$2aStart$" messages of round k. This is true because in round k (a) any acceptor j accepts client requests only after $fhist_j$ and $chist_j$ have been initialized with rh (lines 2.14) and (b) $rh \sqsubseteq chist_j \wedge rh \sqsubseteq fhist_j$ is invariant (lines 2.18-2.20). Therefore, it is sufficient to show that rh, i.e., the history picked by the leader of round k is an extension of h.

Base step: let k be the lowest round number $k > r$ in which some acceptor accepts a history. We show that if h is chosen in round r, then the history picked by the leader of round k is an extension of h. We distinguish the cases when h is chosen (Case A) as classic history or (Case B) as fast history.

Case A implies that a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors have accepted h. Let Q' be the set of $n - f$ acceptors from which the leader of round k receives "$1b$" messages and let $R \subseteq Q'$ be the subset of all acceptors that have voted in round r. By the definition of R and the quorum intersection property, $Q \cap Q' \subseteq R \neq \varnothing$. By Lemma 3, some acceptor in R reports an extension of h and by Lemma 1, all classic histories reported by acceptors in R are compatible. Therefore, `pickClassicHistory` (lines 4.1-4.2) returns an extension of h. Finally, the recovered history is also an extension of h (see line 1.25).

Case B implies that a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors has accepted h as fast history and h is the prefix of some accepted classic history. Let Q' be the set of $n-f$ acceptors from which the leader of round k receives "$1b$" messages. Further let $R \subseteq Q'$ be the set of acceptors that have voted in r and let $M \subseteq R$ denote the set of acceptors that have accepted h in r. By Lemma 3, all acceptors in M report extensions of h. By the intersection property of FQ and Q', $|M| > |R \backslash M|$. Let $Comp$ denote the set of fast histories reported by the acceptors in R. It is not difficult to see hat after lines 5.3-5.7 are executed, $Comp$ contains (a) only compatible histories and (b) some history in $Comp$ is an extension of h. The latter is true because a majority of histories contained in $Comp$ are extensions of h. Therefore, fh, the history returned by `pickFastHistory` (lines 5.1-5.9) is an extension of h. It remains to show that the recovered history is an extension of h. Recall that the assumption that h is chosen implies that h is the prefix of some accepted classic history. By Lemma 1, h is compatible with every classic history and hence with ch, returned by `pickClassicHistory`. Thus, the recovered history $\bigsqcup\{pref \sqsubseteq fhp : pref$ compatible with $ch\}$ is an extension of h.

Induction step: we show that the Lemma is true for $k = r'$ under the assumption that it holds for all $r < k < r'$. Recall that it is sufficient to show that the history picked by the leader of round r' is an extension of h. Let Q' be the quorum of $n - f$ acceptors from which the leader of round r' collects "$1b$" messages. Further, let $R \subseteq Q'$ denote the subset of acceptors that have voted in round r_{Max} (line 1.20). Note that by the quorum intersection property, $r_{Max} \geqslant r$. The case $r_{Max} = r$ is covered by the proof of the base step. Therefore, we only show the case $r_{Max} > r$. The induction hypothesis implies that every acceptor in R has accepted in round r_{Max} an extension of h as classic history. Since classic histories never shrink throughout a round, all histories reported by acceptors in R are extension of h. Further, by Lemma 1, they are compatible histories. Hence, the history picked by the leader of round r' is an extension h. □

Lemma 5 (Consistency). *If client cl learns command history h and client cl' learns command history h', then h and h' are compatible.*

Proof. We assume without loss of generality that h is chosen in round r and h' is chosen in round $r' \geqslant r$. If $r = r'$, then by Lemma 2, h and h' are compatible. Else if $r' > r$, then Lemma 4 implies that h and h' are compatible because they have h' as common extension. □

Lemma 6 (Progress). *If a command c is proposed by a non-faulty client cl, then cl eventually learns a command history containing c.*

Proof. By the property of Ω, exactly one non-faulty leader ld is eventually elected forever. By the logic of Phase 1 (line 1.28-1.30), ld eventually starts a round with a number that is greater than any other round number. Therefore, ld and a quorum Q of $\lceil \frac{n+1}{2} \rceil$ correct acceptors eventually join the highest round ever.

We assume by contradiction that cl never learns a history containing c. This implies that cl keeps on resending c to all replicas forever. Thus, ld eventually sends a "2a" message containing c to all acceptors unless it has already did so. In any case, every acceptor in Q eventually accepts a classic history containing c. From this point on, every acceptor in Q that receives a "*propose*" message from cl sends a "2b" message to cl (line 2.24). Thus, cl eventually collects identical "2b" messages from $\lceil \frac{n+1}{2} \rceil$ acceptors (line 3.18) voting for histories that contain c. Hence, cl stops resending command c, a contradiction. □

Lemma 7. *If h and h' are classic histories accepted in some round r and both h and h' contain c, then $h \sqcap h'$ contains c.*

Proof. By Lemma 1, h and h' are compatible histories. By axiom CS4 for command structures [17] (here called histories), the greatest lower bound $h \sqcap h'$ also contains c. □

Lemma 8. *If two accepted command histories h and h' both contain command c and* `historyDigest`*(c, h) =* `historyDigest`*(c, h'), then $h \sqcap h'$ contains c.*

Sketch. Conceptually, `historyDigest`(c, h) (respectively `getHistoryDigest`(c, h')) first computes the smallest prefix sp (respectively sp') of h that contains c, and then applies incremental hashing to sp (respectively sp') as described above in appendix A. It is not difficult to see that for the described banking example, if the incremental digests of sp and sp' are equal, then sp and sp' must also be equal. As sp and sp' are prefixes of $h \sqcap h'$, c is also contained in $h \sqcap h'$. □

Lemma 9 (Conservatism)**.** *If a client learns a command history h, then h is chosen.*

Proof. Command history h is learned in line 3.19.

Case 1 (learning): a quorum Q of $\lceil \frac{n+1}{2} \rceil$ acceptors j have accepted classic history h_j containing command c, where c is the last command issued by the client. By Definition 3, $\bigsqcap\{h_j | j \in Q\}$, the greatest common classic history accepted by every acceptor in Q, is chosen. We only have to show that the learned history h is a prefix of $\bigsqcap\{h_j | j \in Q\}$. By Lemma 7, $\bigsqcap\{h_j | j \in Q\}$ also contains command c. Since h is a prefix of any classic history containing c, and $\bigsqcap\{h_j | j \in Q\}$ is an extension of some classic history containing c, $\bigsqcap\{h_j | j \in Q\}$ is also an extension of h.

Case 2 (fast learning): a quorum FQ of $\lceil \frac{n+f+1}{2} \rceil$ acceptors j have replied with history digest hd_j = `historyDigest`(c, h_j), where c is the last command issued by the client and h_j is the history accepted by j. Additionally, some h_j is a classic history. By Definition 3, $\bigsqcap\{h_j | j \in FQ\}$, the greatest common history accepted by every acceptor in FQ, is chosen. Therefore, it is sufficient to show that the learned history h is a prefix of $\bigsqcap\{h_j | j \in FQ\}$. Note that c is contained in h_j, otherwise j does not respond. Furthermore, for any two acceptors i and j it holds that $hd_i = hd_j$. Hence, by Lemma 8, $\bigsqcap\{h_j | j \in FQ\}$ contains command c. Since h is a prefix of any classic history containing c, and $\bigsqcap\{h_j | j \in FQ\}$ is an extension of some classic history, $\bigsqcap\{h_j | j \in FQ\}$ is also an extension of h. □

Lemma 10 (Nontriviality)**.** *Every chosen command is proposed by some client.*

Proof. Every chosen command is proposed by some client because no faulty acceptor can undetectably corrupt commands. □

Theorem 1. *Algorithm Hybrid Paxos in Figure 1, 2, and 3 solves generalized consensus on command histories.*

Proof. The theorem follows directly from Lemma 5, 6, 9 and 10. □

```
1.1  $k = 0, 1, 2, \ldots$                                        /* round index */
1.2  $rnd, r_{Max}$ integer, initially 0                           /* round number */
1.3  $chist$ command history, initially $\bot$                     /* is an extension of every chosen history */
1.4  $ch, fh$ command histories                                     /* returned by pick history functions */
1.5  $pfh$ command history, initially $\bot$                       /* largest prefix of $fh$ compatible with $ch$ */
1.6  $recover$ boolean, initially true                              /* indicates the current operational phase, recovery or normal */
1.7  $1bSet$ set of messages, initially empty                       /* set of "$1b$" messages for the current round */
1.8  $leader$ boolean, initially false                              /* this proposer believes to be the leader */

1.9  upon LEADER do
1.10     $leader \leftarrow$ true
1.11     $1bSet \leftarrow \varnothing$
1.12     $recover \leftarrow$ true
1.13     $rnd \leftarrow k \cdot n + l$
1.14     send $\langle$"$1a$", $rnd\rangle$ to all acceptors

1.15 upon NOTLEADER do
1.16     $leader \leftarrow$ false

1.17 upon receive $\langle$"$1b$", $rnd$, $vr$, $fh$, $ch\rangle$ from $j$ do
1.18     $1bSet \leftarrow 1bSet \cup \{(vr, fh, ch, j)\}$

1.19 upon $|1bSet| \geqslant n - f \wedge recover$ do
1.20     $r_{Max} \leftarrow \mathrm{Max}(\{r \mid (r, fh, ch, j) \in 1bSet\})$
1.21     $1bSet \leftarrow \{(vr, fh, ch, j) \in 1bSet | vr = r_{Max}\}$
1.22     $ch \leftarrow$ pickClassicHistory($1bSet$)
1.23     $fh \leftarrow$ pickFastHistory($1bSet$)
1.24     $pfh \leftarrow \bigsqcup\{pref \sqsubseteq fh : pref$ compatible with $ch\}$
1.25     $chist \leftarrow pfh \sqcup ch$
1.26     send $\langle$"$2aStart$", $rnd, chist\rangle$ to all acceptors
1.27     $recover \leftarrow$ false

1.28 upon receive $\langle$"$nack$", $r\rangle$ : $leader \wedge r > rnd$ do
1.29     $k \leftarrow \lfloor \frac{r}{n} \rfloor + 1$
1.30     trigger LEADER

1.31 upon receive $\langle$"$propose$", $c\rangle$ from client $cl$ : $leader \wedge \neg recover$ do
1.32     if $c \notin chist$ then
1.33         $chist \leftarrow chist \bullet c$
1.34         send $\langle$"$2a$", $rnd, c, cl\rangle$ to all acceptors
1.35     send $\langle$"$2b$", $rnd, c,$ historyDigest($c, chist$)$\rangle$ to client $cl$
```

Algorithm 1: Algorithm of Leader l

```
2.1  $rnd$ integer, initially 0                      /* round number, determines leader */
2.2  $fhist$ command history, initially $\perp$      /* fast history */
2.3  $chist$ command history, initially $\perp$      /* classic history */
2.4  $pfh$ command history, initially $\perp$        /* largest prefix of $fhist$ compatible with $chist$ */
2.5  $recover$ boolean, initially true               /* accepts client proposals only if not in recovery */
2.6  upon receive $\langle$"1$a$", $r\rangle$ : $r > rnd$ do
2.7      send $\langle$"1$b$", $r, rnd, chist, fhist\rangle$ to leader
2.8      $recover \leftarrow$ true
2.9      $rnd \leftarrow r$

2.10 upon receive $\langle *, r, *\rangle$ : $r < rnd$ do
2.11     send $\langle$"$nack$", $rnd\rangle$ to $r(\bmod n)$

2.12 upon receive $\langle$"2$aStart$", $r$, $rh\rangle$ : $r \geqslant rnd$ do
2.13     $rnd \leftarrow r$
2.14     $fhist \leftarrow chist \leftarrow rh$
2.15     $recover \leftarrow$ false

2.16 upon receive $\langle$"2$a$", $r, c, cl\rangle$ : $r \geqslant rnd$ do
2.17     $rnd \leftarrow r$
2.18     $chist \leftarrow chist \bullet c$
2.19     $pfh \leftarrow \bigsqcup\{pref \sqsubseteq fhist : pref$ compatible with $chist\}$
2.20     $fhist \leftarrow chist \sqcup pfh$
2.21     send $\langle$"2$b$", $rnd, c,$ historyDigest$(c, chist)\rangle$ to client $cl$

2.22 upon receive $\langle$"$propose$", $c\rangle$ from client $cl \wedge \neg recover$ do
2.23     if $c \in chist$ then
2.24         send $\langle$"2$b$", $rnd, c,$ historyDigest$(c, chist)\rangle$ to client $cl$
2.25     if $c \notin fhist$ then
2.26         $fhist \leftarrow fhist \bullet c$
2.27         send $\langle$"2$bFast$", $rnd, c,$ historyDigest$(c, fhist)\rangle$ to client $cl$
```

Algorithm 2: Algorithm of the Acceptors

```
3.1  $lc$ command, initially $\perp$                                   /* last command issued */
3.2  $2bSet$ set of pairs (integer, digest), initially $\varnothing$   /* "2b" messages */
3.3  $2bFastSet$ set of pairs (integer, digest), initially $\varnothing$   /* "2bFast" messages */
3.4  $ref$ byte array, initially $null$                                /* classic history digest used as reference */
3.5  $pending$ boolean, initially false                                /* pending operations */
3.6  $rnd$ integer, initially 0                                        /* round number determining the current leader */
3.7  upon receive $\langle \text{TYPE}, r, c, hd\rangle$ from $j$ : $\text{TYPE} \in \{$"$2b$","$2bFast$" $\} \wedge r \geqslant rnd \wedge c = lc \wedge pending$ do
3.8      if $r > rnd$ then
3.9          $rnd \leftarrow r$
3.10         $2bFastSet \leftarrow 2bSet \leftarrow \varnothing$
3.11         $ref \leftarrow null$
3.12     if TYPE = "$2b$" then
3.13         $2bSet \leftarrow 2bSet \cup \{(j, hd)\}$
3.14         if $ref = null$ then
3.15             $ref \leftarrow hd$
3.16     $2bFastSet \leftarrow 2bFastSet \cup \{(j, hd)\}$
3.17     $2bFastSet \leftarrow 2bFastSet \setminus \{(i, hd) \in 2bFastSet | hd \neq ref\}$
3.18     if $(|2bFastSet| \geqslant \lceil \frac{n+f+1}{2} \rceil \wedge |2bSet| \geqslant 1) \vee (|2bSet| \geqslant \lceil \frac{n+1}{2} \rceil)$ then
3.19         learn($lc$)
3.20         $2bFastSet \leftarrow 2bSet \leftarrow \varnothing$
3.21         $ref \leftarrow null$
3.22         $lc \leftarrow \perp$
3.23         $pending \leftarrow$ false
3.24 upon propose($c$) $\wedge \neg pending$ do
3.25     $pending \leftarrow$ true
3.26     $lc \leftarrow c$
3.27     while $pending$ do
3.28         send $\langle$"propose", $c\rangle$ to all servers
```

Algorithm 3: Algorithm of the Clients

```
4.1  $ch \leftarrow \bigsqcup\{h | (vr, fh, h, j) \in 1bSet\}$
4.2  return $ch$
```

Function `pickClassicHistory(`$1bSet$`)`

```
5.1 $Comp \leftarrow \{(h,j) | (vr,h,ch,j) \in 1bSet\}$
5.2 if $Comp$ is incompatible then
5.3     repeat
5.4         $Comp \leftarrow Comp \setminus \{(h,i),(h',j) | i \neq j \wedge h$ incompatible with $h'\}$
5.5         $p \leftarrow h \sqcap h'$
5.6         $Comp \leftarrow Comp \cup \{(p,i),(p,j)\}$
5.7     until $Comp$ is compatible
5.8 $fh \leftarrow \bigsqcup\{h | (h,j) \in Comp\}$
5.9 return $fh$
```

Function `pickFastHistory`($1bSet$)

```
6.1  $tmp$ type string, initially empty
6.2  $digest$ type byte array, initially empty
6.3  if $history.cache.contains(cmd.id) \wedge history.type = $ CLASSIC then
6.4      return $history.cache.$get$(cmd.id)$;   /* return digest from cache */

     /* construct digest incrementally */
6.5  if $cmd.type = $ DEPOSIT then
6.6      $history.Deposit.$push$(cmd.id)$
6.7      $digest \leftarrow H(history.digest \circ cmd.id)$
     else
         /* WITHDRAW command */
6.8      while $history.Deposit \neq \varnothing$ do
6.9          $tmp \leftarrow tmp \circ history.Deposit.$pop$()$
6.10     $digest \leftarrow H(history.digest \circ tmp \circ cmd.id)$
6.11     $history.digest \leftarrow digest$
6.12 $history.cache.$put$(cmd.id, digest)$   /* cache history digest */
6.13 return $digest$
```

Function `historyDigest`(*cmd, history*)