Hydrofoil: Reducing Downtime by Combining Leader-Based and Randomized Replicated State Machines

Henry Tang
Class: 2023
Adviser: Wyatt Lloyd

Abstract

This thesis introduces Hydrofoil, a new linearizable replicated state machine that combines both leader-based and randomized leaderless approaches. While the leader is active, Hydrofoil is able to perform as efficiently as other leader-based protocols such as Raft. In addition, by leveraging the power of Ben-Or, a randomized consensus algorithm, Hydrofoil is able to make progress even when the leader is slow, or during the period when a new leader is being elected.

Acknowledgements

As I write these words, I am acutely aware that my time as an undergrad at Princeton is drawing ever closer to its conclusion. This has been a spectacular four-year journey, one filled with ups and downs and turns and twists, that has miraculously brought me to where I am today: sitting in front of my monitor, putting the finishing touches to my thesis. I have many people to thank for getting me this far, far too many to list. Here is my best attempt.

Professor Wyatt Lloyd, thank you so much for being an incredible advisor. I am so grateful to have had the opportunity to work with you on this thesis. Your insight, acumen, and breadth of knowledge have been instrumental to the completion of this thesis. Back when I was a lowly naive sophomore, you were the one who introduced me to the exciting field of distributed systems. Since then, you've guided me when I was lost, encouraged me when I was stressed, and congratulated me when I finished. Ok technically, this last point is just my wishful thinking, but we shall know after my presentation tomorrow! Regardless, this thesis has been a thrilling ride that I would choose to do again if given the chance, although only once I catch up on some sleep.

Professor Jennifer Rexford, thank you for introducing me to computer systems. I still remember taking your class as a frosh, and getting bewildered as I saw assembly code for the first time. I think it's safe to say I've come a long way. Professor Karthik Narasimhan and Ameet Deshpande, thank you both for providing me with my first research opportunity last year. I'll never forget debugging transformer models at 3 am or accidentally deleting our training dataset. Thanks for your understanding regarding this mishap, and for guiding me through the process of regenerating the data. Professor Zachary Kincaid, thanks for teaching not one, but two of my favourite classes at Princeton: compilers and automated reasoning about software. As you would say, cheers!

Aaron, thanks for being an amazing partner in so many classes, and for enlightening me about so many cool tidbits of information. It's awesome that we're both working with Professor Lloyd for our theses, and hopefully, we'll both do great in the presentation tomorrow. We've taken so many classes together that it's crazy to think we only met in the second semester of our sophomore year.

Vaibhav, thanks for always being so supportive and nice. It's kind of crazy that we bumped into each other during international orientation, and then promptly never interacted again until our junior years! I'm glad we became good friends, and I hope you've enjoyed these late-night working sessions in the Wucox and Choi dining halls as much as I have.

Freddy, thanks for being such a sociable individual and for carrying me through quite a few difficult problem sets. I'm probably not the intended audience for most of your jokes, but they do make bring a smile to my face. It's been a lot of fun hanging out in your room, which I can always trust to have food and drinks.

Ruijie, thanks for being a great friend and roommate, and putting up with all my numerous shenanigans. I have fond memories of our exciting adventures exploring the dangerous depths of Moffett Hall. Make sure you fulfill your promise to solve a 5x5 Rubik's cube soon. I will be testing you, and you will not be allowed to use time dilation this time.

Michael, thanks for being another incredible roommate. Your ability to complete your work while training for the football team every day is remarkable. I appreciated how you would always wake up early because it helped me maintain a good sleep schedule too (which has since gone out the window). I'm sorry for the one time I accidentally locked you into the bathroom!

Hyunsung, thanks for your endlessly positive outlook on life, it really is quite inspiring. Dina, thanks for being a great RCA and really caring about all your zees. Agnes, thanks for being a great international coordinator and for introducing me to this amazing campus. Nalin and Ish, thanks for the great times working together on those physics problem sets, and for letting me join your dorm during my junior fall break. Sacheth, thanks for your excellent sense of humour and boundless sarcasm that always makes me laugh.

Finally, I would like to thank my parents for being the best parents that anyone could ever ask for. Thanks for supporting me every step of the way, for always listening to my complaints and problems, and for offering me concrete strategies to deal with them. Thanks for being willing to take my calls no matter how late or egregious the hour is. Thanks for instilling in me the curiousity to learn, the perseverance to continuously pursue my goals, and the willingness to take risks. None of this would have been possible without you, and I am eternally grateful.

Contents

Chapter 1

Introduction

1.1 Problem Background

Formally, a state machine consists of three components: an initial state, a set of possible states, and a set of transitions between these states. These transitions are triggered in response to commands issued through requests from clients. An important application of state machines is in distributed systems, where state machines are practically implemented by a collection of servers that can accept requests from clients, execute the commands, and return responses.

A highly sought after property of these state machines is fault tolerance: the capability of a system to continue functioning even when a portion of the servers fail. Fault-tolerance ensures that many critical components used in the modern internet, including databases, service managers, and web servers, rarely suffer downtime and do not face regular outages. Intuitively, fault-tolerance is achieved by replicating data across a variety of servers: such systems are called replicated state machines (RSMs).

Typically in RSMs, we only handle the possibility of fail-stop failures, which are failures that cause a server to crash and stop executing. This is in contrast to blockchains which must consider Byzantine failures, where servers are taken over by malicious adversaries and can collude against the overall system. Nevertheless, handling fail-stop failures is still quite tricky, as there can be conflicts between the commands being executed on different replicas. Different methods of dealing with these conflicts results in different guarantees of a fault-tolerant system: these guarantees are known as consistency models. A particularly restrictive but useful consistency model is known as linearizability. In a linearziable system, any set of invocations and responses $\mathcal{S}$ satisfy the following two properties [14, 37]:

1. The invocations and responses in $\mathcal{S}$ can be ordered into a serializable sequential history $\mathcal{H}$ of completed operations, such that if the every completed operation in $\mathcal{H}$ was executed in order, it would lead to the same result as the execution from $\mathcal{S}$.

2. If a client receives a response for command σ_1 before any client sends a request for σ_2 to the system in real-time, then σ_1 is ordered before σ_2 in $\mathcal{H}$.

Intuitively, these two conditions mean that the linearizable system executes as if it were a single machine. Since the 1990s, many fault-tolerant linearizable RSMs have been developed. These systems are heavily in use today, with Raft [9], and different varieties of Multi-Paxos [21, 26, 29, 19, 41, 15] currently accepted as the standard.

However, while these two systems perform well, they rely on the notion of a leader. This means that the system inevitably slows down when the leader is slow, and cannot make any progress when a new leader is being elected. In this thesis, we introduce Hydrofoil, a new protocol that aims to address these issues by combining leader by combining leader-based protocols like Raft with randomized leaderless protocols.

1.2 Related Work

The first linearizable RSM protocols were Viewstamped Replication [34] and Paxos [20], which were similar but independently developed in the 1980s. Technically, while Viewstamped Replication is a complete replication service, Paxos is only a consensus protocol that enables servers to agree on a singular value. However, consensus algorithms can often be easily modified to replicate an entire log: each entry is agreed upon separately using consensus. As a result, not too long after the publication of the Paxos protocol in 1998, a true linearizable RSM based on Paxos was developed and named Multi-Paxos [21]. Multi-Paxos also introduced the idea of long-lived stable leader to reduce round trips [21].

Since then, numerous variants to this algorithm have been developed that handle various new considerations. Raft is a standardized and simplified variant of Multi-Paxos that's widely in use today [9]. Generalized Paxos improves message delays by allowing clients to bypss the leader send proposals directly to replicas, and by allowing non-conflicting commands to be executed in any order [22]. Mencius distributes the load more fairly among all replicas by rotating the leader among the replicas [2]. Egalitarian Paxos (EPaxos) allows any replica to propose entries in a decentralized manner, which theoretically allows for higher throughput [32], although more recent studies have shown it demonstrates much worse tail latency [40]. Gryff unifies consensus with shared registers for lower tail latency [5]. In addition to these, many other protocols exist including Disk Paxos [13], Cheap Paxos [25], Fast Paxos [23], Stoppable Paxos [28], Vertical Paxos [24], Zab [18], NOPaxos [27], and SDPaxos [43]. While the examples above only handle fail-stop failures, Byzantine fault-tolerant RSMs do exist and are often used to implement blockchains. Examples include Practical Byzantine Fault Tolerance [8], Aardvark [10], HoneyBadgerBFT [31], and DispersedLedger [42].

Together, RSMs are used as a critical component in many of today's distributed databases, including Google's Spanner [11], Cockroach Labs' CockroachDB [38], Amazon's DynamoDB [36]. They are also used in distributed coordination services [6, 16], cloud storage systems [4, 7], and service managers [17, 30].

One problem the protocols mentioned above all have is that they can have noticeably higher latency when a single replica (typically the leader, or the designated replica in the case of EPaxos) is slow. Since the server hasn't technically crashed, the protocol still works, but operates at significantly reduced speeds. In addition, when a leader does fail, the process of electing a new leader could result in dueling proposers, which can theoretically stall the state machine indefinitely. Since 2020, there has been more of a focus on developing replication systems that can address this problem. In particular, Copilot is a new model that handles up to 1 slow replica by having two leaders with separate logs, who take over the work of the other leader when necessary [33].

Orthogonally to the development of Paxos based algorithms, a number of theoretical results have been published on using randomized algorithms to achieve consensus, starting with Ben-Or's algorithm in 1983 [3]. Since Ben-Or is a randomized protocol, it can circumvent the FLP impossiblility result [12], and will always be able to reach consensus even in asynchronous network conditions [1]. Remarkably, even though the Ben-Or protocol uses randomness as a key component, it is guaranteed to be live and terminate [1]. Recently, the first practical randomized protocol, named Rabia, was introduced [35]. However, the paper was primarily focused on demonstrating the practicality of using a leaderless protocol, and has a number of limitations. It assumes messages are always delivered, and it does not produce performance benefits compared to Raft, especially in the wide-area case.

Sporades is a recent protocol that is safe even in asynchronous networks, while maintaining good performance in the wide-area setting [39]. It does this by switching between synchronous and asynchronous modes of operation. However, it assumes the presence of a global common coin that returns the same value for every replica, which is not a realistic assumption in practical scenarios.

Chapter 2

Design

2.1 Overview

Hydrofoil is a combination of a leader-based protocol and a randomized leaderless protocol. It maintains linearizability and is fault-tolerant as long as a majority of replicas are working. The leader-based component, which we call Raft+, is similar to Raft, while the randomized leaderless protocol, which we call as Ben-Or+, is inspired by Rabia and relies on Ben-Or as a central component. At a high level, Hydrofoil typically uses a leader to replicate and commit entries. However, when the leader is slow, Ben-Or+ takes over and commits entries until the leader has recovered or a new leader is elected. To accomplish this, Hydrofoil allows replicas to commit entries in two different ways: through ReplicateEntries RPCs in Raft+, or through the leaderless Ben-Or+ procedure. The protocol is called Hydrofoil because it is able to continue to make progress, i.e. stay afloat, more often than Raft.

2.2 Assumptions

We are assuming we have a system of $n \coloneqq 2f + 1$ servers, with up to f servers capable experiencing non-Byzantine faults at a time. We assume the system is not completely asynchronous, and there does not exist an adversary that can rearrange the order of all the messages. We do not assume reliable network connections, i.e. we do not assume messages are delivered in order, or even delivered at all.

2.3 Definitions

Here, we establish a few norms of notation that we will be using throughout the rest of this thesis. A *replica* refer to any server involved in the protocol. *Notifications* are network messages sent from one replica to another, without the need for a response. *RPCs* are pairs of network messages that follow a request-response paradigm. A *message* refers to either a notification or an RPC. When the client wants to execute a new *command*, it sends over a proposal request containing the command. When this command is stored on a replica, due to additional fields such as timestamp and server ID metadata associated with it, it becomes known as an *entry*. An entry is considered *committed* when it is impossible for it to be removed from the log, while an entry is *executed* once the command stored in the entry has been run. We only execute entries that are already committed.

2.4 Basics

2.4.1 Server States

Each replica is in one of three stages: *follower*, *candidate*, or *leader*. During normal operation, there is only one leader and all the other replicas are followers. The leader informs followers on what new entries to append to their log using a ReplicateEntries RPC. Followers are passive and do not issue requests on their own (except for Ben-Or+ requests). Candidates are replicas attempting to become the leader. No matter which of the three stages a replica is in, it can simultaneously run Ben-Or+ and commit entries even without acknowledgment from the leader.

2.4.2 Location of Entries

Entries on a replica are stored in two places: the log or the priority queue. Only the leader can add new entries to the log (unless the entry is committed using Ben-Or+). New entries received by non-leader replicas instead added to its min priority queue, which is sorted by timestamp.

2.4.3 Commit Index

Each replica has its own commitIndex, which is the highest index in its own log that the replica knows has been committed. Hydrofoil ensures that an entry is only committed if all previous entries in its log are committed, and that committed entries are never changed in any replica. Consequently, once an entry has been committed and all previously committed entries have been executed, the newly committed entry can also be safely executed.

2.5 Raft+

2.5.1 Term

Like in Raft, each replica has its own term. In each term, at most one replica can be the leader. A replica will reject all Raft+ RPCs that have a lower term. If a replica ever receives a message with a higher term, then it will always return to being a follower.

2.5.2 LeaderTerm

Instead of assigning each entry a term like in Raft, Raft+ introduces a new concept called a leaderTerm. The leaderTerm of a leader is always equal to its own term. When a replica receives a new message with a higher leaderTerm, then it knows to update its own log, and increase its own leaderTerm to that of the RPC sender. In addition, a replica will refuse to update its own log when it receives a new message with a lower leaderTerm and equal or lower commitIndex, even if this message comes from a sender with a higher term.

This property is not strictly necessary to the idea of Hydrofoil, and can be applied to Raft. In particular, using leaderTerms in Raft would eliminate the possibility of having an entry replicated on a majority of replicas without being committed due to having a lower term. This is because replicas will no longer accept the message from an older leader once it has received a replicate entries from a new leader, even if the new leader hasn't received any new entries in its new term. As a consequence, this also means that the leader can commit entries with lower terms, as long as a majority have responded positively to its ReplicateEntries RPC request.

To see this more clearly, we illustrate the case below to highlight the difference.

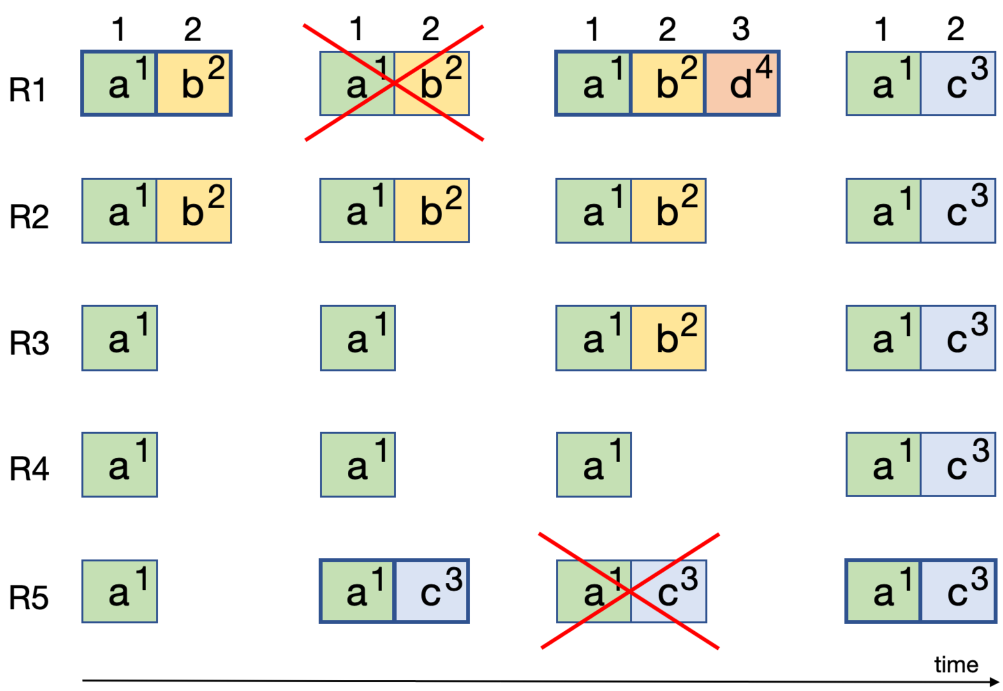

Figure 2.1: Raft with terms stored on entries themselves. Each letter represents a unique entry. Terms are located in the upper right corner of each entry.

Figure 2.1 shows a possible sequence of logs for replicas running Raft. R1 is originally the leader, but when it fails, R5 becomes the leader. Later, R5 fails, and R1 becomes the leader again. It broadcasts entry b to a majority of replicas, but it can't commit it yet, because if it fails, R5 can get a quorum of votes from R3 and R4, since the last term in the log of R3 and R4 is only 2.

Figure 2.2 shows result of the same scenario for replicas running Raft+. In this case, upon receiving a ReplicateEntries RPC from R1 in term 4, R2 and R3 know to set their leaderTerm equal to 4. Since R5 has a lower leaderTerm, it won't be able to become the new leader.

LeaderTerm is useful in Hydrofoil because replicas can broadcast entries even if they're not the leader. Upon receiving such a broadcast, a replica can compare leaderTerms to unambiguously decide if it should update its own log. For example, in the scenario above, imagine that R5 with entry c is broadcasting its log to R3 with entry

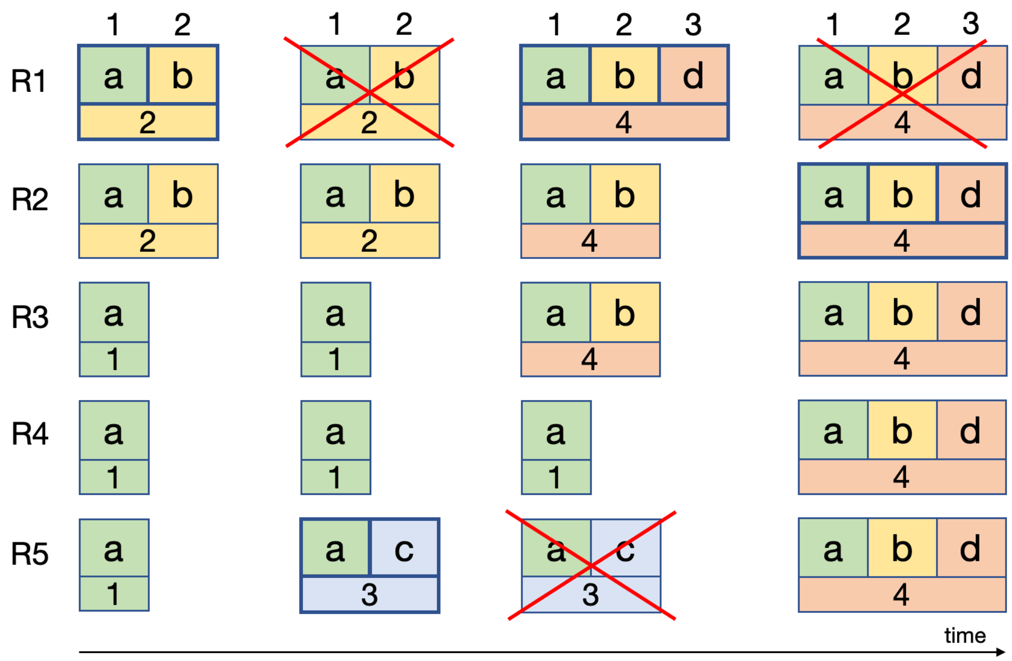

Figure 2.2: Raft+ using leaderTerms. The leaderTerm for each log is written below the log entries.

b. It's not clear whether R3 should be updating its log. With a leaderTerm, however, it's clear R3 shouldn't update its log, because R3 would have a leaderTerm of 4 while R5 only has a leaderTerm of 3.

2.5.3 Electing a Leader

Electing a leader is also quite similar to the equivalent procedure in Raft. However, instead of using the term of the last entry in the log, we use the leaderTerm instead. Therefore, a replica grants its vote to a candidate if the candidate has a higher leaderTerm or has the same leaderTerm but a longer log, and the replica hasn't voted for another replica in the same term. Upon becoming a leader, a replica clears its priority queue and appends these removed entries, sorted by timestamped, to the end of its log.

2.5.4 Replicating Entries

The processor of replicating entries from the leader is largely the same as in Raft. The leader maintains a record of the next indices of its log to send to each replica. Upon receiving a ReplicateEntries RPC, a replica can successfully replicate the log if the starting index for these new entries is less than its own commitIndex. If not, it can still replicate the log if it has the same leaderTerm, and its log matches with a previous log entry sent by the leader. If the replica can't update its log, it asks the leader to send another heartbeat with the log entries starting from the replica's commitIndex. If the leader knows that an entry has been replicated on a majority ($\geq f+1$) replicas, then the leader typically knows that entry is committed. There are additional constraints if the leader is running Ben-Or+ that we explain later.

2.6 Ben-Or+

Ben-Or+ commits entries one at a time, and runs on the first non-committed index, i.e. commitIndex + 1. We call this the benOrIndex. It consists of two components: Ben-Or Broadcast and Ben-Or Consensus, organized according to the structure below.

- Ben-Or+
  - Ben-Or Broadcast
  - Ben-Or Consensus
    - Stage 1
    - Stage 2

2.6.1 Ben-Or Broadcast

During Ben-Or Broadcast, a replica chooses an entry from its log, or if the log is empty, the lowest timestamped entry from its priority queue. The replica then broadcasts this entry to all other replicas using a BenOrBroadcast notification.

Once a replica has received at least $f+1$ BenOrBroadcast notifications, it checks if $\geq f+1$ of the broadcasted entries are the same. If any are, then it sets its initial vote set to 1. Otherwise, it sets its initial vote to 0. Then, it proceeds to Ben-Or Consensus.

2.6.2 Ben-Or Consensus

Ben-Or Consensus is similar to the original Ben-Or algorithm. Replicas proceed through a number of phases until either the value 0 or 1 is decided. Each phase itself consists of two stages. During each stage, the replicas broadcast votes to each other.

In Stage 1, if a replica receives at least $f+1$ messages with the same vote v, then its vote in Stage 2 is v. Otherwise, its vote is ?. In Stage 2, if a replica receives at least $f+1$ of the same non-? vote v, then it decides on v and Ben-Or+ terminates. If it received at least one non-? vote v, then it increments its phase, and restarts Ben-Or Consensus in Stage 1 with initial vote v. If it received only ? votes, it increments its phase, and restarts Ben-Or Consensus with the initial vote determined by a coin flip.

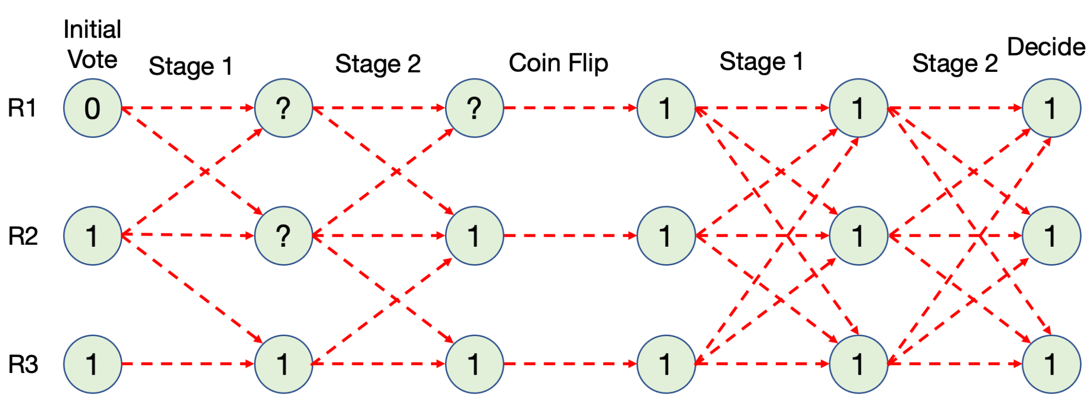

Figure 2.3: One iteration of Ben-Or Consensus.

However, we don't want to just decide on 0s or 1s, and instead want to commit actual entries. To do so, we instead commit the majority entry that we got from Ben-Or Broadcast when Ben-Or would normally decide on value 1. Furthermore, we restart Ben-Or+ on a new iteration when Ben-Or would normally decide on value 0, in hopes that this time, it will actually commit an entry. The procedure is outlined in the diagram below.

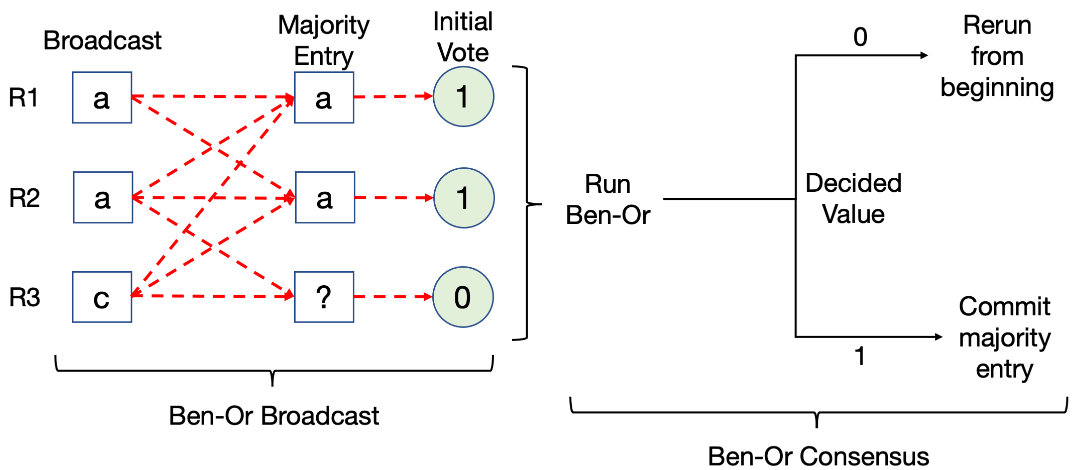

Figure 2.4: Breakdown of the Ben-Or+ protocol.

2.7 Combining the Protocols

2.7.1 Conflicting Entries from Raft+ and Ben-Or+

Purely allowing both Ben-Or+ and Raft+ to commit entries will lead to conflicts and a violation of linearizability. Thus, we need to enforce certain constraints on when replicas can commit entries using either of the two methods. In particular, the protocol must satisfy the two requirements below.

1. Safety: If the leader's ReplicateEntries RPC is successfully returned by at least f other replicas, then the leader should be able to commit and execute right away if it isn't running Ben-Or+. Consequently, if Ben-Or+ runs to completion after the leader has committed, then Ben-Or+ must commit the same entry as in the leader's log.

2. Liveness: If the leader's ReplicateEntries RPC doesn't reach all $f+1$ replicas, then the replicas must be able to reach consensus using Ben-Or+ without assistance from the leader.

The procedure to achieve this requires a number of modifications to the existing protocol. First, when a replica begins Ben-Or+, if it has an entry in its log at the index it's trying to commit at, then it will use a biased coin for this iteration of Ben-Or+. A biased coin always flips to 0.

Secondly, the procedure for deciding the initial vote is changed. When a replica begins Ben-Or Consensus, if it doesn't know the majority entry that was broadcast, then it still sets its initial vote to 0. However, if it knows the majority entry, then it checks to see if the entry it personally broadcasts matches the entry that's currently in its log at benOrIndex (recall that benOrIndex is always commitIndex + 1). If there is a match, then its initial vote is set to 1. Otherwise, its initial vote is set to 0.

When a replica receives a ReplicateEntries RPC from the leader, it checks if it can replicates the leader's log as usual. If it cannot because the previous log entry sent by the leader doesn't match with what is in its own log, then it responds negatively. If it can replicate the leader's log, then it will respond back positively if any of the following conditions are met.

- It is not running Ben-Or+.
- It is in the Ben-Or Broadcast stage.
- It is in the Ben-Or Consensus stage and the entry it broadcast while in the Ben-Or Broadcast stage is the same as the entry the leader is trying to get it to replicate.

Otherwise, the replica then responds back negatively, even if it actually replicated the leader's log.

Finally, upon receiving back a quorum of successful ReplicateEntries RPC responses, a leader can only commit if it is not currently running Ben-Or+, or the first entry it's trying to commit is equal to the majority entry for this iteration of Ben-Or.

As we will show later, this guarantees if a majority of replicas have received the leader's message, then when Ben-Or+ terminates, it will either commit the entry sent by the leader, or it will either terminate on a value of 0, meaning it doesn't commit an entry. In the latter case, when Ben-Or+ restarts, because the leader's entry is now in its log and Ben-Or+ always selects the broadcast entry from its log first, it will always commit on the leader's entry.

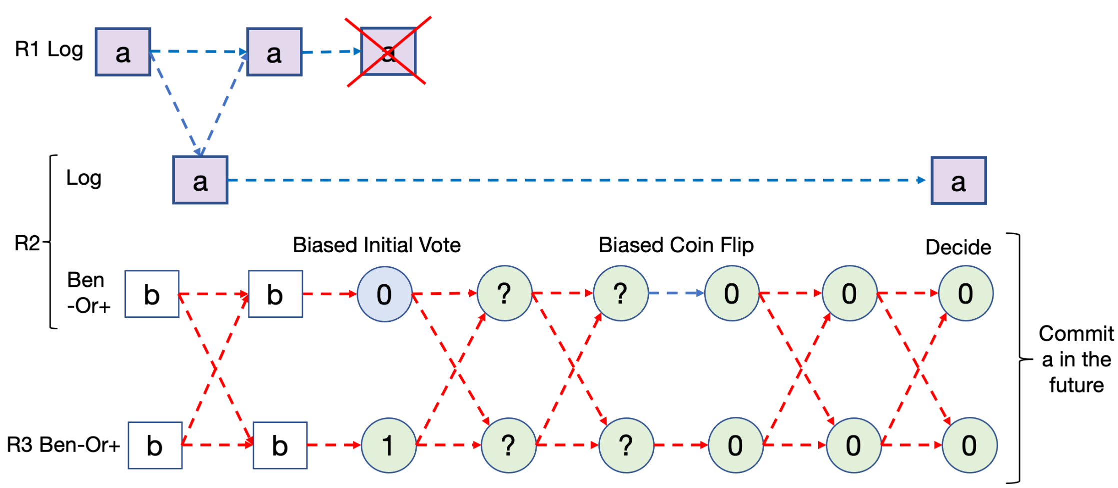

Figure 2.5: An example of how we use the initial vote and biased coins to manipulate the results of Ben-Or+ so it commits on entry a.

In Figure 2.5, we showcase an example of how we combine the two protocols. Here, R1 is the leader and is trying to commit entry a, and sends out a ReplicateEntries RPC (blue line) to the other replicas, but its request to R3 is lost. After this, but before receiving the request from the leader, R2 and R3 begin Ben-Or+ and both broadcast the entry b. After broadcasting, R2 receives the ReplicateEntries request from R1, updates its log, and responds back positively. At some point, R1 receives this response, commits a, and then fails. In order for Hydrofoil to be safe, both R2 and R3 must commit entry a, and not to commit the majority entry of b from the current iteration. To achieve this, R2 sets its initial vote to 0, even though it knows the majority entry. In addition, when it needs to perform a coin flip, it always flips to 0. This behaviour guarantees this iteration will decide on value 0. Since replicas attach parts of their log to Ben-Or+ messages, R3 will eventually learn of the existence of entry a and add it into its own log. Therefore, future iterations can only commit entry a, since it will be the majority entry.

2.7.2 Handling the Movement of Entries

In this protocol, entries can be found in two places: the log and the priority queue. We want to prevent the same entry from appearing in the log in multiple places. To do so, we must maintain a data structure to keep track of if the element is in the log, and or in the priority queue. Whenever we remove an entry from the log, we move it into the other priority queue. This means that, unlike Raft, we never delete entries.

To keep track of if the entry is in the priority queue, we can simply maintain a set since this priority queue won't infinitely grow. For the log, we can instead keep a two lists of length n, where n is the number of servers. The first list (called lastTimeInLog) stores the highest timestamps per server id seen across all entries, while the second list (called lastTimeCommitted) stores the highest timestamp per server id up to the current known commitIndex. we never add an entry to the log if it's timestamp is lower than the corresponding highest timestamp previously seen for that replica in lastTimeInLog. lastTimeCommitted is used to reset lastTimeInLog whenever we have to delete entries in our log.

2.8 Optimizations

2.8.1 Bringing Replicas Up To Date in Ben-Or+

Since Ben-Or+ only needs $f + 1$ replicas to proceed and make progress, it is possible for the other replicas to lag behind and fall out of date. A naive solution to bring these replicas up to date is for each replica to store all previous Ben-Or+ information, and to respond to out of date replicas with what they broadcast in the corresponding iteration, phase, and stage in the past. This, however, is memory intensive and slow, since it requires outdated replicas to still proceed one stage at a time. A much better solution is to skip to the newest iteration, phase, and stage.

Upon receiving a message with a higher iteration, a replica should cancel its current iteration of Ben-Or+, and select a new entry to broadcast from its log or priority queue. If the message has a higher phase, or the same phase and a higher stage, then the sender replica is currently running Ben-Or Consensus. The receiver replica should start running at the same phase and stage iteration, and set its vote to be the sender's vote. This is allowed because the receiver replica could've received all the same messages as the sender, which would've resulted in it going through the same state transitions and having the same vote as the sender currently has.

The one exception is if the receiver is using a biased coin, and receives a message from a sender that's in the same iteration, in a higher phase, and in Stage 1. In this case, the receiver cannot simply set its vote to be that of the sender if the sender's vote is 1, because it's possible the sender made a coin flip at the end of the previous phase that resulted in its current vote of 1, while the receiver will always coin flip a value of 0. To alleviate this, replicas must keep track of whether or not they made a coin flip in the previous iteration, and send this with all their Ben-Or+ related messages. If a receiver with a biased coin gets such a Stage 1 message from a sender that just performed a coin flip, then it will always set its vote to 0, no matter the vote of the sender.

2.8.2 Reissuing Commands

For each entry, we include an additional field that's the indices to live, shortened to ITL. This is basically an indication of how many log indices in the future before the entry commits, after which point we will no longer execute it. This field can be user specified. In Hydrofoil, we enforce this constraint by simply committing the entry as normal. During execution, we check this field, if we are past this ITL index, then we don't execute the command. Effectively, this is a means for the client to know when they can reissue the command, and not have to wait forever for an entry to return.

Chapter 3

Pseudo Code

3.1 Class Structure and Fields

3.1.1 Notation

For simplicity of notation, when unambiguous, RPCs are denoted as `rpc`. Any variable that starts with `rpc`, such as `rpc.term`, refers to the corresponding property/variable within the RPC. Notifications are denoted as `notif` and the notification fields follow the same notation as RPCs. Unless otherwise specified, any variable that isn't referring to a property within an RPC, notification, or message, such as `term`, is instead referring to a field on the replica. Local variables are declared using a `var` keyword.

3.1.2 Replica States

General Fields: Fields used by all replicas	
`serverState`	`Follower`, `Candidate`, or `Leader`
`id`	ID of the replica; distinct for each replica
`term`	term of a replica
`votedFor`	replica that was voted for in the current term
`log[]`	replica's local log; each entry contains the command for the state machine, server that first received this entry, and timestamp when it was received
`pq`	priority queue that stores entries not in the log; entries are sorted in ascending order first by timestamp, then by serverId; this is a min heap
`lastTimeInLog[]`	most recent timestamp out of all entries in the log for a given `serverId`
`lastTimeCommitted[]`	most recent timestamp out of all entries in the log up to `commitIndex` for a given `serverId`
`commitIndex`	index of highest log entry known to be committed
`leaderTerm`	highest term seen from a leader (directly or indirectly)
`lastApplied`	index of last log entry applied to state machine

Leader Fields: Additional fields used by the leader	
`nextIndex[]`	index of the next entry in log to send to each replica
`matchIndex[]`	index of the highest known replicated entry on each replica

Ben-Or+ Fields: Relevant fields to Ben-Or+; found on every replica	
`benOrRunning`	whether or not Ben-Or+ is running on this replica
`iteration`	current iteration of Ben-Or+
`phase`	current phase of Ben-Or+
`stage`	current stage of Ben-Or+; either `Broadcasting`, `Stage1`, or `Stage2`
`broadcastEntry`	entry broadcasted through Ben-Or+ this iteration
`broadcastsReceived[]`	all broadcasted entries received this iteration
`majEntry`	majority Ben-Or+ entry for this iteration (if known)
`vote`	the replica's vote when running Ben-Or Consensus; either 0, 1 for Stage1, and either 0, 1, or ? for Stage2
`votesReceived[]`	all consensus votes received for this particular iteration, phase, and stage
`prevPhaseFinalValue`	the resulting value from all the `votesReceived` at the end of the previous phase; if we did a coin flip last phase, this value is ?, otherwise it is our initial value this phase
`biasedCoin`	whether or not to use a coin that always flips to 0 in Ben-Or Consensus

Configuration Parameters: Configurable parameters of the system	
`electionTimeout`	time without hearing from the leader before a replica becomes a candidate and initializes an election
`hearbeatTimeout`	time interval between when the leader sends a heartbeat
`benOrStartTimeout`	time without hearing from the leader before a replica decides to start Ben-Or+ on its earliest non-committed entry; should be larger than `heartbeatTimeout`
`benOrResendTimeout`	time that a Ben-Or+ replica waits before it resends its Ben-Or+ message
`benOrWaitTimeout`	additional timeout after receiving $f + 1$ messages before the replica begins the next stage in Ben-Or+ in case more messages arrive
`idleTimeout`	timeout before sending a GetCommittedData RPC to bring replica up to date

In reality, to ensure the system remains live, the above timeouts should be randomized like they are in Raft. A good baseline is to treat the parameters above as an upper bound. Then, each time the timer is set, the timeout is chosen to be a uniformly random value between half the upper limit and the upper limit.

3.1.3 RPC Structures

Common Fields of all RPCs: Fields used in the request/response of all RPCs	
Request:	
`senderId`	ID of the sender
`senderTerm`	sender's term
`senderCommitIndex`	sender's `commitIndex`
`senderLeaderTerm`	sender's `leaderTerm`
`senderLogLength`	length of the sender's log
Response:	
`receiverId`	ID of the receiver
`receiverTerm`	receiver's term
`receiverCommitIndex`	receiver's `commitIndex`
`receiverLeaderTerm`	receiver's `leaderTerm`
`receiverLogLength`	length of the receiver's log
`receiverEntries[]`	entries in the receiver's log starting from `senderCommitIndex` + 1
`pqEntries[]`	entries in the receiver's priority queue in the form of a sorted list

The following RPCs contain all the fields above, in addition to the additional fields described below.

ReplicateEntries RPC: Invoked by leader to replicate log entries; also used as heartbeat	
Request:	
`prevLogIndex`	index of log entry immediately preceeding new ones
`prevLogEntry`	entry at position prevLogIndex in leader's log
`entries[]`	log entries to store
Response:	
`newRequestedIndex`	the new value the leader should start sending over from

RequestVote RPC: Invoked by candidates to gather votes	
Request:	
No additional fields	
Response:	
voteGranted	whether or not the receiver granted the candidate the vote

3.1.4 Notification Structures

Unlike RPCs, notification messages do not follow a request/response paradigm. They are a one-way signal from one replica to another; the other side is not obligated to send a reply/response. The sender refers to the replica sending the message, and the receiver refers to the replica receiving the message, no matter what type of message this is.

Common Fields of all Notifications: Fields in every notification message	
`id`	ID of the sender
`term`	sender's `term`
`commitIndex`	sender's `commitIndex`
`leaderTerm`	sender's `leaderTerm`
`logLength`	length of the sender's log
`startIdx`	starting index of entries (for a request notification, it is equal to `commitIndex + 1`, while for a response notification, it is equal to `reqNotif.commitIndex + 1`, where `reqNotif` is the corresponding request notification)
`entries[]`	log entries to update receiver's log with
`pqEntries[]`	entries within the sender's priority queue in the form of a sorted list

BenOrBroadcast Notification: Information broadcast when starting Ben-Or Broadcast	
`iteration`	iteration of Ben-Or+ at the sender
`broadcastEntry`	entry the sender is trying to commit during this iteration of Ben-Or+

BenOrBroadcastResponse Notification: Response sent by a replica in the Ben-Or Broadcast stage after receiving either a BenOrBroadcast or BenOrConsensus notification	
`msgValid`	whether or not the message sender is responding with a non-nil broadcastEntry; if the message sender has `benOrRunning` set to false, then `msgValid` is false
`iteration`	iteration of Ben-Or+ at the sender
`broadcastEntry`	entry the sender is trying to commit during this iteration of Ben-Or+

BenOrConsensus Notification: Information sent around in Ben-Or Consensus	
`iteration`	iteration of Ben-Or+ at the sender
`phase`	phase of Ben-Or+ at the sender
`stage`	stage of Ben-Or+ at the sender
`vote`	sender's vote for Ben-Or Consensus
`majEntry`	majority entry from Ben-Or Broadcast, if known
`prevPhaseFinalValue`	resulting value from all the `votesReceived` at the end of the last phase

BenOrConsensusResponse Notification: Response sent by a replica in the Ben-Or Consensus stage after receiving either a BenOrBroadcast or BenOrConsensus notification	
`msgValid`	whether or not the message sender is responding with a non-nil vote; if the message sender has `benOrRunning` set to false, then `msgValid` is false
`iteration`	iteration of Ben-Or+ at the sender
`phase`	phase of Ben-Or+ at the sender
`stage`	stage of Ben-Or+ at the sender
`vote`	sender's vote for Ben-Or Consensus
`majEntry`	majority entry from Ben-Or Broadcast, if known
`prevPhaseFinalValue`	resulting value from all the `votesReceived` at the end of the last phase

GetCommittedData Notification: Used to broadcast new entries or to get up to date
No additional fields

GetCommittedDataResponse Notification: Response sent by a replica if it is more up to date than the sender of a GetCommittedData notification
No additional fields

3.2 Code Execution

3.2.1 Ordinary Behaviour

Algorithm 1: Rules for All Replicas

```
if lastApplied < commitIndex then
    Apply log[lastApplied] to state machine
    Increment lastApplied
if receive message msg with term < msg.term then
    Set term ← msg.term
    Set votedFor ← −1
    Set serverState ← Follower                    // convert to follower
if receive a request command from a client then
    var t ← current time on this replica
    if serverState is leader then
        Append (command,t) to the end of log
        replicateEntries()
    else
        Push (command,t) to pq
        Broadcast GetCommittedData notification
```

Algorithm 2: Check if replica's log or the log from the RPC/message is more up to date

```
function moreUpToDate(msg):
    if commitIndex ≠ msg.commitIndex then
        return boolToReplica(commitIndex > msg.commitIndex)
    else if leaderTerm ≠ msg.leaderTerm then
        return boolToReplica(leaderTerm > msg.leaderTerm)
    else if len(log) ≠ msg.logLength then
        return boolToReplica(len(log) > msg.logLength)
    else
        return Equal
function boolToReplica(status):
    if status = True then
        return Replica
    else
        return Incoming
```

Algorithm 3: Updates the log using the incoming message

```
function updateLog(msg, startIndex):
    if an existing entry e with index i such that startIdx ≤ i ≤
      msg.commitIndex in log conflicts with msg.entries (different serverId
      or timestamp) then
        Delete existing entry and all that follow it
        Replace with the log with msg.entries
    else if leaderTerm > msg.leaderTerm or
      (leaderTerm = msg.leaderTerm and len(log) ≥ msg.logLength) then
        Don't replace existing log
    else if an existing entry after msg.commitIndex conflicts then
        Delete existing entry and all that follow it
        Replace with the log with msg.entries
    Update commitIndex and logTerm
    if commitIndex changed then
        Stop Ben-Or+ and reset fields related to it
    if serverState ≠ Leader then
        Add all new and deleted entries not in log or pq into pq
    else
        Sort all new and deleted entries not in log from msg.entries and
          msg.pqEntries together by timestamp
        Append these new entries onto the end of log
        If an entry in the log was changed within this function, then set
          matchIndex[i] ← min(matchIndex[i], commitIndex) ∀ replicas i
    Update lastTimeInLog and lastTimeCommitted

function getNewValues(msg):
    if serverState ≠ Leader then
        Add all previously unseen entries from msg.entries and
          msg.pqEntries into pq
    else
        Sort all unseen entries from msg.entries and msg.pqEntries together
          by timestamp
        Append these new entries onto the end of log
        Update matchIndex and nextIndex for leader
    Update lastTimeInLog and lastTimeCommitted
```

Algorithm 4: Rules for Followers

```
if electionTimeOut elapses without receiving ReplicateEntries RPC from
  current leader (term ≤ msg.term) or granting vote to candidate then
    serverState ← Candidate                    // convert to candidate
```

3.2.2 Electing a Leader

Algorithm 5: Rules for Candidates

```
upon conversion to candidate then
    startElection()
if receive ReplicateEntries from the new leader (rpc.term ≥ currentTerm)
  then
    serverState ← Follower                      // convert to follower
if electionTimeOut elapses without becoming leader or receiving
  ReplicateEntries RPC from current leader then
    startElection()
```

Algorithm 6: Start Election

```
function startElection():
    Increment currentTerm
    Set votedFor ← id                                  // vote for self
    Send RequestVote RPCs to all other replicas
    upon receive RequestVote RPC response then
        if moreUpToDate(rpc) = Incoming then
            updateLog(rpc, commitIndex+1)
        else
            getNewValues(rpc)
    if receive votes from a majority of replicas (≥ f RPCs return with
      rpc.votedFor = True) then
        serverState ← Leader                          // convert to leader
```

Algorithm 7: Handle RequestVote RPC

```
function handleRequestVote(rpc):
    if term < rpc.currentTerm then
        rpc.voteGranted ← False
    else if (rpc.votedFor = -1 or rpc.senderId) and
      (leaderTerm > rpc.leaderTerm or
      (leaderTerm = msg.leaderTerm and len(log) ≥ msg.logLength))
      then
        votedFor ← rpc.senderId
        rpc.voteGranted ← False
    else
        rpc.voteGranted ← True
```

3.2.3 Replicating Entries

Algorithm 8: Rules for Leaders

```
upon being elected as leader then
    Pop all entries from pq and append to end of log
    Set matchIndex for all replicas to 0
    Set nextIndex to len(log)+1
    replicateEntries()
if heartbeatTimeout elapses without having sent ReplicateEntries RPC then
    replicateEntries()
if ∃ k such that k > commitIndex and a majority of matchIndex ≥ k and
  (benOrRunning = False or majEntry = log[commitIndex+1]) then
    Set commitIndex ← k
    Stop Ben-Or+ and reset fields related to it
```

Algorithm 9: Handle ReplicateEntries RPC

```
function handleReplicateEntries(rpc):
    if term > rpc.senderTerm or moreUpToDate(rpc) = Replica then
        rpc.newRequestedIndex ← commitIndex+1
        return
    serverState ← Follower
    if (leaderTerm ≠ rpc.leaderTerm and log has an entry at
     rpc.prevLogIndex that matches prevLogEntry) or rpc.prevLogIndex ≤
     commitIndex then
        updateLog(rpc, rpc.prevLogIndex+1)
        if benOrRunning = False, benOrStage = Broadcasting, or
         log[oldCommitIndex] = broadcstEntry then
            rpc.newRequestedIndex ← len(log)
        else
            rpc.newRequestedIndex ← commitIndex+1
    else
        rpc.newRequestedIndex ← commitIndex+1
```

Algorithm 10: Replicate log entries on replicas

```
function replicateEntries():
    Broadcast ReplicateEntries RPCs
    upon receiving ReplicateEntries RPC response then
        if moreUpToDate(rpc) = Incoming then
            updateLog(rpc, max(commitIndex, rpc.prevLogIndex)+1)
        else
            getNewValues(rpc)
            nextIndex[rpc.receiverId] ← rpc.newRequestedIndex−1
            matchIndex[rpc.receiverId] ← rpc.newRequestedIndex
```

3.2.4 Executing Ben-Or+

Algorithm 11: Ben-Or+ related rules

```
if serverState ≠ Leader, and benOrRunning = False then
    if benOrStartTimeout elapses without receiving a ReplicateEntries RPC
      from current leader, and the replica contains uncommitted entries in its
      log or pq then
        startBenOrBroadcast()
    else if idleTimeout elapses since ending Ben-Or+ or last broadcasting out
      a GetCommittedData RPC to all replicas then
        Broadcast GetCommittedData notification
if benOrRunning = True, and benOrResendTimeout elapses since last
  broadcasting BenOrBroadcast/BenOrConsensus notification without
  iteration, phase, or stage changing then
    if benOrStage = Broadcasting then
        Broadcast BenOrBroadcast notification
    else
        Broadcast BenOrConsensus notification
```

3.2.5 Ben-Or Broadcast

Algorithm 12: Select Entry to Broadcast

```
function selectBroadcastEntry():
    // Only called if there are uncommitted entries in the log or
       pq
    if log contains an entry at benOrIndex then
        return log[benOrIndex]
    else
        return pq.peek()
```

Algorithm 13: Ben-Or+ Broadcast Stage

```
function startBenOrBroadcast():
    broadcastEntry ← selectBroadcastEntry()
    benOrRunning ← True
    phase ← 0
    stage ← Broadcasting
    Add broadcastEntry to broadcastsReceived
    Broadcast BenOrBroadcast notifications to all other replicas
    Wait for f BenOrBroadcast/BenOrBroadcastResponse notifications;
      beyond f responses, wait for an additional benOrWaitTimeout
    if entry e appears ≥ f + 1 times in broadcastsReceived then
        majEntry ← e
        if log contains an entry at index commitIndex + 1 that's different
          from broadcastEntry then
            vote ← 0
        else
            vote ← 1
    else
        vote ← 0
    if log has an entry at benOrIndex then
        biasedCoin ← True
    else
        biasedCoin ← False
    startBenOrConsensusStage1()
```

Algorithm 14: Update the log if we're not too far out of date

```
function updateLogIfPossible(notif):
    if moreUpToDate(notif) = Incoming then
        if commitIndex+1 ≥ notif.startIndex then
            updateLog(rpc, commitIndex+1)
            return True
        else
            return False
    else
        getNewValues(notif)
        return True
```

Algorithm 15: Respond to Ben-Or+ Notification Request

```
function respondToNotification(destServerId):
    if benOrRunning = False or benOrStage = Broadcasting then
        Send BenOrBroadcastReply notification notif to destServerId with
          notif.msgValid ← benOrRunning
    else
        Send BenOrConsensusReply notification to destServerId
```

Algorithm 16: Handle BenOrBroadcast/BenOrBroadcastResponse Notification

```
function handleBenOrBroadcast(notif):
    var updated ← updateLogIfPossible(notif)
    if updated = False then
        Send GetCommittedData notification to notif.id
        return without sending response
    if notif is a BenOrBroadcastResponse notification and notif.msgValid
      = False then
        return without sending response
    if commitIndex = notif.commitIndex then
        if iteration < notif.iteration then
            Cancel current iteration of Ben-Or+
            iteration ← notif.iteration
            startBenOrBroadcast()
        else if benOrRunning, iteration = notif.iteration, stage =
          Broadcasting, and we haven't received another such notification this
          iteration from notif.id then
            Add notif.broadcastEntry into broadcastsReceived
    respondToNotification(notif.id)
```

3.2.6 Ben-Or Consensus

Algorithm 17: Ben-Or+ Consensus Stage 1

```
function startBenOrConsensusStage1():
    benOrRunning = True
    stage ← Stage1
    Add vote to votesReceived
    Broadcast BenOrConsensus notifications to all other replicas
    Wait for at least f BenOrConsensus/BenOrConsensusResponse
      notifications for Stage 1; beyond f responses, wait for an additional
      BenOrWaitTimeout
    if value v appears ≥ f + 1 times in votesReceived then
        vote ← v
    else
        vote ← ?
    startBenOrConsensusStage2()
```

Algorithm 18: Ben-Or+ Consensus Stage 2

```
function startBenOrConsensusStage2():
    benOrRunning = True
    stage ← Stage2
    Add vote to votesReceived
    Broadcast BenOrConsensus notifications to all other replicas
    Wait for at least f BenOrConsensus/BenOrConsensusResponse
      notifications for Stage 2; beyond f responses, wait for an additional
      BenOrWaitTimeout
    if value 0 appears ≥ f + 1 times in votesReceived then
        Reset all Ben-Or+ related fields other than iteration
        iteration ← iteration + 1
        if serverState ≠ Leader then
            startBenOrBroadcast()
    else if value 1 appears ≥ f + 1 times in votesReceived then
        var newCommitIndex ← commitIndex+1
        if log[newCommitIndex] doesn't match majEntry then
            Remove all entries starting from newCommitIndex from log
            Set log[newCommitIndex] ← majEntry
            if serverState ≠ Leader then
                Add all deleted entries into pq
            else
                Sort all deleted entries by timestamp
                Append these new entries onto the end of log
                matchIndex[i] ← min(matchIndex[i], commitIndex) ∀
                  replicas i
            Update lastTimeInLog and lastTimeCommitted
        commitIndex ← newCommitIndex
        Stop Ben-Or+ and reset fields related to it
    else
        if a non-? value v appears in votesReceived then
            prevPhaseFinalValue ← v
            vote ← v
        else
            prevPhaseFinalValue ← ?
            vote ← coinFlip()
        phase ← phase+1
        startBenOrConsensusStage1()
```

Algorithm 19: Coin Flip that Returns 0 or 1

```
function coinFlip():
    if biasedCoin = true then
        return 0
    else
        return 0 or 1 with equal probability
    end
```

Algorithm 20: Update from consensus notification with a higher iteration

```
function consensusHigherIterationUpdate(notif):
    Cancel current iteration of Ben-Or+
    iteration, phase, stage, majEntry ← notif.iteration,
     notif.phase, notif.stage, notif.majEntry
    if log contains an entry at commitIndex + 1 then
        biasedCoin ← True
    else
        biasedCoin ← False
    if notif.stage = Stage1 then
        if notif.phase = 0 then
            vote ← (majEntry ≠ Nil)
        else
            if notif.prevPhaseFinalValue = ? then
                vote ← coinFlip()
            else
                vote ← notif.vote
    else
        vote ← notif.vote
    if stage = Stage1 then
        startBenOrConsensusStage1()
    else
        startBenOrConsensusStage2()
```

Algorithm 21: Update from consensus notification with a higher phase/stage

```
function consensusHigherPhaseStageUpdate(notif):
    iteration, phase, stage ← notif.iteration, notif.phase,
     notif.stage
    if notif.majEntry ≠ Nil then
        majEntry ← notif.majEntry
    if notif.stage = Stage1 then
        if notif.phase = 0 then
            if benOrRunning = True and broadcastEntry ≠ majEntry then
                vote ← 0
            else
                vote ← (majEntry ≠ Nil)
            if log contains an entry at commitIndex + 1 then
                biasedCoin ← True
            else
                biasedCoin ← False
        else
            if notif.prevPhaseFinalValue = ? then
                vote ← coinFlip()
            else
                vote ← notif.vote
    else
        vote ← notif.vote
    if stage = Stage1 then
        startBenOrConsensusStage1()
    else
        startBenOrConsensusStage2()
```

Algorithm 22: Handle BenOrConsensus/BenOrConsensusResponse Notification

```
function handleBenOrConsensus(notif):
    var updated ← updateLogIfPossible(notif)
    if updated = False then
        Send GetCommittedData notification to notif.id
        return without sending response
    if notif is a BenOrConsensusResponse notification and notif.msgValid
      = False then
        return without sending response
    if commitIndex = notif.commitIndex then
        if iteration < notif.iteration then
            consensusHigherIterationUpdate(notif)
        else if (iteration = notif.iteration and phase < notif.phase)
          or (iteration = notif.iteration, phase = notif.phase, and
          stage is earlier than notif.stage) then
            consensusHigherPhaseStageUpdate(notif)
        else if benOrRunning, iteration = notif.iteration, stage =
          Broadcasting, and we haven't received another such notification this
          iteration from notif.id then
            if notif.majEntry ≠ Nil then
                majEntry ← notif.majEntry
            Add notif.vote into votesReceived
    respondToNotification(notif.id)
```

Algorithm 23: Handle GetCommittedDataReply Notification

```
function handleGetCommittedDataReply(notif):
    if moreUpToDate(notif) = Incoming then
        updateLog(notif, commitIndex+1)
    else
        getNewValues(notif)
```

Algorithm 24: Handle GetCommittedData Notification

```
function handleGetCommittedData(notif):
    var updated ← updateLogIfPossible()
    if updated = False then
        Send GetCommittedData notification to notif.id
        return without sending response
    else
        Send back GetCommittedDataReply notification
```

Chapter 4

System Properties and Guarantees

4.1 Safety

We will now provide a proof sketch that Hydrofoil is safe, i.e. it satisfies linearizability. This is only a sketch and not a complete proof, because Lemma 4.1.2 has yet to be fully proved.

4.1.1 Total Order

Here, we prove that client commands submitted to Hydrofoil are committed in some total order.

On any particular replica r, each entry e can be committed at index i in one of three ways. The first is a Raft+ commit, the second is a Ben-Or+ commit, and the last method is if another replica with a higher commitIndex sends a message to r, and r updates its log (and commitIndex) using updateLog. We denote the three commit methods as leader-based, leaderless, and update commits respectively.

Lemma 4.1.1. *For a replica r, once an entry e at index i in the log satisfies $i \leq r.commitIndex$, then e will always remain at index i in r's log.*

Proof. This is a consequence of how log updates happen. There are only two occasions where a replica updates its log: in the updateLog function, and during a leaderless commit. In Hydrofoil, each time we call updateLog, we start the log replacement process at an index $\geq$ commitIndex+1. Similarly, a leaderless commit can only change index commitIndex+1. Since the commitIndex never decreases, any entry at index $\leq$ commitIndex will never be changed. □

Lemma 4.1.2. *If two replicas with the same leaderTerm have the same entry in their log at index i, and there have been no safety violations (i.e. no two replicas have committed different entries at the same index), then they have the same entries at all indices $\leq i$.*

Proof. We will actually prove a stronger result. In particular, we will prove that the above property holds for not just any two replicas with the same leaderTerm in real-time, but for all logs in history that have had the same leaderTerm. Denote this stronger property as the Log Equality Property.

We will use proof by contradiction. Let t be the first leaderTerm where the logs of two replicas don't satisfy this property. Consider the first moment in real time where any two replicas r_1 and r_2 no longer satisfy this property, and assume that r_2 is the replica that changed its log at this moment. Note that r_1's log could be an older version of its log, since we're proving the stronger version of this property. At that point in time, r_1 had leaderTerm t.

There are three possibilities for why r_2 changed its log: it committed an entry using Ben-Or+, it received a message from another replica, or it received a message from the leader of term t. In the first case, r_2 must already have a leaderTerm of t. Let e_b be the entry committed by Ben-Or+. Let i_b be r_2's benOrIndex before it committed e_b. If r_2 didn't originally have an entry in its log at i_b, then appending e_b does not violate this property. If r_2 did originally have an entry at i_b, then let that entry be e. If $e = e_b$, then the log of r_2 does not change, so the property still holds. If $e \neq e_b$, then r_2 clears its log of all the entries starting from i_b, and appends e_b. Now, the only entry that r_1 and r_2 could have that is different (meaning both logs are long enough to contain this index and the entry at that index is different) is e_b, but since it's at the end of r_2's log, this property still holds.

The second possibility is that r_2 received a message from another replica r_3 that caused it to update its log. Note by our assumption, r_3 must have had leaderTerm $\leq t$ when it sent this message or else r_2 would update its leaderTerm to be $> t$ if it also updated its log.

If r_3 has leaderTerm $< t$, then for r_2 to have leaderTerm t after receiving this message, it must have already had leaderTerm t before receiving this message. r_2 would only update its log (and leaderTerm) if r_3 has a higher commitIndex. Now, if r_3's log up to r_3.commitIndex is a prefix of r_2's log, then there would be no change to r_2's log, so the Equivalent Log Property still holds. If r_3 has a different entry before r_3.commitIndex, then we consider two subcases depending on if r_2 is the leader for term t.

If r_2 is not the leader for term t, then it would replace all entries in its log from r_2.commitIndex+1 to r_3.commitIndex with the entries from r_3's log, and remove any entries after. If r_2 is the leader for term t, then it will additionally append the entries removed from its own log with the remaining entries in r_3's log sorted in timestamp order.

Since there have been no conflict violations yet, this means that r_2's and r_3's log were identical (and remain identical after the update) up to r_2.commitIndex. This means we only need to show there are no violations of the Equivalent Log Property result from new entries at indicies $> r_2$.commitIndex. We do not yet have a proof of this fact. Our empirical tests have always found this to be the case, but this does not actually prove this result for this case.

If instead r_3 has leaderTerm t when it sent the message, then we first break down the case where r_2 has leaderTerm $< t$, or has leaderTerm t but is not the leader. r_2 will only potentially update its log if r_3 has a higher or equal commitIndex. r_2 will either keep its own log, or if there is a conflict, it will change its log to match that of r_3. r_2 will only keep its own log if r_3's log is a prefix of its own log. If r_2 already had leaderTerm t, then by our assumption that this is the first violation, the Log Equality Property held before r_2's log update. After the update, it could've only replaced parts of its log with new entries from replica r_3, which means the Log Equality Property still holds. If r_2 originally had a lower leaderTerm, then if it now has r_3's log, the property trivially still holds. If r_3's log is now a prefix of its own log, the remaining entries in its log either match with those in the log of some other replica r_4 with leaderTerm t, or not. If it doesn't match, then the property trivially still holds. If it does match, then r_2's log must be a prefix of r_4's log, so the property still holds.

Now if r_2 is the leader for term t, then it could append newly seen entries to the end of its log after receiving this message from r_3. To prove the Log Equality Property, we must show that these newly added entries will either be at new indices not previously in the log of any other replica with leaderTerm t, or they will not match. Once again, we do not have a proof of this hypothesis, although empirical evidence also suggests this.

The final possibility is that r_2 received a message from the leader. In particular, the only different case we need to consider is a ReplicateEntries RPC from the leader. The one additional time r_2 can update the log in such a case (as opposed to when we receive an ordinary message) is if it has the same leaderTerm and a previous log entry matches. In this case, since this property hasn't been violated before, all entries up to this previous log entry are the same between r_2 and the leader. Afterward, r_2 will either take on the leader's log, or the leader's log is a prefix of r_2's log. In either case, the property still holds.

Assuming we can prove the two remaining cases, we would have a contradiction and prove this lemma. ☐

Lemma 4.1.3. *If Ben-Or+ on a replica is to terminate on value 1, then that value knows the majority entry from the broadcast stage.*

Proof. We will use induction on the claim that any replica that receives a value 1 or ? knows what the majority value is.

First, it's obvious that only a single command can be decided on as the majority in Ben-Or Broadcast. Now consider phase 1. If any replica s receives a 1 or ? in Stage 2 from server s', then s' received a 1 in Stage 1. Since we always attach the majority entry to Ben-Or notifications, s' knows the majority entry, and so does s.

Since we keep passing around the majority entry with notifications in higher phases, by the same reasoning, we are done. ☐

Lemma 4.1.4. *Suppose that replica r makes a leaderless commit of an entry e at index i in iteration β. Then, no other replicas can make a leaderless commit of another entry e' at index i.*

Proof. Let's say some r' commits e' in iteration β'. This means that e was the majEntry ($\geq f+1$ replicas broadcast e) in iteration β, while e' was the majEntry in iteration β'. By Lemma 4.1.3, we know that all replicas will know the majEntry if they commit it. Since we only have $2f+1$ replicas, then by the pigeonhole principle it is impossible for e and e' to be the majEntry in the same iteration, so $\beta \neq \beta'$.

Without loss of generality, assume that $\beta < \beta'$. For e to be committed using the leaderless protocol, it must have received Stage 2 votes from at least $f+1$ replicas running Ben-Or+ for index i in iteration β'. By the pigeonhole principle, at least one of these replicas, which we denote r_b must have participated in the β iteration of Ben-Or+ for index i.

By the properties of Ben-Or, if one replica decides on a value 0 or 1, then all replicas participating in Ben-Or eventually decide on that vote. Since r committed e in iteration β, this means that r decided on value 1. Therefore, if r_b ran to completion using Ben-Or+, r_b also decided on value 1 and committed an entry at index i, and wouldn't participate in any future iterations of Ben-Or+ for index i.

If r_b did not run to completion using Ben-Or+, then it must have committed an entry at index i using either the leader-based or update commit protocol. In either case, since $i \leq r_b$.commitIndex now, it won't be participating in any higher iteration of Ben-Or+ for iteration i. □

Lemma 4.1.5. *Consider the commit of an index i. All replicas can have an initial log entry at i, however, we assume that all leaders with leaderTerm $\geq t$ will have entry e at i. Then, if there is a leader-based commit for a leader with leaderTerm $\geq t$, then Ben-Or+ will always commit on the same entry e.*

Proof. If the leader r committed e, then it received successful ReplicateEntries RPC responses from $\geq f$ replicas for an index i' that's $\geq i$. By Lemma 4.1.2, this means that e is at index i on all these $\geq f$ replicas. Now, the leader is either running Ben-Or+ or not. Let's first consider if it isn't running Ben-Or+ when it committed.

Let β be the highest iteration of Ben-Or+ currently being run for index i. If the leader's entry e was broadcast by a majority of replicas in Ben-Or+ consensus, then this iteration will either commit e or commit nothing and start a new iteration. But since all future leaders also have entry e at i, a majority of replicas will never broadcast anything other than e, so e will for sure be committed.

Now, consider the case when some other entry e' is broadcast by a majority of entries in iteration β. This means that at least one of the replicas that received e from the leader, broadcast e'. This means that this replica received the leader's message after it already broadcast e', and thus has biasedCoin set to true, and its initial vote set to 0.

Since there are $\leq f$ entries with vote $= 1$ in Stage 1, then the votes in Stage 2 are either all 0 or ?. At the end of Stage 2, either the replicas decide on 0, and thus nothing is committed this iteration, or it restarts Stage 1. But since biasedCoin is true for all replicas that have e, they will all start Stage 1 again with vote $= 0$, so we're at worst back essentially to the same situation. Therefore, this will eventually decide on committing nothing this iteration.

The next iteration, the majority of replicas have entry e in their log, so they will broadcast e. Therefore, if an entry is committed by Ben-Or+, it will be e.

Going back, let's now consider if the leader was running Ben-Or+ on iteration β_r when it made its leader-based commit. In order for it to commit using the leader-based protocol, it must be the case that $e =$ r.log[i] $= r$.majEntry. If iteration β_r ends up deciding on value 1 and committing, then it must commit e since it's the majority entry. In addition, the leaderless commit of e in iteration β_r is done without any involvement from the leader after r makes its leader-based commit, since r would no longer be participating in Ben-Or+ for iteration β_r as it has already committed up to index i. Then, we know from Lemma 4.1.4 that it's impossible for some replica to make a leaderless commit of a different entry e'.

If the entry isn't committed in iteration β_r, then the situation is effectively equivalent no matter if r committed e while it was still running Ben-Or+, or if it had simply stopped running and waited until this iteration β_r finished and r.benOrRunning $=$ False before it commits. The only difference is that in the former case, a few more replicas might have committed e a little earlier through the update protocol after receiving ReplicateEntries RPCs from r. This reason for this equivalence is due to the fact that r doesn't participate in the decision to commit nothing in iteration β_r, since it stops running Ben-Or+ for index i due to it already having been committed. In this case, if any replica makes a leaderless commit at index i, it will be e from the logic earlier (when we argued regarding the case if the leader wasn't running Ben-Or+ when it made the leader-based commit). □

Lemma 4.1.6. *Suppose that replica r makes a leader-based commit of an entry e at index i in its log in term t, where t is the smallest iteration for a leader-based commit. Then, if there do not exist total order violations up to index i, the following two properties hold:*

1. *All leaders with a term $t' > t$ will also have entry e at index i.*
2. *Leaderless commits for index i will only ever commit e.*

Proof. By Lemma 4.1.5, we know that if no leader with term $\geq t$ has a different entry than e at index i, then there will not be a leaderless commit for a non-e entry at index i. What remains is to prove that no elected leader with a higher term will have a different entry at i.

Let r_l is the leader with the smallest term that's still $> t$. Since r_l is a leader, then by construction r_l got back votes using RequestVote RPCs from $f + 1$ replicas, which we denote S'. Let S the set of replicas that responded positively to r's ReplicateEntries RPCs. By the pigeonhole principle, there exists at least one replica in both S and S', which we denote as the vote r_v. r_v must have received the RequestVote from r_l after the ReplicateEntries from r, because otherwise it would've rejected the ReplicateEntries RPC on the basis of having a higher term: its current term would be t_l, which is higher than t.

For r_l to have gotten a vote from r_v, it must have either had a higher leaderTerm, or the same leaderTerm and an equally long or longer log. If it had a higher leaderTerm, then some other leader must've first been elected with a term $> t$, which goes against our assumption. Thus, r_l also has leaderTerm t and has a log length that's $\geq$ to the log length of r. Given that it has leaderTerm t, then its log must be a prefix of r's log, unless it has made leaderless commits after it received and accepted ReplicateEntries from r. If it didn't make such a leaderless commit, then its log contains e at index i. If it did make such a leaderless commit, then let's say these leaderless commits after its last successful ReplicateEntries from r happened at indices $[i_s, i_e]$. Observe that all such leaderless commits from indices $[i_s, i - 1]$, committed the same entries as r did, since by assumption index i is the first index where there is a different entry committed across the replicas.

Let β be the highest iteration of Ben-Or+ at the time that r committed entry e. Assuming there is no other leader that causes any of these replicas in S to modify their log entry at i, then by Theorem 4.1.5, iteration β will either result in committing e, or it will select value 0 leading to a new iteration. Then by the same theorem, further iterations will eventually commit e. Since r_l is the first leader with a term $\geq t$, then at the time of becoming the new leader, if it's entry at index i is committed, the entry committed must've been e. Therefore, in all cases, upon becoming the leader, r_l has entry e in its log at index i.

Therefore, when r_l sends out ReplicateEntries RPCs, it will not cause any replica to to modify their log entry at i to be something other than e. Now, it is easy to see through induction that any future leader will have entry e at index i in its log, and so no leader will cause any replica in S to modify their entry at index i. Furthermore, applying Theorem 4.1.5 again, we can conclude that leaderless commits for index i will only ever commit e. □

Theorem 4.1.7. *Hydrofoil satisfies the total order property of linearizability.*

Proof. Since each replica executes commands based on their own linear log, and we execute commands in the order of commit, proving this property amounts to proving that each log commits entries in the same order. Specifically, in this context, when we say an entry e was committed on replica r, we mean that the index of e in replica r's log is $\leq r$.commitIndex. This is different from a entry being committed in general, which just means that this entry is for sure committed at a specific index. An entry can be committed at an index before any replica's commitIndex exceeds that index. Now we are ready to begin this proof.

We will prove this property by contradiction. Let index i be the first index where at least two different entries are committed, which we denote as e_1 and e_2.

By Lemma 4.1.1, committed entries do not change, so we have to only consider the three possibilities of committing a new entry. Note that the update commit method is only possible once e has been committed using one of the first two methods at index i at some other replica first. Therefore, without loss of generality, we only need to consider three possible cases. Consider any two replicas r_1 and r_2 that committed e_1 and e_2 at index i using either leader-based or leaderless methods.

Case 1: r_1 and r_2 make leaderless commits of e_1 and e_2.

By Lemma 4.1.4, this is impossible.

Case 2: r_1 makes a leader-based commit of e_1, while r_2 makes a leaderless commit of e_2.

Without loss of generality, assume that out of all different entries committed using the leader-based protocol at index i, e_1 was committed on r_1 in the smallest term.

By the second property of Lemma 4.1.6, we know that if r_1 made a leader-based commit, then r_2 cannot have committed a different entry e_2 using the leaderless protocol. Thus, this is impossible.

Case 3: r_1 and r_2 make leader-based commits of e_1 and e_2.

Let r_1 and r_2 commit e_1 and e_2 in term t_1 and t_2 respectively. Without loss of generality, suppose that $t_1 < t_2$. By the first property of Lemma 4.1.6, r_2 has entry e_1 in its log at index i. Therefore, it can only make a leader-based commit of e_1 at index i, so this is impossible.

All 3 cases are impossible, so there is a contradiction. This mans that all replicas commit (and consequently execute) in the same total order. Therefore, Hydrofoil satisfies the total order property of linearizability.

□

4.1.2 Real Time Order

Theorem 4.1.8. *Hydrofoil satisfies the real-time order property of linearizability.*

Proof. Suppose that command c_1 is executed and its corresponding response reaches a client a_1 before client a_2 sends its command c_2 to the system in real-time. Since response happens after execution, and execution happens after commit, this means that in real-time, command c_1 was committed on some replica r_1 before c_1 was sent as a request in real-time. Therefore, r_1 has an entry e_1 containing c_1 at an index $\leq r_1$.commitIndex. Denote this point $m \coloneqq r_1$.commitIndex.

By the total order property of Hydrofoil, we know that all entries up to $\leq m$ in r_1 will become committed entries on all replicas. Therefore, when r_2 is committed, it will be at a higher index in the log than m, and therefore will be executed after m in the total order (which is the same as the committed log order).

This proves that Hydrofoil satisfies the real-time order property of linearizability. □

4.2 Liveness

We do not have a thorough proof of liveness. However, since both Ben-Or and Raft are live in ordinary network conditions (with the use of techniques like exponential backoff), then Hydrofoil is also likely to be live under the same conditions.

4.2.1 Termination in Asynchronous Networks

Ben-Or is guaranteed to commit even in an adversarial network with no message drops, as long as we do a random coin flip with non zero probabilities of both options (even if the probabilities of heads/tails are not the same) [1].

However, this is not true for Ben-Or+. As long as potentially even one server has a totally biased coin, there is no termination guarantee for Ben-Or+.

To see such a case, consider when we have five servers, all of which are working. One of them is biased and is biased to 0, and its initial vote is 0. The other four servers are not biased. Replicas are numbered 1 to 5.

Stage 1 vote: 0, 1, 1, 1, 1

Replicas 1 to 3 all see the 0 vote in among the first three received broadcasts in Stage 1, while Replicas 4, and 5 only see votes of 1 in their first three received broadcasts in Stage 1. This results in the following.

Stage 2 vote: ?, ?, ?, 1, 1

In Stage 2, Replicas 2 to 5 all see at least one vote of 1 among their first three received broadcasts in Stage 2, so their initial vote for the next round is set to 1. However, Replica 1 only sees ?s, and thus must perform a coin flip to determine its initial vote for the next phase. However, since it is a biased coin, its coin flip always results in 0. Therefore, the vote at the end of the next phase is deterministically exactly the same as it was in the previous phase.

In particular, in an asynchronous network, the adversary has the capability to rearrange the order of messages in a manner consistent with the situation just described. Therefore, we conclude that as long as 1 coin is biased, Ben-Or no longer works under asynchronous conditions.

It's important to recognize how this situation is different if the coin is not completely biased (always returning 0 or 1). As discussed previously, in Stage 2, all votes are either in $\{0, ?\}$ or they're in $\{1, ?\}$. Therefore, it is impossible to deterministically generate initial votes of both 0 and 1 in the next phase, since any such situation is only possible if at least one replica performs a coin flip. Since the adversary has no control over the coinflip (which is truly random), the adversary cannot deterministically force a repeat of the votes at the beginning of every phase.

Since adversarial asynchronous network conditions are unlikely, and Raft doesn't even work under asynchronous conditions, this isn't a serious concern.

4.3 Commit Order of Entries

In addition to real-time ordering on clients, Hydrofoil satisfies another property on servers.

Theorem 4.3.1. *If a server receives a request of e_1, then e_2 later, no server will commit e_2 before e_1.*

Proof. First, we will prove that any replica that has seen e_2 will also have seen e_1. However, this is fairly obvious because we send over all of priority queue and log every time, so all replicas will see this.

Next, we prove that if either e_1 is before e_2 in the log, e_1 is in the log while e_2 is in the priority queue, or both are in the priority queue. We do this by contradiction. There are two possibilities, either e_1 is after e_2, or e_2 is in the log while e_1 is in the priority queue.

In the former case, this means that some leader must've previously only had a log with e_2, before it then became aware of e_1 and added it to its priority queue. But this is not possible, because e_1 and e_2 are always broadcast together as in the previous proof, or only e_1 is broadcast. The latter case is similar: we can't have pulled e_2 out first into the log.

Next, we will prove that e_1 always commits before e_2. Recall there are two possible ways for an entry to commit: either it is received by a leader through a replicate entries, or it is committed through Ben-Or+. As we have just proven, all logs will contain e_1 first, so it will be committed first if it's done through a replicate entries. If it's committed through Ben-Or+, then by the selection criteria, we will always choose e_1 before e_2 as our proposed entry. Thus, we are done.

Recall that in non adversarial environments, these entries will always commit. This means that eventually these entries will be committed, and e_1 will be executed before e_2 as desired. □

Chapter 5

Evaluation

To evaluate the performance of Hydrofoil, we compared it to its two individual components: Raft+ and Ben-Or+. Raft+ and Ben-Or+ are very similar to Raft and Rabia specifically. The main differences are summarized below.

Raft	**Raft+**
Client must send request to leader	Client can send request to any replica
Each entry has its own term	Uses leaderTerm

Rabia	**Ben-Or+**
After deciding value 0 in Ben-Or, commit an empty entry	After deciding value 0 in Ben-Or, restart Ben-Or+ for the same *index*
Assumes a reliable network connection	Doesn't assume a reliable network connection

We decided to run experiments against Raft+ and Ben-Or+ because there are fewer differences between them and Hydrofoil. This lets us truly test whether our hypothesis that Hydrofoil can perform as well as its leader-based and leaderless components. The implementations were coded in Go.

In our experiment, we setup 5 replicas on a local area network. A single client then repeatedly sends requests to the system and waits for a response. Upon receiving one, the client immediately sends another request. This allows us to measure the max throughput of the system.

At time 1s, a replica was disconnected from the system. In the case of Hydrofoil and Raft+, this replica was the leader. In the case of Ben-Or+, this was a random replica. Then, at time 3s, the disconnected replica was reconnected to the network. The requests were sent to all connected replicas uniformly. To reduce variance, we averaged the throughput over 10 runs per protocol.

The maximum election timeout is 1s, and the maximum Ben-Or Start Timeout is 50ms. In reality, this election timeout is longer than the typical values used for Raft; the original Raft paper recommends a timeout of 150ms-300ms. However, to really investigate and illustrate the differences between the protocols, we decided to use a longer timeout. In addition, the election timeout typically isn't the full 1s. Following our recommendation earlier to randomize the timeouts, our actual election timeout is between 500ms and 1s, and the Ben-Or start timeout is between 25ms and 50ms.

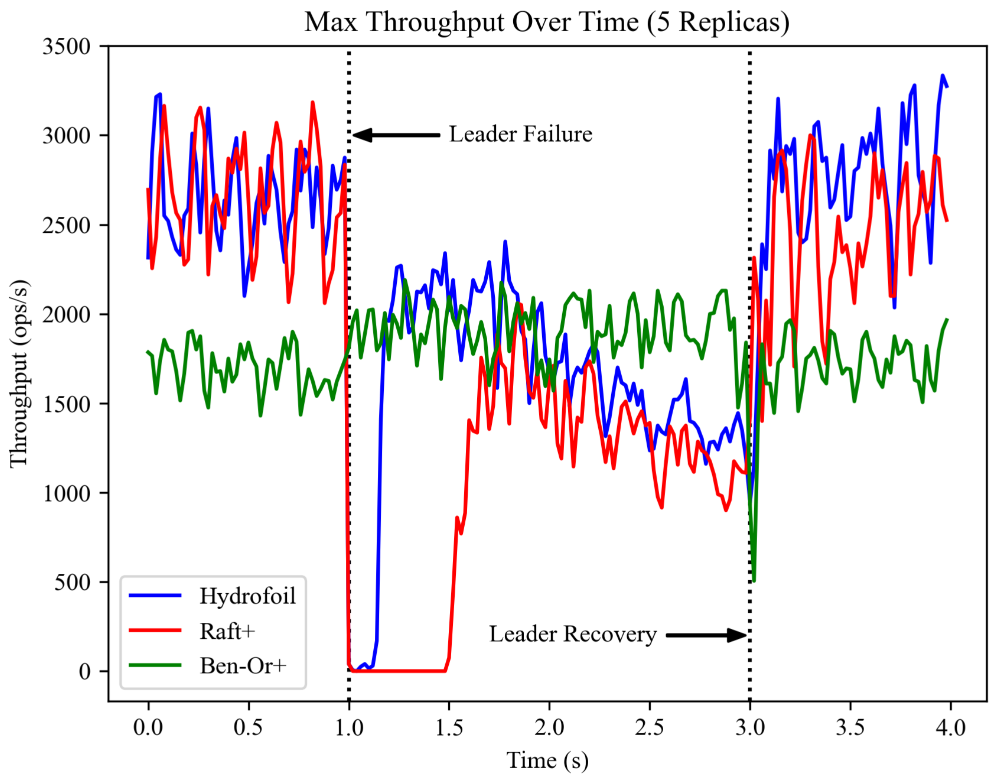

Figure 5.1: Throughput of Hydrofoil, Raft+, and Ben-Or+ over time with 5 replicas in a local area network. The leader is disconnect at time 1s in the graph, and is reconnected at time 3s.

As we can see in Figure 5.1, in the normal case with no failures, Hydrofoil and Raft+ both outperform Ben-Or+. This is expected, because committing a value using Ben-Or+ requires many additional rounds of communication.

However, when the leader fails at time 1s, both Hydrofoil's and Raft+'s throughput drops down to 0. Ben-Or+'s throughput remains unchanged, since it does not rely on the notion of a leader. However, while Raft+ must wait until a new leader is elected, which takes at least 500ms, to start committing entries again, Hydrofoil only needs to wait for the duration of its Ben-Or start timeout. At this point, Hydrofoil is using Ben-Or+ to commit entries, and it able to match the performance of the Ben-Or+ only system. At time 3s, the disconnected replica is reconnected, and throughput of both Hydrofoil and Raft+ return to their original levels.

An interesting observation is that both Hydrofoil and Ben-Or+'s throughput starts dropping at around time 1.75s and it continues to drop until the original replica is returned to the network. The reason for this behavior is that the newly elected leader, who is elected around 1.75s on average, is unaware that another replica has been disconnected. Let's denote this replica r. Every time the new leader sends a ReplicateEntries RPC to r, for instance when it is sending a heartbeat, it must send over all entries starting from nextIndex[r]. However, since r is disconnected, the leader is never able to update nextIndex[r]. In the meantime, its log is continually increasing in length as the client sends more requests. It becomes more and more expensive to prepare all the log entries starting from nextIndex[r] and send them over the network, gradually slowing the system down. Ben-Or+ isn't affected by this, since it doesn't use ReplicaEntries RPCs. A replica only broadcasts entries starting from its own commitIndex, allowing it to sidestep this problem.

This problem, however, can be resolved with an optimization. Instead of always sending all entries from nextIndex[r] to the end of the log, we instead include a cap on the number of entries sent. Until we get a positive response back from r, we don't send it newer entries beyond this cap.

Keeping this in mind, we can conclude that Hydrofoil indeed performs as well as Raft+ and Ben-Or+. In a network with no failures, Hydrofoil's performance is just as good as the performance of a leader-based protocol like Raft+. When there are failures, Hydrofoil is able to recover faster by committing using Ben-Or+, and shows no adverse effects when replicas rejoin the network.

Chapter 6

Conclusion and Future Work

In conclusion, Hydrofoil is a new RSM that can sustain the high throughput of leader-based protocols, while being versatile enough to recover quickly from leader failures or slowdowns. It achieves this by allowing new entries to be committed in two different ways. The first is Raft+, a leader-based replication scheme similar to Raft, where an elected leader attempts to get other replicas to replicate its own log. The second is Ben-Or+, a randomized leaderless replication scheme based on Ben-Or, that decides on entries to commit using a series of broadcasts. To ensure that these two protocols always agree to commit the same entry without sacrificing throughput, we introduce the use of a biased coin that always flips to 0. In normal conditions, which is when the network is not experiencing sudden failures and a majority of servers are functioning, Hydrofoil will use Raft+ to commit entries. Since Raft+ is a leader-based protocol similar to Raft, Hydrofoil is able to maintain a high throughput and low latency in such scenarios. During periods of leader contention, failures, or slowdowns, Hydrofoil will automatically start Ben-Or+ after a short timeout, which is typically much shorter than an election timeout, allowing the system to continue to commit with a respectable throughput while the leader recovers or elections complete.

Hydrofoil is hypothesized to satisfy a number of desired properties including linearizability, and fault tolerance up to a minority of replica failures. Clients can send entries to all replicas, not just the leader. Howevever, we acknowledge our proof of the Hydrofoil's linearizability guarantee, in particular the total order component, remains incomplete. A key component of Lemma 4.1.2 remains unproven. Future work could look into fully proving linearizability, or adjusting the algorithm in case safety contradictions are found.

Another property of Hydrofoil is that no request received by the system is ever lost, although clients have some control over the execution of requests by setting timeouts. Hydrofoil does not assume reliable network connections between replicas or clients, but we have shown that it isn't always live in completely asynchronous networks in the presence of a strong adversary. While the system is intuitively live in partially synchronous networks, especially if we augment the protocol using standard techniques such as exponential backoff, we have not proven this definitively.

Future experiments could also test the effectiveness of this procedure in the geo-replicated case. Leaderless protocols like Rabia are known to slow down considerably with longer network roundtrip times, so it would be interesting to explore if there still remain any benefits to using Ben-Or+. Finally, while our implementation of Hydrofoil passes the test cases that we have generated, it has not been formally proven to be correct. Future work could focus on providing a formally verified implementation, especially as new formal methods techniques and verification languages come out every year.

Bibliography

[1] Marcos K Aguilera and Sam Toueg. The correctness proof of ben-or's randomized consensus algorithm. *Distributed Computing*, 25(5):371–381, 2012.

[2] Catalonia-Spain Barcelona. Mencius: building efficient replicated state machines for wans. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, 2008.

[3] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, PODC '83, page 27–30, New York, NY, USA, 1983. Association for Computing Machinery.

[4] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *NSDI*, pages 463–478, 2020.

[5] Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[6] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350, 2006.

[7] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[8] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[9] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407, 2007.

[10] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, Mirco Marchetti, et al. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. The USENIX Association, 2009.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[12] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[13] Eli Gafni and Leslie Lamport. Disk paxos. In *Distributed Computing: 14th International Conference, DISC 2000, Maurice Herlihy, editor. Lecture Notes in Computer Science number 1914, Springer-Verlag,(2000) 330-344.*, 2003.

[14] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[15] Heidi Howard and Richard Mortier. Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–9, 2020.

[16] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[17] Michael Isard. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review*, 41(2):60–67, 2007.

[18] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[19] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, pages 1–6, 2008.

[20] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, May 1998.

[21] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[22] Leslie Lamport. Generalized consensus and paxos. 2005.

[23] Leslie Lamport. Fast paxos. *Distributed Computing*, 19:79–103, 2006.

[24] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical paxos and primary-backup replication. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 312–313, 2009.

[25] Leslie Lamport and Mike Massa. Cheap paxos. In *International Conference on Dependable Systems and Networks, 2004*, pages 307–314. IEEE, 2004.

[26] Butler Lampson. The abcd's of paxos. In *PODC*, volume 1, page 13, 2001.

[27] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *OSDI*, pages 467–483, 2016.

[28] Dahlia Malkhi, Leslie Lamport, and Lidong Zhou. Stoppable paxos. Technical Report MSR-TR-2008-192, April 2008.

[29] David Mazieres. Paxos made practical, 2007.

[30] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, et al. Turbine: Facebook's service management platform for stream processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1591–1602. IEEE, 2020.

[31] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[32] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372, 2013.

[33] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 583–598, 2020.

[34] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 8–17, 1988.

[35] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. Rabia: Simplifying state-machine replication through randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 472–487, 2021.

[36] Somasundaram Perianayagam, Akshat Vig, Doug Terry, Swami Sivasubramanian, James Christopher Sorenson III, Akhilesh Mritunjai, Joseph Idziorek, Niall Gallagher, Mostafa Elhemali, Nick Gordon, et al. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, 2022.

[37] Gal Sela, Maurice Herlihy, and Erez Petrank. Brief announcement: Linearizability: A typo. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 561–564, 2021.

[38] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1493–1509, 2020.

[39] Pasindu Tennage, Antoine Desjardins, and Eleftherios Kokoris Kogias. Mandator and sporades: Robust wide-area consensus with efficient request dissemination. *arXiv preprint arXiv:2209.06152*, 2022.

[40] Sarah Tollman, Seo Jin Park, and John K Ousterhout. Epaxos revisited. In *NSDI*, pages 613–632, 2021.

[41] Robbert Van Renesse and Deniz Altinbuken. Paxos made moderately complex. *ACM Computing Surveys (CSUR)*, 47(3):1–36, 2015.

[42] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.

[43] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. Sdpaxos: Building efficient semi-decentralized geo-replicated state machines. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 68–81, 2018.