Bipartisan Paxos: A Family of Fast, Leaderless, Modular State Machine Replication Protocols

ABSTRACT

In this paper, we present Bipartisan Paxos: a family of state machine replication protocols that achieve low latency, high throughput, and simplicity. The Bipartisan Paxos protocols can commit a command in two message delays (the theoretical minimum), achieving low latency. They do not depend on a distinguished leader for normal processing or conflict resolution, achieving high throughput. They are designed with modularity and incrementality in mind, achieving simplicity.

1 INTRODUCTION

Consensus and state machine replication are fundamental problems in distributed systems that are both well-studied in academia and widely implemented in industry. Paxos [8], one of the earliest asynchronous consensus protocols, was developed roughly 30 years ago and has since become the de-facto standard in industry [2, 4–6]. Since that time, Paxos and Multi-Paxos (the state machine replication protocol built on Paxos) have been improved along three core dimensions: latency, throughput, and simplicity.

First, the latency of Paxos—i.e. the minimum number of message delays between when a value is proposed by a client and when it is chosen by the protocol—is higher than necessary [11]. Fast Paxos [10] improves Paxos' latency to its theoretical minimum by allowing clients to propose commands directly to acceptors. Second, the throughput of Multi-Paxos is bottlenecked by the throughput of a single leader. Fast Paxos partially resolves this problem by allowing clients to bypass the leader, but this leads to high conflict rates, lowering the throughput of the protocol. Generalized Paxos [9] and GPaxos [24] reduce the number of conflicts by taking advantage of the commutativity of state machine commands, but still rely on a single arbiter to resolve conflicts when they arise. EPaxos [19, 20] and Caesar [1] are both fully leaderless and improve on Generalized Paxos by not relying on a single process either during normal processing or conflict resolution. Third, Paxos and Multi-Paxos have developed a reputation for being overly complicated, leading to a number of publications attempting to clarify the protocols [12, 13, 16, 25] and a number of protocols touted as simpler alternatives [21, 23].

Despite the large body of work, no state machine replication protocol has claimed the trifecta of low latency, high throughput, and simplicity. Existing protocols typically sacrifice one of these features for the other two. In this paper, we present Bipartisan Paxos (or BPaxos, for short): a family of asynchronous state machine replication protocols that accomplish all three. The BPaxos protocols can commit a command in two message delays (the theoretical minimum). They are also fully leaderless and do not depend on a distinguished leader for normal processing or conflict resolution. Furthermore, we employ three techniques to make the BPaxos protocols as simple as possible. First, the protocols are modular. Every protocol is composed of small subcomponents, each of which can be understood individually. Second, we re-use existing algorithms to implement these subcomponents whenever possible, reducing the cognitive burden of understanding a new protocol that is written entirely from scratch. Third, the three BPaxos protocols—Simple BPaxos, Unanimous BPaxos, and Majority Commit BPaxos—are all incremental refinements of one another. We begin with a very simple protocol, Simple BPaxos, and then slowly increase the complexity. This allows us to decouple the nuances of the protocols, understanding each in isolation.

2 THE BIPARTISAN PAXOS PROTOCOLS

The BPaxos protocols are state machine replication protocols. We assume an asynchronous network model, deterministic execution, and fail-stop failures, i.e., nodes can fail by crashing but cannot act maliciously. Throughout the paper, we assume at most f processes can fail. Every BPaxos protocol involves a set of $f + 1$ deterministic state machine replicas that all start in the same initial state. A set of clients repeatedly propose commands to be executed by the state machine replicas.

Traditional state machine replication protocols [8, 14] reach consensus on a totally ordered log of state machine commands, as illustrated in Figure 1a. State machine replicas then execute commands in log order. Because every replica starts in the same initial state and executes deterministic commands in exactly the same order, they all remain in sync. While simple, agreeing on a totally ordered sequence of state machine commands can be overly prescriptive [9, 20]. Say two commands x and y **conflict** if they do not commute–i.e. there exists a state in which executing x and then y does not produce the same responses and final state as executing y and then x. If two commands *do* conflict (e.g., a=1 and a=2), then they *do* need to be executed by every state machine replica in the same order. But, if two commands do *not* conflict (e.g., a=2 and b=1), then they do *not* need to be totally ordered. State machine replicas can execute them in either order.

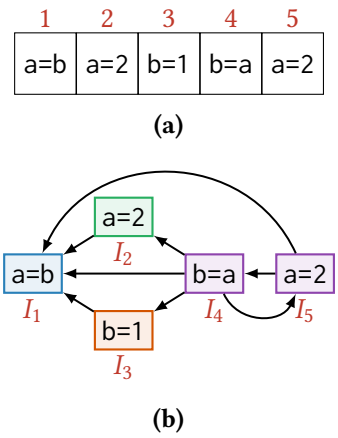


Figure 1

The BPaxos protocols leverage this observation and order commands only if they conflict. To do so, the BPaxos protocols abandon the totally ordered log and instead agree on a directed graph of commands such that every pair of conflicting commands has an edge between them. We call these graphs **partial BPaxos graphs**. An example partial BPaxos graph is illustrated in Figure 1b. Note that every pair of conflicting commands in Figure 1b has an edge between them, and some conflicting commands (e.g., b=a and a=2) have edges in both directions, forming a cycle. The process by which these partial BPaxos graphs are built is explained in the next couple of sections.

State machine replicas execute commands in these partial BPaxos graphs in reverse topological order, one strongly connected component at a time, executing commands within a strongly component in an arbitrary but deterministic order. Executing commands in this way, state machine replicas are guaranteed to remain in sync. As an example, consider again the partial BPaxos graph in Figure 1b. This graph has four strongly connected components, each shaded a different color. Replicas execute these four strongly connected components in reverse topological order. Replicas first execute a=b, then a=2 and b=1 in either order, and then b=a and a=2 in some arbitrary but deterministic order (e.g., in order of increasing hash).

Moreover, traditional state machine replication protocols assign every command a unique log entry number, as shown in Figure 1a. The BPaxos protocols follow suit and also assign each command a unique identifier, called an **instance**. For example, the command b=1 in Figure 1b is in instance I_3. Every instance I in a partial BPaxos graph has a number of outbound edges to other instances called the **dependencies** of I, denoted $\text{deps}(I)$. For example, in Figure 1b, $\text{deps}(I_5) = \{I_1, I_4\}$. The BPaxos protocols construct partial BPaxos graphs one instance at a time, reaching consensus on the command x in the instance as well as the instance's dependencies $\text{deps}(I)$. That is, for every instance I, the BPaxos protocols agree on a tuple $(x, \text{deps}(I))$. The correctness of the BPaxos protocols hinges on the following two key invariants.

Invariant 1. The BPaxos protocols successfully implement consensus for every instance I. That is, at most one value $(x, \text{deps}(I))$ is chosen in instance I (consistency), and if the value $(x, \text{deps}(I))$ is chosen, then it was previously proposed (nontriviality).

Invariant 2. If $(x, \text{deps}(I_x))$ is chosen in instance I_x and $(y, \text{deps}(I_y))$ is chosen in instance I_y, and if x and y conflict, then either $I_x \in \text{deps}(I_y)$ or $I_y \in \text{deps}(I_x)$ or both.

There are three BPaxos protocols: Simple BPaxos, Unanimous BPaxos, and Majority Commit BPaxos, summarized in Table 1. The three protocols differ in various ways (e.g., quorum sizes, commit latencies), but all three follow the structure described above. They all reach consensus on a partial BPaxos graph one instance at a time; they all execute commands in a partial BPaxos graph in reverse topological order; and they all maintain Invariant 1 and Invariant 2. In the rest of the paper, we introduce the three protocols one by one and prove that each maintains the two key invariants. For a more formal overview of the BPaxos protocols and a discussion on why the two key invariants suffice for correctness, refer to Appendix A.

Table 1: A summary of the BPaxos protocols

Protocol	Classic Quorum Size	Fast Quorum Size	Commit Latency
Simple BPaxos (Section 3)	$f+1$	N/A	4 message delays
Unanimous BPaxos (Section 6)	$f+1$	$2f+1$	2 message delays
Majority Commit BPaxos (Section 8)	$f+1$	$f+\lfloor\frac{f+1}{2}\rfloor+1$	2 message delays

3 SIMPLE BPAXOS

In this section, we introduce **Simple BPaxos**, a BPaxos protocol that is designed to be as simple as possible. Simple BPaxos consists of three logical components: a set of Simple BPaxos nodes, a dependency service, and a consensus service. The dependency service helps maintain Invariant 2, the consensus service helps maintain Invariant 1, and the Simple BPaxos nodes glue the two together. We explain each of these three components in turn.

3.1 Simple BPaxos Nodes

We assume a fixed set $b_1, \ldots, b_{f+1}$ of $f+1$ Simple BPaxos nodes. Each Simple BPaxos node b_i is a state machine replica and is responsible for learning a partial BPaxos graph and for processing commands as described in Section 2. Clients send state machine commands to BPaxos nodes for execution. When a BPaxos node b_i receives a command x, it selects a globally unique instance I for the command and sends the tuple (I, x) to the dependency service. The dependency service replies with a tuple $(I, x, \text{deps}(I))$ where $\text{deps}(I)$, the dependencies of I, is a set of instances. b_i then proposes the value $(x, \text{deps}(I))$ to the consensus service in instance I, and the consensus service replies with some chosen value $(x', \text{deps}(I)')$ (which is equal to $(x, \text{deps}(I))$ in the failure-free case). At this point, the command x' with dependencies $\text{deps}(I)'$ is chosen in instance I and is added to b_i's partial BPaxos graph for eventual execution (see Appendix A for details). b_i also informs the other Simple BPaxos nodes that the value $(x', \text{deps}(I)')$ has been chosen in instance I. After b_i executes command x, it responds to the client with the corresponding state machine response. This communication pattern is illustrated in Figure 2.

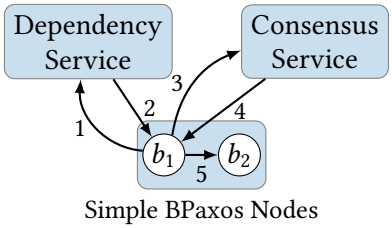


Figure 2: Simple BPaxos

3.2 Dependency Service

Upon receiving a tuple (I, x) from a Simple BPaxos node, the dependency service replies with a tuple $(I, x, \text{deps}(I))$ with the following guarantee.

Invariant 3. If two conflicting commands x and y in instances I_x and I_y yield responses $(I_x, x, \text{deps}(I_x))$ and $(I_y, y, \text{deps}(I_y))$ from the dependency service, then either $I_x \in \text{deps}(I_y)$ or $I_y \in \text{deps}(I_x)$ or both.

There are two things to note about the dependency service. First, the dependency service has a precondition that at most one command can be sent to the dependency service in any given instance. That is, if the dependency service receives tuples (I, x) and (I, y), then $x = y$. Second, the dependency service may process a tuple (I, x) more than once, yielding different responses each time. For example, Simple BPaxos node b_i may send (I, x) to the dependency service and get a response $(I, x, \{I_1, I_2\})$. Later, b_j might send (I, x) to the dependency service and get a different response of $(I, x, \{I_2, I_3\})$. Note that even though the dependency service may produce different responses for the same request, the dependency service maintains Invariant 3 for every possible pair of dependency service responses.

We now describe how to implement the dependency service. We employ $2f+1$ dependency service nodes $d_1, \ldots, d_{2f+1}$. When a Simple BPaxos node b_j sends the tuple (I, x) to the dependency service, it sends the tuple to all $2f+1$ of the dependency service nodes. Every dependency service node d_i maintains a partial

BPaxos graph C_i.[1] When d_i receives the tuple (I, x) from a Simple BPaxos node, it performs the following actions. First, if C_i does not already contain vertex I, then d_i inserts vertex I into C_i with label x and with edges to every other instance I' in C_i that is labelled with a command that conflicts with x. Second, d_i returns the tuple $(I, x, \text{out}(I))$ where $\text{out}(I)$ is the set of instances in C_i with an inbound edge from I.

When a Simple BPaxos node b_j receives replies $(I, x, \text{deps}(I)_{i_1}), \ldots, (I, x, \text{deps}(I)_{i_{f+1}})$ from a quorum Q of $f+1$ dependency service nodes $d_{i_1}, \ldots, d_{i_{f+1}}$, it takes $(I, x, \text{deps}(I)_{i_1} \cup \ldots \cup \text{deps}(x)_{i_{f+1}})$ to be the response from the dependency service. That is, b_j computes $\text{deps}(I)$ by taking the union of dependencies from a majority of the dependency service nodes. This implementation maintains Invariant 3.

Proof. Consider conflicting commands x and y in instances I_x and I_y. Assume request (I_x, x) yielded a dependency service reply $(I_x, x, \text{deps}(I_x))$ that was formed from a quorum Q_x. Similarly, (I_y, y) yielded reply $(I_y, y, \text{deps}(I_y))$ formed from a quorum Q_y. Any two quorums intersect, so $Q_x \cap Q_y$ is nonempty. Let d_i be a dependency service node in this intersection. d_i either received (I_x, x) or (I_y, y) first. If it received I_x first, then I_y has an edge to I_x in C_i, so $I_x \in \text{deps}(I_y)$. Symmetrically, if it received I_y first, then I_x has an edge to I_y in C_i, so $I_y \in \text{deps}(I_x)$. □

3.3 Consensus Service

We assume a consensus service that implements consensus for every instance I. A Simple BPaxos node can propose to the consensus service that some value $v = (x, \text{deps}(I))$ be chosen in some instance I. The consensus service replies with the value that has been chosen in instance I, which may or may not be v. Note that we overload the term "instance". The consensus service implements one *instance* of consensus for every *instance* I. The consensus service can be implemented with any consensus protocol (e.g., Paxos [8, 12], Fast Paxos [10], Flexible Paxos [7]).

3.4 An Example

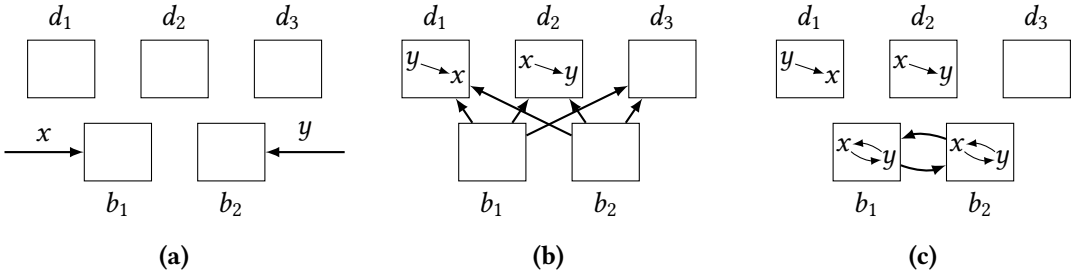


Figure 3: Example Simple BPaxos execution

An example execution of Simple BPaxos with $f = 1$ is illustrated in Figure 3. In Figure 3a, Simple BPaxos node b_1 receives command x from a client, while b_2 receives conflicting command y. b_1 sends tuple (I_x, x) to the dependency service nodes, while b_2 sends (I_y, y). d_1 receives x and then y, d_2 receives y and then x, and d_3 does not receive anything because of a network partition. d_1 replies with values $(I_x, x, \emptyset)$ and $(I_y, y, \{I_x\})$, while d_2 replies with values $(I_y, y, \emptyset)$ and $(I_x, x, \{I_y\})$. b_1 and b_2 compute responses $(I, x, \emptyset \cup \{I_y\})$ and $(I, y, \{I_x\} \cup \emptyset)$ from the dependency service respectively, and propose values $(x, \{I_y\})$ in instance I_x and $(y, \{I_x\})$ in instance I_y to the consensus service (not illustrated in Figure 3). The consensus service chooses these proposed values, and after communicating with one another, both b_1 and b_2 learn the partial BPaxos graph shown in Figure 3c. Then, they both execute x and y in some arbitrary but deterministic order (e.g., in increasing hash order).

[1] Technically, C_i is a conflict graph. See Appendix A.1.

3.5 Recovery

Note that it is possible that a command x chosen in instance I depends on an unchosen instance I'. If instance I' remains forever unchosen, then the command x will never be executed. To avoid this liveness violation, if any Simple BPaxos node b_i notices that instance I' has been unchosen for some time, b_i can propose to the consensus service that the command noop be chosen in instance I' with no dependencies. noop is a distinguished command that does not affect the state machine and does not conflict with any other command. Alternatively, b_i can contact the dependency service and check if any dependency service node has recorded a command y in instance I'. If such a command exists, b_i can send the tuple (I', y) to the dependency service, and propose y with the resulting dependencies to the consensus service. If no such y exists, b_i can propose a noop.

3.6 Safety

We now prove that Simple BPaxos maintains Invariant 1 and Invariant 2. Simple BPaxos maintains Invariant 1 trivially by leveraging the consensus service. Simple BPaxos maintains Invariant 2 by maintaining the following invariant.

Invariant 4. For every instance I, a value $(x, \text{deps}(I))$ is chosen in instance I only if $(I, x, \text{deps}(I))$ is a response from the dependency service or if $(x, \text{deps}(I)) = (\text{noop}, \emptyset)$.

Invariant 2 follows immediately from Invariant 3 and Invariant 4. Simple BPaxos maintains Invariant 4 because Simple BPaxos nodes only propose dependencies computed by the dependency service (or noops). Note that proposing noops does not affect Invariant 2 because noops do not conflict with any other command, so Invariant 2 holds vacuously in that case.

3.7 Reducing Commit Latency

For ease of exposition, we stated that clients forward state machine commands to Simple BPaxos nodes and that Simple BPaxos nodes are responsible for proposing values to the dependency service and to the consensus service. In reality, there is nothing preventing clients from taking a more active role in the protocol. A client can behave exactly like a BPaxos node, assigning instances to commands and proposing values to the dependency service and consensus service. By allowing clients to take a more active role, the commit latency of the protocol is reduced. To keep things as simple as possible, in the rest of the paper, we will continue to describe the BPaxos protocols with passive clients, but keep in mind that clients can always take a more active role to reduce commit latency. Note that even with active clients, BPaxos nodes are still responsible for learning and executing commands and for initiating the recovery procedure.

3.8 Discussion

Simple BPaxos exemplifies the simplicity of the BPaxos protocols. Simple BPaxos is composed of three subcomponents—the Simple BPaxos nodes, the dependency service, and the consensus service—that can all be understood independently. In fact, all of the BPaxos protocols follow this structure. Moreover, by leveraging an existing consensus protocol to implement the consensus service, Simple BPaxos is able to avoid a lot of unnecessary complexity. Note that Simple BPaxos achieves high throughput by being completely leaderless but does *not* achieve optimal commit latency, even with active clients. The remaining BPaxos protocols are leaderless and do achieve the optimal commit latency (in the best case).

4 AN ASIDE: FAST PAXOS

The remaining BPaxos protocols all leverage Fast Paxos [10]. We assume a familiarity with Fast Paxos, but pause briefly to highlight the salient bits of Fast Paxos here. For a more in-depth discussion of Fast Paxos, refer to Appendix B. Fast Paxos proceeds in a series of integer-valued rounds with 0 being the smallest round and -1 being a null round. Every round is classified either as a fast round or a classic round. In phase 2a of the algorithm, a leader has to choose a value to send to the acceptors. The logic for choosing this

Algorithm 1 Fast Paxos Phase 2a

1: $M \leftarrow$ phase 1b messages from a quorum Q
2: $k \leftarrow$ the largest vote round in M
3: $V \leftarrow$ the vote values in M for round k
4: **if** $k = -1$ **then**
5: send any proposed value
6: **else if** $V = \{v\}$ **then**
7: send v
8: **else if** $\exists v \in V.\ O4(v)$ **then**
9: send v
10: **else**
11: send any proposed value

Algorithm 2 Fast Paxos Phase 2a Tweak

1: $M \leftarrow$ phase 1b messages from a quorum Q
2: $k \leftarrow$ the largest vote round in M
3: $V \leftarrow$ the vote values in M for round k
4: **if** $k = -1$ **then**
5: send any proposed value
6: **else if** $k \neq 0$ **then**
7: send unique $v \in V$
8: **else if** $\exists v \in V$ maybe chosen in round 0 **then**
9: send v
10: **else**
11: send any proposed value

value is shown in Algorithm 1 where $O4(v)$ is true if there exists a fast quorum $\mathcal{F}$ of acceptors such that every acceptor in $\mathcal{F} \cap Q$ voted for v in round k. The key invariant of Fast Paxos is that if a value v was *maybe* chosen in a round less than i, then a leader must propose v in round i. This is the case in line 7 and line 9 of Algorithm 1. A leader can propose an arbitrary value in round i only if it has concluded that no value was chosen in any round less than i, as is the case in line 5 and line 11.

If we assume that round 0 is a fast round and every other round is a classic round, we can simplify the standard phase 2a algorithm shown in Algorithm 1 to the variant shown in Algorithm 2. As with Algorithm 1, a leader is sometimes forced to propose a value v if it was maybe previously chosen (i.e. line 7 and line 9 of Algorithm 2). The process of determining whether a value v in line 8 of Algorithm 2 was maybe chosen in round 0 is left intentionally abstract. The correctness proof of this alternative phase 2a is given in Appendix B.

5 UNSAFE BPAXOS

Simple BPaxos is easy to understand, but it has suboptimal commit latency. With active clients, it takes at least two round trips before a command is chosen: one round trip to the dependency service and one round trip to the consensus service. In this section, we present a purely pedagogical BPaxos protocol called **Unsafe BPaxos**. Unsafe BPaxos attempts to choose commands in one round trip, but as the name suggests, Unsafe BPaxos is unsafe. It does not properly implement state machine replication. Still, understanding why Unsafe BPaxos is unsafe leads to some fundamental insights into the BPaxos protocols still to come.

5.1 The Protocol

Unsafe BPaxos consists of the same three components as Simple BPaxos: a set of Unsafe BPaxos nodes, a dependency service, and a consensus service. Unsafe BPaxos is largely identical to Simple BPaxos except for the following differences. First, we implement the consensus service using Fast Paxos, with one instance of Fast Paxos for every instance I. The consensus service is implemented as a set $a_1, \ldots, a_{2f+1}$ of $2f + 1$ Fast Paxos acceptors, and the $f + 1$ Unsafe BPaxos nodes play the role of Fast Paxos leaders. We employ classic quorums of size $f + 1$ and fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor + 1$. For every instance I, we let round 0 be a fast round and every other round be a classic round. Doing so, we can skip phase 1a, phase 1b, and phase 2a of round 0 and have every Fast Paxos acceptor initialized in round 0 as if it had received a phase 2a message with distinguished value *any*. Thus, a Fast Paxos acceptor can vote for the first proposal that it receives for instance I. Second, we physically co-locate the $2f + 1$ dependency service nodes and the $2f + 1$ Fast Paxos acceptors such that dependency service node d_i and Fast Paxos acceptor a_i are on the same physical machine.

As with Simple BPaxos, when an Unsafe BPaxos node b_i receives a state machine command x from a client, it selects a globally unique instance I and sends the tuple (I, x) to the dependency service. Upon receiving

(I, x), dependency service node d_j computes its reply $(I, x, \text{deps}(I)_j)$. An Unsafe BPaxos dependency service node behaves identically to a Simple BPaxos dependency service node, with the one exception that d_j does not return its reply $(I, x, \text{deps}(I)_j)$ directly to b_i. Instead, it proposes the value $(x, \text{deps}(I)_j)$ in instance I to a_j (the co-located Fast Paxos acceptor). As we described above, a_j votes for the first proposal that it receives for instance I, so a_j votes for the value $(x, \text{deps}(I)_j)$ in instance I and relays its phase 2b vote back to b_i. If b_i receives a fast quorum of phase 2b votes for the same value $v = (x, \text{deps}(I)_{j_1}) = \cdots = (x, \text{deps}(I)_{j_m})$ in instance I (where $m = f + \lfloor \frac{f+1}{2} \rfloor + 1$), then b_i learns that v is chosen. b_i updates its partial BPaxos graph and informs the other Unsafe BPaxos nodes. This communication pattern is illustrated in Figure 4.

If b_i does *not* receive a fast quorum of phase 2b votes for some value v in round 0, then it is unsure whether or not a value was chosen and begins recovery for instance I. Another Unsafe BPaxos node b_j may also begin recovery for instance I if it detects that instance I has been unchosen for some time. Unsafe BPaxos node b_i recovers instance I by attempting to get a value chosen in instance I in a higher Fast Paxos round. That is, b_i chooses a larger round number and executes the full two phases of Fast Paxos. If b_i executes line 5 or line 11 of Algorithm 1, b_i can propose the value $(\text{noop}, \emptyset)$, or it can can contact the dependency service to see if a command x has already been proposed in instance I and then propose $(x, \text{deps}(I))$ where $\text{deps}(I)$ is computed by the dependency service.

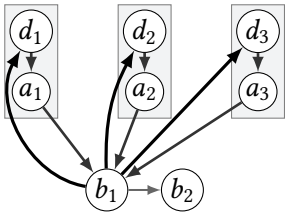


Figure 4: Unsafe BPaxos

5.2 Lack of Safety

We now explain why Unsafe BPaxos is unsafe. Consider an Unsafe BPaxos deployment with $f = 2$. Assume Unsafe BPaxos node node b_i is recovering instance I_x. b_i sends phase 1a messages in round 1 to the Fast Paxos acceptors and receives the following phase 1b responses: a_3 and a_4 voted for $(x, \{I_y\})$ in round 0, and a_5 voted for $(x, \{I_z\})$ in round 0. b_i then executes Algorithm 1. The value $v = (x, \{I_y\})$ satisfies $O4(v)$ because the fast quorum $\{a_1, a_2, a_3, a_4\}$ may have unanimously voted for v in round 0. Thus, b_i executes line 9 of Algorithm 1 and proposes the value $(x, \{I_y\})$. This is incorrect! It's possible that a_1 and a_2 did *not* vote for $(x, \{I_y\})$ in round 0. For example, they could have voted for $(x, \{I_z\})$. In this case, $\{I_y\}$, the dependencies proposed by b_i, were not computed by the dependency service since $\{I_y\}$ is only the union of a *minority* of dependency service node replies. This violates Invariant 4 and consequently Invariant 2. See Appendix C for a more complete example.

This example illustrates a fundamental tension between preserving Invariant 1 and preserving Invariant 2. Maintaining Invariant 1 in isolation is easy (e.g., use Paxos), and maintaining Invariant 2 in isolation is also easy (e.g., use the dependency service). But, maintaining both invariants simultaneously is tricky. The consensus service wants to get *any* value chosen, while the dependency service requires that only *certain* values be chosen. This leads to situations, like the one in our example above, where a BPaxos node is forced to propose a particular value $(x, \text{deps}(I_x))$ in order to preserve Invariant 1 (e.g., because the value may have been chosen in an earlier Fast Paxos round) and simultaneously forced *not* to propose the value in order to preserve Invariant 2 (e.g., because $\text{deps}(I_x)$ may not have been computed by the dependency service).

In the next few sections, we introduce a handful of BPaxos protocols, each of which runs into this fundamental tension. Every BPaxos protocol must resolve the tension between the consensus service and dependency service, forcing the two parties to cooperate in order to maintain Invariant 1 and Invariant 2. The distinguishing feature of every protocol is how it resolves the tension. The cooperation between these two opposing parties is why we named our protocols Bipartisan Paxos.

6 UNANIMOUS BPAXOS

Unanimous BPaxos is identical to Unsafe BPaxos except for the following small modifications. First, we increase the fast quorum size from $f + \lfloor \frac{f+1}{2} \rfloor + 1$ to $2f + 1$. Thus, choosing a value in round 0 requires a unanimous vote. Second, we implement Unanimous BPaxos with the Fast Paxos tweak shown in Algorithm 2 where value v in line 8 may have been chosen in round 0 only if every acceptor in Q voted for v in round

0. Like Unsafe BPaxos, Unanimous BPaxos can choose a command in one round trip (in the best case), but unlike Unsafe BPaxos, Unanimous BPaxos is safe. Unanimous BPaxos maintains Invariant 1 trivially by using Fast Paxos, and like Simple BPaxos, Unanimous BPaxos maintains Invariant 2 by maintaining Invariant 4 (see Appendix D for proof). Intuitively, increasing the fast quorum size from $f + \lfloor \frac{f+1}{2} \rfloor + 1$ to $2f + 1$ resolves the tension between maintaining Invariant 1 and Invariant 2 by eliminating the scenarios in which a BPaxos node b_i is simultaneously forced to propose a value v to maintain Invariant 1 and forced *not* to propose v to maintain Invariant 2. In particular, whenever a Unanimous BPaxos node b_i is forced to propose a value $v = (x, \text{deps}(I_x))$ on line 7 or line 9 of Algorithm 2, the large fast quorum sizes guarantee that $\text{deps}(I)$ was computed by the dependency service and is therefore safe to propose.

7 DEADLOCK BPAXOS

In this section, we present **Deadlock BPaxos**, a slight tweak to Unanimous BPaxos. Like Unanimous BPaxos, Deadlock BPaxos can choose a command in one round trip, but unlike Unanimous BPaxos, Deadlock BPaxos only requires fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor + 1$ (the same as Fast Paxos). While Deadlock BPaxos is safe, it is not very live. There are failure-free situations in which Deadlock Paxos can deadlock. Thus, Deadlock BPaxos, like Unsafe BPaxos, is a purely pedagogical protocol. In the next section, we fix Deadlock BPaxos's lack of liveness.

7.1 Pruned Dependencies

We discuss Deadlock BPaxos momentarily, but first we pause to understand one of the key invariants that it maintains. Recall that in order to preserve Invariant 2, both Simple BPaxos and Unanimous BPaxos maintain Invariant 4. Deadlock BPaxos does not maintain this invariant. Instead, it maintains an invariant that is weaker but still sufficient to imply Invariant 2. The motivation for the weaker invariant is as follows.

Imagine a Deadlock BPaxos node b_i sends command x to the dependency service in instance I, and the dependency service responds with $(I, x, \text{deps}(I))$. Let $I' \in \text{deps}(I)$ be one of I's dependencies. Assume that that b_i knows that I' has been chosen with value $(y, \text{deps}(I'))$. Further assume that $I \in \text{deps}(I')$. In this case, there is no need for b_i to include I' in the dependencies of I! Invariant 2 asserts that if two instances I and I' are chosen with conflicting commands x and y, then either $I \in \text{deps}(I')$ or $I' \in \text{deps}(I)$. Thus, if I' has already been chosen with a dependency on I, then there is no need to propose I with a dependency on I'. Similarly, if I' has been chosen with noop, then there is no need to propose I with a dependency on I' because x and noop do not conflict. Let $(I, x, \text{deps}(I))$ be a response from the dependency service. Let $P \subseteq \text{deps}(I)$ be a set of instances I' such that I' has been chosen with noop or I' has been chosen with $I \in \text{deps}(I')$. We call $\text{deps}(I) - P$ the **pruned dependencies** of I with respect to P. Deadlock BPaxos maintains Invariant 5. Invariant 3 and Invariant 5 imply Invariant 2 (see Appendix E.1 for proof).

Invariant 5. For every instance I, a value v is chosen in instance I only if $v = (\text{noop}, \emptyset)$ or if $(I, x, \text{deps}(I))$ is a response from the dependency service and $v = (I, x, \text{deps}(I) - P)$ where $\text{deps}(I) - P$ are the pruned dependencies of I with respect to some set P.

7.2 The Protocol

Deadlock BPaxos is identical to Unanimous BPaxos except for the following modifications. First, Deadlock BPaxos fast quorums are of size $f + \lfloor \frac{f+1}{2} \rfloor + 1$. Second, every Fast Paxos acceptor maintains a partial BPaxos graph in exactly the same way as the set of Deadlock BPaxos nodes. When an acceptor learns that an instance I has been chosen with value $(x, \text{deps}(I))$, it adds I to its partial BPaxos graph labelled with x and with edges to every instance in $\text{deps}(I)$. We will see momentarily that whenever a Deadlock BPaxos node b_i sends a phase 2a message to acceptors with value $v = (x, \text{deps}(I) - P)$ for some pruned dependencies $\text{deps}(I) - P$, b_i includes P and all of the values chosen in P in the phase 2a message. Thus, when an acceptor receives a phase 2a message, it updates its partial BPaxos graph with the values chosen in P. Third, as discussed above, Deadlock BPaxos maintains Invariant 5 instead of Invariant 4. Fourth and most substantially, when a Deadlock BPaxos node b_i executes line 8 of Algorithm 2 for instance I, it performs a

much more sophisticated procedure to determine if some value $v \in V$ may have been chosen in round 0. This procedure is shown in Algorithm 3.

Algorithm 3 Deadlock BPaxos recovery of instance I by b_i

```
1:  if ∄v ∈ V such that O4(v) then
2:      send any value satisfying Invariant 5
3:  else
4:      (x, deps(I)) ← v
5:  send (I, x) to quorum Q of dependency service nodes
6:  (I, x, deps(I)_Q) ← the dependency service response
7:
8:  P ← ∅
9:  for I' ∈ deps(I)_Q − deps(I) do
10:     if b_i doesn't know if I' is chosen then
11:         recover I', blocking until I' is recovered
12:     if I' chosen with noop or with I ∈ deps(I') then
13:         P ← P ∪ {I'}
14:     else
15:         contact a quorum Q' of acceptors
16:         if an acceptor in Q' knows I is chosen then
17:             abort recovery; I has already been chosen
18:         else
19:             send any value satisfying Invariant 5
20: send (x, deps(I)) along with values chosen in P
```

In line 1, b_i determines whether there exists some $v \in V$ that satisfies $O4(v)$. $O4(v)$ is a necessary condition for v to have been chosen in round 0, so if no such v exists, then no value could have been chosen in round 0. Thus, in line 2, b_i is free to propose any value that maintains Invariant 5. Otherwise, there does exist a $v = (x, \text{deps}(I)) \in V$ satisfying $O4(v)$. As we saw with Unsafe BPaxos, if v was maybe chosen in round 0, then b_i *must* propose v in order to maintain Invariant 1. But simultaneously to maintain Invariant 2, b_i must *not* propose v unless $\text{deps}(I)$ was computed by the dependency service. Unanimous BPaxos resolved this tension by increasing fast quorum sizes. Deadlock BPaxos resolves the tension by performing a more sophisticated recovery procedure. In particular, b_i does a bit of detective work to conclude either that v was definitely not chosen in round 0 (in which case, b_i can propose a different value) or that $\text{deps}(I)$ happens to be a pruned set of dependencies (in which case, b_i is safe to propose v).

In lines 4 and 5, b_i sends (I, x) to the dependency service nodes co-located with the acceptors in Q and receives the corresponding dependency service reply $(I, x, \text{deps}(I)_Q)$.[2] $\text{deps}(I)_Q$ is the union of the dependencies computed by these co-located dependency service nodes. Moreover, a majority of these co-located dependency service nodes computed the dependencies of I to be $\text{deps}(I)$ (because $O4(v)$). Thus, $\text{deps}(I) \subseteq \text{deps}(I)_Q$.

Next, b_i enters a for loop in an attempt to prune $\text{deps}(I)_Q$ until it is equal to $\text{deps}(I)$. That is, b_i attempts to construct a set of instances P such that $\text{deps}(I) = \text{deps}(I)_Q - P$ is a set of pruned dependencies. For every, $I' \in \text{deps}(I)_Q - \text{deps}(I)$, b_i first recovers I' if b_i does not know if a value has been chosen in instance I'. After recovering I', assume b_i learns that I' is chosen with value $(x', \text{deps}(I'))$. If $x' = \text{noop}$ or if $I \in \text{deps}(I')$, then b_i can safely prune I' from $\text{deps}(I)_Q$, so it adds I' to P. Otherwise, b_i contacts some quorum Q' of acceptors. If any acceptor a_j in Q' knows that instance I has already been chosen, then b_i can abort the recovery of I and retrieve the chosen value directly from a_j. Otherwise, in line 19, b_i concludes that v was not chosen in round 0 and is free to propose any value that maintains Invariant 5. We will explain momentarily why b_i is able to make such a conclusion. It is not obvious. Finally, if b_i exits the for loop, then it has successfully pruned $\text{deps}(I)_Q$ into $\text{deps}(I)_Q - P = \text{deps}(I)$ and can safely propose it without violating Invariant 1 or Invariant 2. As described above, when b_i sends a phase 2a message with value v, it also includes the values chosen in every instance in P.

We now return to line 19 and explain how b_i is able to conclude that v was not chosen in round 0. On line 19, b_i has already concluded that I' was not chosen with noop and that $I \notin \text{deps}(I')$. By Invariant 5, $\text{deps}(I') = \text{deps}(I')_{\mathcal{R}} - P'$ is a set of pruned dependencies where $\text{deps}(I')_{\mathcal{R}}$ is a set of dependencies computed by a quorum $\mathcal{R}$ of dependency service nodes. Because $I \notin \text{deps}(I')_{\mathcal{R}} - P'$, either $I \notin \text{deps}(I')_{\mathcal{R}}$ or $I \in P'$. I cannot be in P' because if I' were chosen with dependencies $\text{deps}(I')_{\mathcal{R}} - P'$, then some quorum of acceptors would have received P' and learned that I was chosen. But, when b_i contacted the quorum Q' of acceptors,

[2] Note that (I, x) messages can be piggybacked on the phase 1a messages previously sent by b_i. This eliminates the extra round trip.

none knew that I was chosen, and any two quorums intersect. Thus, $I \notin \text{deps}(I')_{\mathcal{R}}$. Thus, every dependency service node in $\mathcal{R}$ processed instance I' before instance I. If not, then a dependency service node in $\mathcal{R}$ would have computed I as a dependency of I'. However, if every dependency service node in $\mathcal{R}$ processed I' before I, then there cannot exist a fast quorum of dependency service nodes that processed I before I'. In this case, $v = (x, \text{deps}(I))$ could not have been chosen in round 0 because it necessitates a fast quorum of dependency service nodes processing I before I' because $I' \notin \text{deps}(I)$.

7.3 Safety

Deadlock BPaxos maintains Invariant 1 by implementing Fast Paxos with the phase 2a tweak outlined in Algorithm 2 and expanded in Algorithm 3. As described above, when a node b_i executes Algorithm 3, it makes sure to propose a value v if v was maybe chosen in round 0. b_i proposes a different value in phase 2a only if it concludes that no value was chosen in round 0. Thus, Algorithm 3 faithfully implements Algorithm 2, and Deadlock BPaxos successfully maintains Invariant 1. As explained above, Deadlock BPaxos maintains Invariant 2 by maintaining Invariant 5. The proof that Deadlock BPaxos maintains Invariant 5 is a straightforward extension of the proof given in Appendix D with a case analysis on Algorithm 3 using the arguments presented above.

Unfortunately, while Deadlock BPaxos is safe, it is not very live. There are certain failure-free situations in which Deadlock BPaxos can permanently deadlock. The reason for this is line 11 of Algorithm 3 in which a Deadlock BPaxos node defers the recovery of one instance for the recovery of another. There exists executions of Deadlock BPaxos with a chain of instances $I_1, \ldots, I_m$ where the recovery of every instance I_i depends on the recovery of instance $I_{i+1 \bmod m}$. See Appendix E.2 for a concrete example.

8 MAJORITY COMMIT BPAXOS

8.1 The Protocol

Majority Commit BPaxos is identical to Deadlock BPaxos except for the following modifications to prevent deadlock. First, we change the condition under which a value is considered chosen on the fast path. Deadlock BPaxos considered a value $v = (x, \text{deps}(I))$ chosen on the fast path if a fast quorum $\mathcal{F}$ of acceptors voted for v in round 0. Majority Commit BPaxos, on the other hand, considers a value $v = (x, \text{deps}(I))$ chosen on the fast path if a fast quorum $\mathcal{F}$ of acceptors voted for v in round 0 *and* for every instance $I' \in \text{deps}(I)$, there exists a quorum $\mathcal{Q}' \subseteq \mathcal{F}$ of $f + 1$ acceptors that knew I' was chosen at the time of voting for v. Second, when a Fast Paxos acceptor a_i sends a phase 2b vote in round 0 for value $v = (x, \text{deps}(I))$, a_i also includes the subset of instances in $\text{deps}(I)$ that a_i knows are chosen, as well as the values chosen in these instances. Third, Majority Commit BPaxos nodes execute Algorithm 3 but with the lines of code shown in Algorithm 4 inserted after line 4.

Algorithm 4 Majority Commit BPaxos recovery of instance I by b_i

```
5:  A ← the set of acceptors in Q that voted for v in round 0
6:  if ∃I' ∈ deps(I) such that no acceptor in A knows that I' is chosen then
7:      send any value satisfying Invariant 5
8:  if b_i was recovering I' and deferred to the recovery of I then
9:      if I' ∈ deps(I) then
10:         abort recovery of I'; I' has already been chosen
11:     else
12:         in instance I', send any value satisfying Invariant 5
```

We now explain Algorithm 4. On line 5, b_i computes the subset $\mathcal{A} \subseteq \mathcal{Q}$ of acceptors that voted for v in round 0. On line 6, b_i determines whether there exists some instance $I' \in \text{deps}(I)$ such that no acceptor in $\mathcal{A}$ knows that I' is chosen. If such an I' exists, then v was not chosen in round 0. To see why, assume for contradiction that v was chosen in round 0. Then, there exists some fast quorum $\mathcal{F}$ of acceptors that

voted for v in round 0, and there exists some quorum $Q' \subseteq \mathcal{F}$ of acceptors that know I' has been chosen. However, any two quorums intersect, but no acceptor in Q both voted for v in round 0 and knows that I' was chosen. This is a contradiction. Thus, b_i is free to propose any value satisfying Invariant 5 on line 7.

Next, it's possible that b_i was previously recovering instance I' with value $v' = (x', \text{deps}(I'))$ and executed line 11 of Algorithm 3, deferring the recovery of instance I' until after the recovery of instance I. If so, b_i continues on line 9. If $I' \in \text{deps}(I)$, then some acceptor $a_j \in \mathcal{A}$ knows that I' is chosen. Thus, b_i can abort the recovery of instance I' and retrieve the chosen value directly from a_j. Otherwise, $I' \notin \text{deps}(I)$. In this case, no value was chosen in round 0 of instance I', so b_i is free to propose any value satisfying Invariant 5 in instance I'. Here's why. $I' \notin \text{deps}(I)$, so every dependency service node co-located with an acceptor in $\mathcal{A}$ processed I before I'. $|\mathcal{A}| \geq \lfloor \frac{f+1}{2} \rfloor + 1$, so there strictly fewer than $f + \lfloor \frac{f+1}{2} \rfloor + 1$ remaining dependency service nodes that could have processed I' before I. If b_i was recovering instance I' but deferred to the recovery of instance I on line 11 of Algorithm 3, then $I \notin \text{deps}(I')$. In order for I' to have been chosen in round 0 with $I \notin \text{deps}(I')$, it requires that at least $f + \lfloor \frac{f+1}{2} \rfloor + 1$ dependency service nodes processed I' before I, which we just concluded is impossible. Thus, I' was not chosen in round 0.

8.2 Safety and Deadlock Freeness

Majority Commit BPaxos' proof of safety is a natural extension of Deadlock BPaxos's proof of safety. Majority Commit BPaxos maintains Invariant 1 by implementing Fast Paxos, and it maintains Invariant 2 by maintaining Invariant 5. Moreover, Majority Commit BPaxos is deadlock free. If a Majority Commit BPaxos node b_i is recovering instance I' and defers to the recovery of instance I, then either b_i will recover I using line 7 of Algorithm 4 or b_i will recover I' using line 10 or line 12 of Algorithm 4. In either case, any potential deadlock is avoided.

9 RELATED WORK

Egalitarian Paxos Many of the core ideas behind BPaxos were taken directly from Egalitarian Paxos (EPaxos) [19, 20]. For example, EPaxos and the BPaxos protocols both construct directed graphs of commands and execute commands in reverse topological order, one strongly connected component at a time. However, EPaxos and the BPaxos protocols also have a number of fundamental differences (e.g., BPaxos is more modular and has optimal commit latency). See Appendix F for a more detailed comparison.

A Family of Leaderless Generalized Consensus Algorithms In [15], Losa et al. propose a generic generalized consensus algorithm that is structured as the composition of a generic dependency-set algorithm and a generic map-agreement algorithm. The invariants of the dependency-set and map-agreement algorithm are essentially identical to Invariant 2 and Invariant 1, and the example implementations of these two algorithms in [15] form an algorithm similar to Simple BPaxos. Our paper builds on this body of work by introducing Incorrect BPaxos, Unanimous BPaxos, Deadlock BPaxos, and Majority Commit BPaxos.

Caesar Caesar [1] is very similar to EPaxos and Majority Commit BPaxos, but Caesar allows a command to be chosen on the fast path even if there does *not* exist a fast quorum of nodes that agree on the command's dependencies. Instead, Caesar assigns timestamps to commands and only requires that a fast quorum of nodes agree on a command's timestamp. Doing so, Caesar implements generalized consensus without maintaining Invariant 1. This makes the protocol more challenging to understand.

Generalized Paxos Generalized Paxos [9] exploits the commutativity of state machine commands to reduce the number of conflicts that arise in non-generalized consensus algorithms. However, when a collision occurs, a distinguished leader is responsible for arbitrating collision recovery. The BPaxos protocols do not depend on a leader for normal case processing or collision recovery.

Complementary Paxos Variants S-Paxos [3] is a Paxos variant that decouples control flow from data flow. Flexible Paxos [7] is a Paxos variant that allows for flexible and dynamic quorum sizes. Speculative Paxos [22] is a Paxos variant in which replicas speculatively execute state machine commands before they are known to be chosen. We believe that the BPaxos protocols could benefit from ideas in all three papers.

REFERENCES

[1] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. 2017. Speeding up Consensus by Chasing Fast Decisions. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*. IEEE, 49–60.

[2] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. 2011. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, Vol. 11. 223–234.

[3] Martin Biely, Zarko Milosevic, Nuno Santos, and Andre Schiper. 2012. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*. IEEE, 111–120.

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

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

[6] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2013. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31, 3 (2013), 8.

[7] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2017. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone (Eds.), Vol. 70. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 25:1–25:14. https://doi.org/10.4230/LIPIcs.OPODIS.2016.25

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

[9] Leslie Lamport. 2005. Generalized consensus and Paxos. (2005).

[10] Leslie Lamport. 2006. Fast paxos. *Distributed Computing* 19, 2 (2006), 79–103.

[11] Leslie Lamport. 2006. Lower bounds for asynchronous consensus. *Distributed Computing* 19, 2 (2006), 104–125.

[12] Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

[13] Butler Lampson. 2001. The ABCD's of Paxos. In *PODC*, Vol. 1. 13.

[14] Barbara Liskov and James Cowling. 2012. Viewstamped replication revisited. (2012).

[15] Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. 2016. Brief announcement: A family of leaderless generalized-consensus algorithms. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*. ACM, 345–347.

[16] David Mazieres. 2007. Paxos made practical. *Unpublished manuscript, Jan* (2007).

[17] Antoni Mazurkiewicz. 1985. Semantics of concurrent systems: a modular fixed-point trace approach. In *Advances in Petri Nets 1984*. Springer, 353–375.

[18] Antoni Mazurkiewicz. 1995. Introduction to trace theory. In *The Book of Traces*. World Scientific, 3–41.

[19] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. *A proof of correctness for Egalitarian Paxos.* Technical Report. Technical report, Parallel Data Laboratory, Carnegie Mellon University.

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

[21] Diego Ongaro and John K Ousterhout. 2014. In search of an understandable consensus algorithm.. In *USENIX Annual Technical Conference*. 305–319.

[22] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks.. In *NSDI*. 43–57.

[23] Denis Rystsov. 2018. CASPaxos: Replicated State Machines without logs. *arXiv preprint arXiv:1802.07000* (2018).
[24] Pierre Sutra and Marc Shapiro. 2011. Fast genuine generalized consensus. In *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE, 255–264.
[25] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *ACM Computing Surveys (CSUR)* 47, 3 (2015), 42.

A FORMALIZING THE BIPARTISAN PAXOS PROTOCOLS

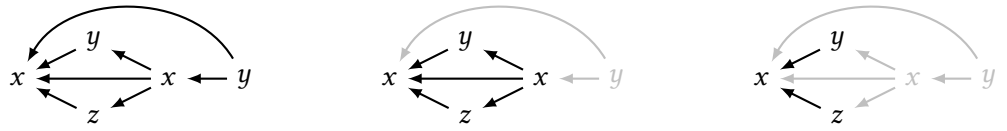
In this section, we formalize the BPaxos protocols by way of generalized consensus on conflict graphs. We begin with a review of conflict graphs and generalized consensus and then formalize BPaxos graphs and the BPaxos protocols.

A.1 Conflict Graphs and Mazurkiewicz Traces

Consider a set *Cmd* of commands and a binary reflexive relation D over *Cmd*. We say two commands $x, y \in Cmd$ **conflict** if $(x, y) \in D$, and we say they are **independent** otherwise. A **conflict graph** [18] (with respect to *Cmd* and D) is a directed acyclic graph $C = (V, E, \varphi)$ where

- V is a set of vertices;
- $E \subseteq V \times V$ is a set of edges;
- $\varphi : V \to Cmd$ is a function that labels every vertex with a command; and
- for every pair of vertices $v_1, v_2 \in V$, there exists an edge between v_1 and v_2 if and only if $\varphi(v_1)$ and $\varphi(v_2)$ conflict.

We say $C' = (V', E', \varphi|_{V'})$ is a **suffix** of C if C' is a subgraph of C such that for every edge $(v_1, v_2) \in E$, if $v_1 \in V'$, then $(v_1, v_2) \in E'$. An example conflict graph is shown in Figure 5a with $Cmd = \{x, y, z\}$ and $D = \{(x,x), (x,y), (y,x), (x,z), (z,x)\}$. Two suffixes of this conflict graph are shown in Figure 5b and Figure 5c.



(a) A conflict graph C **(b) A suffix of** C **(c) Another suffix of** C

Figure 5

We associate every conflict graph with the set of command strings that can be obtained by a reverse topological sort of the conflict graph. For example, the conflict graph in Figure 5a can be reverse topological sorted in two ways, yielding the two command strings $xyzxy$ and $xzyxy$. Notice that these two command strings can be obtained from one another by interchanging their second and third commands, two commands that do not conflict. This is true in general. Any two command strings associated with a conflict graph can be obtained from the other by repeatedly interchanging adjacent independent commands. These sets of command strings are known as Mazurkiewicz traces [17, 18] and formalize the orders in which replicated state machines can execute commands while remaining in sync.

A.2 Generalized Consensus

Generalized consensus [8, 24] involves a set of processes known as **learners** attempting to reach consensus on a growing value. Though generalized consensus is defined in terms of an abstract data structure known as a command-structure set, we restrict our attention to generalized consensus on conflict graphs. More formally, given a set *Cmd* of commands and conflict relation D, we consider a set $l_1, l_2, \ldots, l_n$ of learners where each learner l_i manages a conflict graph C_i. Over time, a set of client processes propose commands,

and learners add the proposed commands to their conflict graphs such that the following four conditions are maintained.

- **Nontriviality:** The vertices of every conflict graph are labelled only with proposed commands.
- **Stability:** Every conflict graph C_i at time t is a suffix of C_i at any time after t.
- **Consistency:** For every pair of conflict graphs C_i and C_j, there exists a conflict graph C such that C_i and C_j are both suffixes of C.
- **Liveness:** If a command is proposed, then eventually every conflict graph contains it.

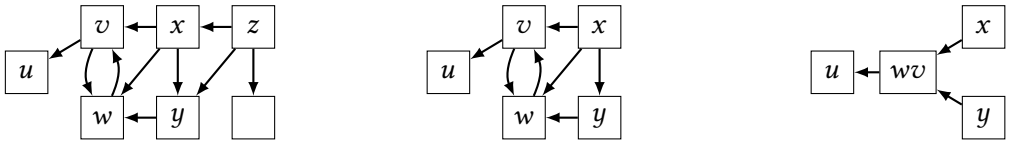
A.3 BPaxos Graphs and Partial BPaxos Graphs

Consider a set *Cmd* of commands and a symmetric conflict relation D. A **BPaxos graph** (with respect to *Cmd* and D) is a directed graph $B = (V, E, \varphi)$ where

- V is a set of vertices;
- $E \subseteq V \times V$ is a set of edges;
- $\varphi : V \to Cmd$ is a function that labels every vertex with a command; and
- for every pair of vertices $v_1, v_2 \in V$, there exists an edge between v_1 and v_2 if (but not only if) $\varphi(v_1)$ and $\varphi(v_2)$ conflict.

Intuitively, a BPaxos graph is a potentially cyclic conflict graph that can have spurious edges between vertices labelled with non-conflicting commands.

A **partial BPaxos graph** $B = (V, E, \varphi)$ is a BPaxos graph except that $\varphi : V \rightharpoonup Cmd$ is partial. Intuitively, a partial BPaxos graph is a BPaxos graph for which the labels of some vertices are unknown. We say a vertex v in a partial BPaxos graph is **eligible** if v and all vertices reachable from v are labelled. The **eligible suffix** of a partial BPaxos graph B is the suffix of B consisting of all eligible vertices. An example partial BPaxos graph is illustrated in Figure 6a, and its eligible suffix is shown in Figure 6b. In this example, $Cmd = \{u, v, w, x, y, z\}$ and $D = \{(v,u), (u,v), (v,w), (w,v), (x,v), (v,x), (x,w), (w,x), (y,w), (w,y), (z,x), (x,z), (z,y), (y,z)\}$. Note that B in Figure 6a has an edge between commands x and y even though x and y do not conflict. This is a spurious edge and is allowed in a partial BPaxos graph but not in a conflict graph. Also note that command z has an edge to a vertex that is unlabelled.



(a) A partial BPaxos graph B **(b) The eligibile suffix B' of B** **(c) The condensation of B'**

Figure 6

The **condensation** of BPaxos graph B is the graph obtained by first removing spurious edges between non-conflicting vertices in B and then by contracting every strongly connected component. Every strongly connected component labelled with commands $x_1, \ldots, x_n$ is replaced with a single vertex labelled with a command string $x_{i_1} x_{i_2} \cdots x_{i_n}$ that is obtained from an arbitrary but fixed ordering of the commands $x_1, \ldots, x_n$. The condensation of a BPaxos graph with respect to *Cmd* and D is a conflict graph with respect to Cmd^+ and D^+ where Cmd^+ is the set of non-empty command strings and where $(x_1 x_2 \cdots x_n, y_1 y_2 \cdots y_m) \in D^+$ if there is some x_i and y_j that conflict in D. An example condensation is shown in Figure 6c. Note that the spurious edge between x and y was pruned; that the strongly connected component consisting of commands v and w was contracted into a single vertex labelled with the command string wv; and that the condensation is a conflict graph.

A.4 Problem Description

The BPaxos protocols implement state machine replication by way of generalized consensus. We assume an asynchronous network model, deterministic execution, and fail-stop failures, i.e., nodes can fail by crashing

but cannot act maliciously. Throughout the paper, we assume at most f processes can fail. We consider a set $b_1, b_2, \ldots, b_{f+1}$ of deterministic state machine replicas that all start in the same initial state. We consider a set Cmd of state machine commands and define a conflict relation D such that two commands $x, y \in Cmd$ conflict if they do not commute.

Every BPaxos protocol is a generalized consensus protocol, with state machine replicas playing the role of learners. Each replica b_i learns a conflict graph C_i with respect to Cmd^+ and D^+. Every replica b_i executes the commands in C_i in reverse topological order as they are learned. For example, if a replica has learned the conflict graph shown in Figure 6c, it can execute commands in the order $uwvxy$ or $uwvyx$. Executing commands in this way, replicas are guaranteed to stay in sync, producing identical responses for every command they execute.

A.5 The BPaxos Protocols

Every BPaxos protocol implements generalized consensus by first reaching consensus on a partial BPaxos graph. When a client process sends a state machine command x to a BPaxos protocol, a process implementing the protocol eventually assigns the command to an instance I and a set of dependencies $\text{deps}(I)$. Instances are called instances because the BPaxos protocol implements one instance of consensus for every instance I. That is, the BPaxos protocol eventually reaches consensus on the value $(x, \text{deps}(I))$ in instance I. Every state machine replica b_i maintains a partial BPaxos graph B_i where instances play the role of vertices. When replica b_i learns that an instance I has been chosen with value $(x, \text{deps}(I))$, it adds vertex I to B_i labelled with x and with outbound edges to every instance in $\text{deps}(I)$ (adding previously unseen instances to B_i as necessary). Letting C_i be the condensation of the eligible suffix of B_i, the BPaxos protocol successfully implements generalized consensus with replica b_i learning conflict graph C_i so long as Invariant 1 and Invariant 2 are maintained.

Invariant 1 allows replicas to reach consensus on their partial BPaxos graphs. Invariant 2 ensures that every replica's partial BPaxos graph really is a partial BPaxos graph. These two invariants suffice to ensure stability and consistency for the following reasons. Stability is guaranteed because a replica b_i's partial BPaxos graph B_i (and hence its condensation C_i) only grows over time. Consistency is guaranteed because every condensation C_i is a suffix of the condensation of the eligible suffix of the global partial BPaxos graph, the graph formed from complete knowledge of every chosen instance. Moreover, the BPaxos protocols do provide liveness with certain assumptions about the niceness of the network, but we do not focus on liveness in this paper. Finally, every BPaxos protocol ensures nontriviality in a straightforward way.

B FAST PAXOS

B.1 Standard Fast Paxos

Fast Paxos [10] is a two-phase consensus algorithm structured around a set of clients, leaders, acceptors, and learners. For a full description of Fast Paxos, we refer the reader to [10], but we highlight the most relevant bits of Fast Paxos here. Fast Paxos proceeds in a series of integer-valued rounds with 0 being the smallest round and -1 being a null round. Every round is classified either as a fast round or as a classic round. In phase 2a of Fast Paxos, a leader has to choose a value to send to the acceptors. The logic for choosing this value is shown in Algorithm 1 where $O4(v)$ is true if there exists a fast quorum $\mathcal{F}$ of acceptors such that every acceptor in $\mathcal{F} \cap \mathcal{Q}$ voted for v in round k. The sizes of fast and classic quorums ensure that at most one value $v \in V$ satisfies $O4(v)$ in any given round. For example, Fast Paxos is commonly deployed with $n = 2f + 1$ acceptors, classic quorums of size $f + 1$, and fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor + 1$. In this case, a value $v \in V$ satisfies $O4(v)$ only if a set $\mathcal{A} \subseteq \mathcal{Q}$ of $\lfloor \frac{f+1}{2} \rfloor + 1$ or more acceptors voted for v in round k. Because $\mathcal{A}$ constitutes a majority of $\mathcal{Q}$, at most one $v \in V$ can satisfy $O4(v)$.

To prove the correctness of Fast Paxos, it suffices to prove the statement $P(i)$ that says that if an acceptor votes for a value v in round i, then no value besides v has been or will be chosen in any round j less than i. We prove this claim by induction. $P(0)$ is trivial because there are no rounds less than 0. For the general case, we perform a case analysis on Algorithm 1, noting that an acceptor will only vote for value v in round

i if some leader proposes it (or proposes distinguished value *any*). First, assume $k \neq -1$. We perform a case analysis on j.

- **Case $k < j < i$.** If $k < j < i$, then no value has been or will be chosen in round j because the phase 1 quorum Q of acceptors did not vote in any round larger than k and promised not to vote in any round less than i. In order for a value w to get chosen in round j, it must get a quorum Q_w of acceptors to vote for it, but Q and Q_w intersect, so no value can be chosen in round j.
- **Case $k = j$.** We perform a case analysis on whether a leader executes line 7, line 9, or line 11.
  - **Case line 7.** If $V = \{v\}$, then the quorum Q of acceptors have either voted for v in round k (and hence will never vote again in round k) or promised to never vote in round k. Thus, no other value w can receive a quorum Q_w of votes in round k.
  - **Case line 9.** If k is a classic round, then $V = \{v\}$. Thus, by virtue of not executing line 7, we know that k is a fast round in this case. In order for a value v to be chosen in fast round k, v needs to receive phase 2b votes from some fast quorum $\mathcal{F}$ of acceptors. Thus, there needs to exist some $\mathcal{F}$ such that every acceptor in $\mathcal{F} \cap Q$ voted for v in round k. By construction of fast quorums, there can exist at most one such value, so no value other than v could have been chosen in round k.
  - **Case line 11.** In this case, k is a fast round. $O4(v)$ is a necessary condition for value v to be chosen in round k, so if no such v exists, then no value could have been chosen in round k. Thus, the leader is free to propose any value.
- **Case $j < k$.** We again perform a case analysis on whether a leader executes line 7, line 9, or line 11.
  - **Case line 7.** By $P(k)$, no value other than v has been or will be chosen in round j.
  - **Case line 9 or line 11.** Let $v_1, v_2 \in V$. By $P(k)$ applied to v_1 and $P(k)$ applied to v_2, no value has been or will be chosen in round j.

Finally, if $k = -1$, then we know that no value has been or will be chosen in any round less than i by the same line of reasoning as the first case above, so the leader is free to propose anything. This corresponds to line 5.

B.2 Fast Paxos Tweak

If we assume that round 0 is a fast round and every other round is a classic round—as we do in the BPaxos protocols—then we can simplify the standard phase 2a algorithm shown in Algorithm 1 to the variant shown in Algorithm 2. The proof of this tweak's correctness is a simplification of the Fast Paxos proof given above. We again prove $P(i)$ by induction. $P(0)$ is still trivial. For the general case, we again perform a case analysis on j. First, assume $k \neq -1$.

- **Case $k < j < i$ or $k = j$.** The proof for these two cases is identical to the Fast Paxos proof above.
- **Case $j < k$.** We perform a case analysis on whether a leader executes line 7, line 9, or line 11.
  - **Case line 7.** By $P(k)$, no value other than v has been or will be chosen in round j.
  - **Case line 9 or line 11.** $k = 0$, and there are no rounds less than 0, so this case holds trivially.

Again, if $k = -1$, then we know that no value has been or will be chosen in any round less than i by the same line of reasoning as the first case above.

C UNSAFE BPAXOS

In this section, we give a concrete execution of Unsafe BPaxos that illustrates its lack of safety. Consider an Unsafe BPaxos deployment with $f = 2$ and $n = 2f + 1 = 5$. Assume Unsafe BPaxos node b_1 receives command x from a client and Unsafe BPaxos b_2 receives a conflicting command y from a client. b_1 sends (I_x, x) to the dependency service, and b_2 sends (I_y, y) to the dependency service. Dependency service nodes d_1 and d_2 receive (I_x, x) and propose $(x, \emptyset)$ in instance I_x to Fast Paxos acceptors a_1 and a_2. Similar, d_4 and d_5 receive (I_y, y) and propose $(y, \emptyset)$ in instance I_y to acceptors a_4 and a_5. Then, b_1 and b_2 crash and all other messages are dropped. Unsafe BPaxos node b_3 then attempts to recover I_x. In phase 1 of Fast Paxos, b_3 receives phase 1b messages from a_1, a_2, and a_3. Because a_1 and a_2 both voted for the value $(x, \emptyset)$ in round 0, b_3 is forced to propose the value $(x, \emptyset)$ in phase 2 (line 6 of Algorithm 1). Assume this value gets chosen.

Then, b_3 recovers I_y. In phase 1 of Fast Paxos, b_3 receives phase 1b messages from a_3, a_4, and a_5. a_4 and a_5 both voted for the value $(y, \emptyset)$ in round 0, so b_3 is forced to propose the value $(y, \emptyset)$ in phase 2 (line 6 of Algorithm 1). Assume this value gets chosen. Now, instances I_x and I_y have both been chosen with conflicting commands x and y, but neither instance depends on the other. This violates Invariant 2, and can result in two replicas executing conflicting commands in different orders.

D UNANIMOUS BPAXOS

In this section, we prove that Unanimous BPaxos maintains Invariant 4. If a value $v = (x, \text{deps}(I))$ is chosen in round 0 of instance I, then every single acceptor voted for v in round 0. An acceptor a_j only votes for a value v in round 0 if its co-located dependency service node d_j proposed it. Thus, every single dependency service node proposed $(x, \text{deps}(I))$, so $(I, x, \text{deps}(I))$ is a valid response from the dependency service.

Otherwise, $v = (x, \text{deps}(I))$ is chosen in round $i > 0$. To prove Unanimous BPaxos maintains Invariant 4 in this case, we prove the claim $P(i)$ for $i > 0$ that says if a Fast Paxos acceptor votes for value $(x, \text{deps}(I))$ in round i of instance I, then either $(I, x, \text{deps}(I))$ is a response from the dependency service, or $(x, \text{deps}(I)) = (\text{noop}, \emptyset)$. We prove $P(i)$ by induction, noting that in order for v to be chosen in round i, a Unanimous BPaxos node b_j must have proposed v in round i. Thus, we perform a case analysis on Algorithm 2, the logic that b_j uses to select the value v.

- **Case line 5.** In this case, b_j either chooses to propose a noop or propose a response from the dependency service, so $P(i)$ holds trivially.
- **Case line 7.** In this case, $V = \{(x, \text{deps}(I))\}$ is a singleton set, and $P(i)$ holds directly from $P(k)$.
- **Case line 9.** In order for a value $v = (x, \text{deps}(I))$ to be chosen in round 0, every single acceptor must have voted for v in round 0. Thus, every acceptor in Q voted for v in round 0. Thus, every dependency service node co-located with an acceptor in Q proposed $(x, \text{deps}(I))$, so $(I, x, \text{deps}(I))$ is a valid response from the dependency service.
- **Case line 11.** In this case, b_j either chooses to propose a noop or propose a response from the dependency service, so $P(i)$ holds trivially.

E DEADLOCK BPAXOS

E.1 Pruned Dependencies

Here, we prove that Invariant 3 and Invariant 5 imply Invariant 2. Consider two conflicting commands x and y chosen in instances I_x and I_y. x and y conflict, so neither is noop. Thus, by Invariant 5, I_x is chosen with pruned dependencies $\text{deps}(I_x) - P_x$ and I_y is chosen with pruned dependencies $\text{deps}(I_y) - P_y$ derived from dependencies $\text{deps}(I_x)$ and $\text{deps}(I_y)$ computed by the dependency service. We want to show that $I_x \in \text{deps}(I_y) - P_y$ or $I_y \in \text{deps}(I_x) - P_x$, or both. By Invariant 3, either $I_y \in \text{deps}(I_x)$ or $I_x \in \text{deps}(I_y)$ or both. Without loss of generality, assume $I_y \in \text{deps}(I_x)$. If I_y is also in $\text{deps}(I_x) - P_x$, then we're done. Otherwise, I_y has been pruned from $\text{deps}(I_x)$ (i.e. $I_y \in P_x$). This happens only if y is a noop or if I_y has been chosen with $I_x \in \text{deps}(I_y) - P_y$. y is not a noop, so $I_x \in \text{deps}(I_y) - P_y$.

E.2 Deadlock Example

Consider a Deadlock BPaxos deployment with $n = 9$ dependency service nodes (i.e., $f = 4$). Imagine four of these nodes have failed. The conflict graphs of the remaining five ordering service nodes are illustrated in Figure 7. All five dependency service nodes have processed five commands x_0 through x_4 in instances I_0 through I_4 respectively. Every command x_i conflicts with $x_{i-1 \bmod 5}$ and $x_{i+1 \bmod 5}$. For example, x_0 conflicts with x_4 and x_1. Dependency service node d_1 has processed the five commands in the order x_0, x_1, x_2, x_3, x_4, dependency service node d_2 has processed the five commands in the order x_1, x_2, x_3, x_4, x_0, and so on.

Imagine Deadlock BPaxos node b_i attempts to recover instance I_0. $\lfloor \frac{f+1}{2} \rfloor + 1$ acceptors have voted for $\text{deps}(x_0) = \{x_4\}$, but d_2 voted for $\{x_1, x_4\}$. Thus, b_i attempts to recover I_1. $\lfloor \frac{f+1}{2} \rfloor + 1$ acceptors have voted for $\text{deps}(x_1) = \{x_0\}$, but d_3 voted for $\{x_0, x_2\}$, so b_i attempts to recover I_2. This continues until b_i attempts

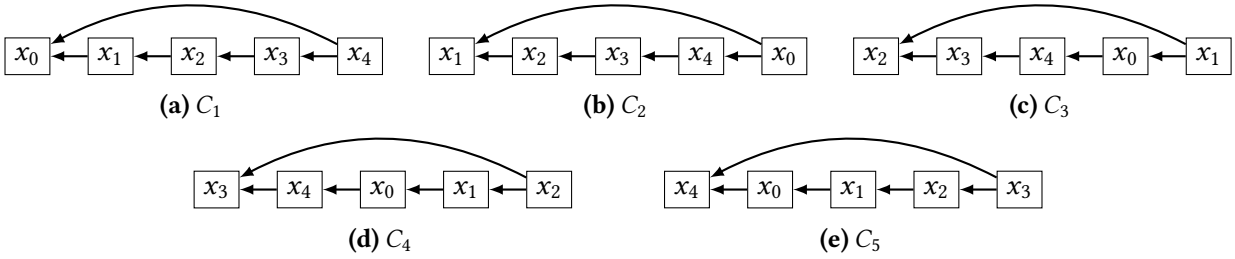


Figure 7: A Deadlock BPaxos deadlock

to recover I_4. $\lfloor \frac{f+1}{2} \rfloor + 1$ acceptors have voted for $\text{deps}(x_4) = \{x_3\}$, but d_1 voted for $\{x_0, x_3\}$, so b_i attempts to recover I_0. This is a deadlock.

F COMPARING EPAXOS AND BPAXOS

Similarities. The BPaxos protocols and EPaxos share a lot in common. This is no accident. The BPaxos protocols are heavily inspired by EPaxos. Here, we list some of the similarities.

- EPaxos nodes construct directed graphs of commands and execute commands in reverse topological order one strongly connected component at a time. BPaxos borrows this execution model directly, formalizing it using notions of generalized consensus, dependency graphs, eligible suffixes, and condensations.
- EPaxos also maintains Invariant 1 and Invariant 2, the two invariants core to the BPaxos protocols.
- Moraru et al. present two EPaxos variants in [20]. The basic EPaxos protocol is a simpler version of EPaxos with large fast quorum sizes, while the full EPaxos protocol is a much more complex version of EPaxos with smaller quorum sizes and sophisticated recovery. This parallels our presentation of a simple protocol with large quorum sizes (Unanimous BPaxos) and a more complex protocol with smaller quorum sizes (Majority Commit BPaxos).

Differences. Here, we list some of the fundamental differences between EPaxos and the BPaxos protocols.

- EPaxos requires at least three message delays to commit a command, whereas Unanimous BPaxos and Majority Commit BPaxos only require two (the theoretical minimum).
- The BPaxos protocols considers a value v chosen on the fast path if there exists a fast quorum of acceptors that voted for v in round 0. EPaxos, on the other hand, considers a value v chosen on the fast path in instance I only if the leader of instance I receives a fast quorum of votes for v in round 0. That is, in EPaxos, an instance's leader has the ultimate authority on whether a value is chosen on the fast path.
- Because of the previous two differences, EPaxos allows for smaller fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor$ compared to Majority Commit BPaxos' fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor + 1$. These smaller fast quorums come at the cost of increased commit latency.
- Majority Commit BPaxos considers a value $v = (x, \text{deps}(I))$ chosen on the fast path only if there exists some fast quorum $\mathcal{F}$ of acceptors that voted for v in round 0 and if for every $I' \in \text{deps}(I)$, there exists a quorum $Q' \subseteq \mathcal{F}$ of acceptors that know I' is chosen. EPaxos has a similar condition, but only requires that one acceptor in $\mathcal{F}$ know that I' is chosen, rather than a quorum.
- The BPaxos protocols are more modular than EPaxos, making them easier to understand. All three of the BPaxos protocols are composed of a set of BPaxos nodes, a dependency service, and a consensus service. This allows us to understand the BPaxos protocols piece by piece. EPaxos also has notions of consensus and dependency computation, but they are tightly coupled. Moreover, the BPaxos protocols heavily leverage Fast Paxos as a consensus subroutine. EPaxos does leverage some elements of Paxos,

but it does not rely on an existing consensus protocol for correctness. Thus, proving the correctness of EPaxos requires proving that EPaxos successfully implements consensus.