Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus

Erica Blum*
Reed College
Portland, OR, United States
ericablum@reed.edu

Jonathan Katz
University of Maryland
College Park, MD, United States
jkatz2@gmail.com

Julian Loss
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
loss@cispa.de

Kartik Nayak
Duke University
Durham, NC, USA
kartik@cs.duke.edu

Simon Ochsenreither
Saarland University
Saarbrücken, Germany
s.ochsenreither@gmail.com

ABSTRACT

Protocols for state-machine replication (SMR) often trade off performance for resilience to network delay. In particular, protocols for asynchronous SMR tolerate arbitrary network delay but sacrifice throughput/latency when the network is fast, while partially synchronous protocols have good performance in a fast network but fail to make progress if the network experiences high delay. Existing *hybrid* protocols are resilient to arbitrary network delay and have good performance when the network is fast, but suffer from high overhead ("thrashing") if the network repeatedly switches between being fast and slow, e.g., in a network that is typically fast but has intermittent message delays.

We propose *Abraxas*, a generic approach for constructing a hybrid protocol from any "fast" protocol Π_{fast} and asynchronous protocol Π_{slow} to achieve (1) security and performance equivalent to Π_{slow} under arbitrary network behavior, and (2) performance equivalent to Π_{fast} when conditions are favorable. We instantiate Abraxas with the best existing protocols for Π_{fast} (Jolteon) and Π_{slow} (2-chain VABA), and show experimentally that the resulting protocol significantly outperforms Ditto, the previous state-of-the-art hybrid protocol.

KEYWORDS

State Machine Replication, Asynchrony, Hybrid Protocol

1 INTRODUCTION

Protocols for state-machine replication (SMR) [18, 22, 24] form the core of distributed ledger technologies or blockchains. SMR allows a distributed set of parties to agree on an unbounded, ordered sequence (i.e., a *chain*) of *blocks*, each of which contains some predetermined number of *transactions*. Security for SMR in the presence of some fraction of malicious parties requires two fundamental properties: *consistency* and *liveness*. Consistency requires that all honest parties agree on any blocks they output. Liveness guarantees progress, in the sense that if all honest parties hold some transaction as input, then that transaction will eventually be included in some block output by those parties.

Resilience to network delays vs. performance. The SMR literature primarily considers three network settings: synchronous, partially synchronous, and asynchronous. In a synchronous network it is assumed that all messages are delivered within some fixed (known) time Δ after they are sent. This allows tolerating up to $t < n/2$ malicious parties. In practice, however, it is often difficult to guarantee complete synchrony. This has motivated the design of partially synchronous [7, 9, 10, 25] and asynchronous [15, 19**? ?** , 20] protocols, which can tolerate only $t < n/3$ corrupted parties.

In the partially synchronous setting, the network is initially asynchronous but is guaranteed to become fully synchronous after some unknown (but finite) *global stabilization time* (GST). This assumption allows for highly efficient *deterministic* protocols by relying on a *leader* who is responsible for driving the protocol's progress. For example, after the GST such protocols can confirm blocks within three rounds of communication if the leader is not faulty [10]. Moreover, in that case the latency and throughput of such protocols can be strictly superior to those of synchronous protocols, as they run in time depending on the actual network latency δ rather than the (often pessimistic) worst-case synchrony parameter Δ. However, before the GST or when the leader is malicious, these protocols need to invoke a relatively expensive leader-rotation sub-protocol. In those cases, partially synchronous protocols may perform repeated leader rotations during which no progress is made.

In the asynchronous setting, messages may be delayed for arbitrarily long periods of time, with the only guarantee being that they are eventually delivered. Any secure asynchronous protocol must be randomized. This, unfortunately, represents a major bottleneck: indeed, compared to their (deterministic) partially synchronous counterparts, state-of-the-art asynchronous protocols usually require far more rounds to output a block (e.g., > 10 rounds in expectation [2, 15]). This results not only in high latency, but also poor throughput since parties do not start processing the next block until the previous block has been confirmed.

Achieving the best of both. Ideally, an SMR protocol would be secure even under worst-case conditions, while having good performance under good conditions. With this goal in mind, Kursawe

*Portions of this work were performed while at University of Maryland.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623191

and Shoup initiated a line of work on so-called *hybrid* (or *optimistic*) protocols [17, 23]. Such protocols have a deterministic "fast path" that makes progress quickly when conditions are favorable (e.g., when the leader is honest and the network is fast), as in partially synchronous protocols, as well as a fallback "slow path" that guarantees liveness when conditions are not favorable, as in asynchronous protocols. By alternating between these paths, hybrid protocols can achieve high efficiency as well as strong security. The technique was later studied by Aublin et al. [4], who introduced a generic framework for constructing hybrid protocols; and Pass and Shi [21], who further refined and popularized the hybrid technique in their Thunderella protocol. In the last few years, hybrid protocols have continued to receive intense interest in both the asynchronous [13, 19] and synchronous [1, 3, 21] settings.

One drawback of existing hybrid protocols in the asynchronous setting is that their performance suffers when they fall back frequently to the slow path. In particular, most hybrid protocols first attempt to run a fast path, and fall back to a slow path only after some generous time-out period (as otherwise the protocol might fall back unnecessarily). Thus, even the best hybrid protocols [13, 19] are significantly less efficient than purely asynchronous protocols under worst-case network conditions. This unsatisfactory state of affairs raises a natural question: *Are there asynchronous hybrid protocols whose performance is comparable to the best partially synchronous protocols when conditions are favorable, and to the best asynchronous protocols when conditions are unfavorable?*

DAG-based protocols. Recent DAG-based protocols [11, 14, 16] partially answer this question, as they maintain stable throughput and latency even as network conditions fluctuate. DAG-based protocols proceed in steps, during each of which all n parties propose $O(n)$ blocks in parallel (a *layer*), and every proposal points back to multiple blocks in a previous layer. An advantage with this approach is that they decouple liveness of the protocol from transaction broadcast which allows them to commit many blocks at once when they recover from an intermittent liveness loss. A problem with these DAG-based approaches, however, is that they require $\Omega(n^3)$ total communication per layer, which means they induce high network congestion when the number of parties grows large. Furthermore, due to the need to store n blocks per layer—in addition to all the pointers to blocks in previous layers—the memory footprint for parties running DAG-based protocols is $\Omega(n^2)$.

1.1 Our Contribution: Abraxas

Addressing the above, we propose a compiler called *Abraxas*[1] that provides a generic way of using any SMR protocol Π_{fast} as the fast path along with any asynchronous SMR protocol Π_{slow} as the slow path. Abraxas is secure (under arbitrary network conditions) as long as Π_{slow} is secure; it has performance comparable to Π_{fast} when network conditions are favorable, and performance comparable to Π_{slow} when conditions are unfavorable. Importantly, Abraxas maintains stable throughput even when it switches between its fast and slow paths as network conditions change. In contrast, the throughput of prior hybrid protocols in that case would fall below what the slow path could ensure on its own because those protocols alternate between the slow and fast paths. Instead, we design Abraxas so that *the slow path is always running in the background*, even while the protocol is utilizing the fast path, and even during leader rotation. (For this reason, Abraxas does not fit into the general framework of Aublin et al. [4].) Thus, Abraxas continues to make progress even while switching between the paths. This, in particular, complicates a proof of consistency, since parties may have inconsistent views about what was output on the fast path when the protocol falls back to the slow path. Thus, the main technical challenge of our construction is to maintain consistency between the two paths in this situation. We refer to Section 1.2 for further details of how we accomplish that.

Although the DAG-based Bullshark protocol [14] also achieves good performance under both favorable and unfavorable network conditions, it suffers from the drawback of requiring $\Omega(n^3)$ communication per layer of n committed blocks (as mentioned above). In contrast, instantiating Abraxas with state-of-the-art components (see below) gives a protocol that uses only $O(n^2)$ communication per committed block. Although the communication complexity *per block* is asymptotically the same for both approaches, the former approach will lead to network congestion more quickly as n increases, resulting in roughly the same throughput but with significantly higher bandwidth requirements in order to maintain equivalent latency. Additionally, Bullshark requires $\Omega(n^2)$ memory usage per party (as in all DAG-based protocols), whereas Abraxas* only requires $O(n)$ memory (since votes in our protocol can be stored as aggregated certificates).

Evaluation. We evaluate Abraxas*, an instantiation of Abraxas using current state-of-the-art sub-protocols for Π_{fast} and Π_{slow} (namely, Jolteon [13] and 2-chain VABA [13], respectively). Table 1 compares the theoretical performance of Abraxas* with the state-of-the-art hybrid protocols Bolt-Dumbo [19] and Ditto [13], as well as Bullshark [14], a leading DAG-based protocol. Focusing on the latency, we see that in good conditions (i.e., a synchronous network with a good leader) Abraxas* performs as well as Ditto and Bolt-Dumbo, whereas in bad network conditions Abraxas* performs as well as Ditto and significantly better than Bolt-Dumbo. We stress, however, that these two extremes do not tell the whole story; a key advantage of Abraxas* is that it performs well *even when conditions repeatedly switch between good and bad* (something we refer to as *throughput stable* in Table 1), as justified by the experiments we describe next. (For completeness, we note that the DAG-based Bullshark protocol is also throughput stable, but incurs cubic communication complexity as discussed earlier.)

Specifically, our experiments show that Abraxas* matches or outperforms Ditto [13], a state-of-the-art hybrid protocol for the asynchronous setting. We evaluate performance when the leader crashes 0%, 5%, 10%, 20%, and 100% of the time. Different leader-failure rates simulate the effect of different network conditions, since the effect of a failed leader is the same as the effect of a good leader with a poor network connection. Thus, a 0% leader-failure rate corresponds to a stable, fast network with a responsive leader, which is ideal for partially synchronous protocols and the fast path of hybrid protocols. At the other extreme, a 100% leader-failure rate causes hybrid protocols to rely entirely on the slow path, and emulates a slow network with a DDoS'd leader, where

[1] Abraxas is a mythical figure often depicted on charms and amulets as a hybrid of multiple creatures.

Table 1: Comparison with state-of-the art hybrid and DAG-based protocols. We call a protocol *throughput stable* if its throughput remains good under all network conditions.

Protocol	Communication (fast network)	Communication (slow network)	Latency (fast network)	(Expected) latency (slow network)	Throughput stable?
Bolt-Dumbo [19]	$O(nB)$	$O(nB + n^3\kappa))$	5	28	No
Ditto [13]	$O(nB)$	$O(n^2B)$	5	10.5	No
Bullshark [14]	$O(n^3 + n^2B)$	$O(n^3 + n^2B)$	2	6	Yes
Abraxas*	$O(n^2B)$	$O(n^2B)$	5	10.5	Yes

Notes: Communication is the number of bits needed to commit a block, where κ is a computational security parameter and $B = \Omega(\kappa)$ is the blocksize; note that Bullshark commits n blocks at a time. Latency is measured in rounds; the latencies of Ditto and Abraxas* are amortized (i.e., we display the expected rounds to commit k blocks divided by k for $k \to \infty$). Both Bolt-Dumbo and Abraxas* are generic frameworks and so can be instantiated with various paths to achieve different communication complexities and latencies.

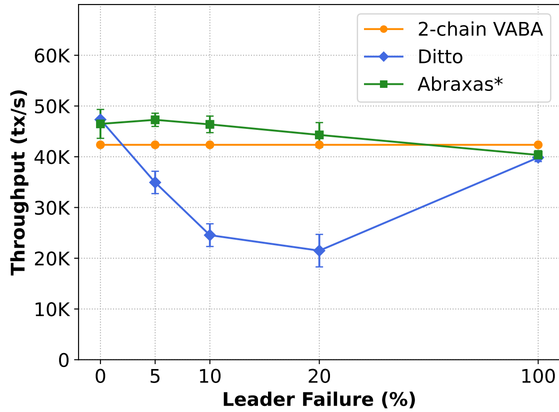


Figure 1: Throughput under varying probabilities of leader failure for $n = 32$ parties.

asynchronous protocols are better. The remaining cases correspond to a fluctuating network that forces continual switching between the fast and slow paths.

Figure 1 (cf. Section 6) summarizes the results of one set of experiments comparing the throughput of Abraxas* and Ditto (instantiated with the same sub-protocols as Abraxas*) at each protocol's saturation point. Although the throughputs of Abraxas* and Ditto are comparable at the two extremes, it is evident that Abraxas* performs much better under a moderate failure rate, i.e., whenever there is any switching (even if it is relatively infrequent) between good and bad network conditions. The latency of Abraxas* is also significantly better than that of Ditto in those cases; we refer to Section 6 for further details.

Figure 1 also includes results for VABA, a leading asynchronous protocol that always runs on a "slow path" and does not take advantage of a fast network. It is interesting to note that the throughput of VABA is comparable to—though still generally worse than—that of Abraxas*, except at the 100% failure rate that an asynchronous protocol is designed for. But precisely because VABA always runs a "slow path," the latency of VABA is roughly 3× worse than that of Abraxas* when run in a fast network (see Section 6).

1.2 Overview of Abraxas

Abraxas incorporates two sub-protocols Π_{fast} and Π_{slow} that run on what we call the fast path and slow path, respectively, and that each maintain their own chain C_{fast} and C_{slow}. (These chains are only maintained internally by the parties; Abraxas itself also maintains a *main chain* with the blocks that the parties actually output.) Π_{fast} outputs blocks quickly (i.e., has low latency) in a synchronous network; however, it may fail to make any progress in an asynchronous network (and may remain permanently stuck even if the network later becomes synchronous). Π_{slow} has worse latency but guarantees liveness even in a fully asynchronous network.

At a high level, our protocol works as follows. Like many hybrid protocols, Abraxas has a steady state and a recovery state; however, unlike most hybrid protocols, in Abraxas the slow path is run at all times (even during the steady state). During the steady state, the fast path drives consensus, and the slow path is used only to detect problems with the fast path. Informally, if the fast path is unable to "keep ahead" of the slow path, then the fast path is no longer working as it should, and parties switch to the recovery state. Defining a useful notion of "keeping ahead" turns out to be a key technical challenge; we explore this point in greater detail in Section 3.

During the recovery state, parties stop running the fast path altogether, and use the slow path to drive consensus. Informally, in the recovery state parties use C_{slow} to agree on how many blocks of the (now defunct) fast chain should be output on the main chain. The main challenge is ensuring that the agreed-upon number of blocks is high enough to include any blocks output by an honest party during the steady state. (This is a common challenge in hybrid protocols; the Bolt-Dumbo Transformer [19], for example, overcomes this challenge using a specialized subprotocol called tcv-BA.) Once parties have agreed on which (if any) blocks from C_{fast} they should output on the main chain, they additionally take advantage of any extra progress the slow chain may have made in the meantime by placing all outstanding transactions (up to a certain index) in a fresh block, and outputting that block on the main chain. At that point, the parties return to the steady state and begin generating a fresh instance of C_{fast}.

Compared to other hybrid protocols instantiated with the same sub-protocols, Abraxas achieves higher throughput because it always runs the slow path in the background, instead of only running the slow path once a problem is detected. Thus, the work invested

in building the slow chain pays off each time the protocol completes a recovery phase.

2 MODEL AND DEFINITIONS

In our setting, protocols are run among a fixed set of n parties, up to t of which may be adaptively *corrupted*.[2] Corrupted parties are controlled by an adversary, who may cause them to deviate arbitrarily from the protocol. Parties who are not corrupted are called *honest*. Throughout, we assume $n \geq 3t + 1$. We consider an *asynchronous* setting, in which an adversary is allowed to delay messages for arbitrary lengths of time (subject to the restriction that messages sent by honest parties must eventually be delivered).

We are primarily interested in *state machine replication* (SMR) protocols. At a high level, such protocols allow parties (also called nodes or replicas) to agree on a growing *chain* of objects called *blocks*. Each block contains an ordered list of input values called *transactions*. Transactions are generated and added to a local buffer via some external mechanism whose details are unimportant for our purposes; we denote the local buffer belonging to P_i by buf_i.

Each party's local chain is modeled as a write-once array of blocks (denoted C). We say that *P_i outputs (or commits) block k* when P_i writes to $C[k]$. For convenience, we assume each entry of C is initialized to the special value $\perp$ at the start of the protocol, so that $C[k] = \perp$ iff P_i has not yet output block k. Furthermore, we assume a party outputs block k only after outputting blocks $1, \ldots, k-1$. Most existing blockchain protocols naturally satisfy this requirement, and in any case, one can enforce it by buffering out-of-order blocks until all earlier blocks have been output, at the cost of potentially increasing latency. Lastly, we assume a party deletes a transaction from its buffer once it observes that transaction in some block in its chain.

Different definitions of SMR exist in the literature; for concreteness, we use the following definition of SMR adapted from Blum et al. [6]:

Definition 2.1 (State Machine Replication). Let Π be a protocol executed by parties $P_1, \ldots, P_n$ who are provided with transactions as input and each locally maintain a chain C as described above.

- **Completeness:** Π is **complete** if for all $k \geq 1$, every honest party eventually outputs block k.
- **Consistency:** Π is **consistent** if for all honest parties P_i, P_j, whenever $C_i[k] \neq \perp, C_j[k] \neq \perp$ then $C_i[k] = C_j[k]$.
- **Liveness:** Π is **live** if for any transaction tx that is in all honest parties' buffers, every honest party eventually outputs a block containing tx.

If Π is complete, consistent, and live in the presence of up to t corrupted parties, then Π is ***t*-secure**.

We write $C[: k]$ as shorthand for $C[1], \ldots, C[k]$. The *length* of a chain, denoted by $\mathsf{len}(C)$, is equal to the largest index k such that $C[k] \neq \perp$.

2.1 Cryptographic Primitives

We let κ denote a cryptographic security parameter. We assume a collision-resistant hash function H with output of length $O(\kappa)$.

A k-out-of-n (non-interactive) threshold signature scheme consists of algorithms for key generation, signing, verification, and combination. We assume that prior to the protocol, a trusted dealer performs setup for the scheme, generating a public key pk, secret keys $sk_1, \ldots, sk_n$, and public verification keys $pk_1, \ldots, pk_n$. Each party P_i receives sk_i, pk, and $(pk_1, \ldots, pk_n)$. Party P_i can use its secret key sk_i to create a signature share σ_i on a message m. A signature share from party P_j on a message m can be verified using the corresponding verification key pk_j; a signature share is *valid* if the verification algorithm succeeds and *invalid* otherwise. A set of k valid signature shares on a message m can be combined into a signature σ, which can be verified using the public key pk. We assume that signature shares and signatures have length $O(\kappa)$.

We follow the standard convention for this line of work and assume that signature shares and signatures are *perfectly unforgeable*. This means that it is not possible to produce a signature share on a message m that verifies under pk_i unless P_i explicitly generates such a share (or is corrupted); similarly, an adversary who corrupts t parties cannot generate a signature on a message m that verifies under pk unless $k - t$ honest parties generate signature shares for that message.

3 ABRAXAS

Abraxas synthesizes the output of two arbitrary SMR protocols Π_{fast} and Π_{slow} running as a "fast path" and "slow path," respectively. Security of Abraxas as an asynchronous SMR protocol relies only on security of Π_{slow} in the same sense (and for the same number of corruptions), and requires no assumptions about Π_{fast}. The most interesting instantiations of Abraxas, however, are where Π_{fast} also ensures consistency (though not necessarily completeness or liveness) in an asynchronous network.

Overview of the protocol. Parties in Abraxas proceed in a sequence of *eras*, where in each era a party begins in the *steady state* (corresponding to the fast path) and switches to the *recovery state* (corresponding to the slow path) if a problem is detected; see Algorithms 1–4.

Algorithm 1: Main protocol wrapper

```
1 throughout:
2     whenever a transaction tx is input to buf_main, input tx
        to buf_fast and buf_slow;
3     whenever a block B is output to C_main, delete each
        tx ∈ B from buf_fast and buf_slow;
4 define main:
5     set ℓ* := 0, e := 1;
6     begin running Π_slow (which writes to C_slow);
7     steadyState();
```

In the steady state (cf. Algorithms 2 and 3), a fresh instance of Π_{fast} is used to drive consensus. We denote the instance associated with era e by Π^e_{fast} and the chain it maintains by C^e_{fast}, omitting the superscript when e is irrelevant or clear from context. If a party detects a problem and falls back to the recovery state (cf. Algorithms 4 and 5), it terminates Π^e_{fast} and relies on Π_{slow} to

[2] As long as the cryptographic primitives and subprotocols are adaptively secure, so is Abraxas.

drive consensus. (The same instance of Π_{slow} is run throughout the protocol, and parties continue to run Π_{slow} even while in the steady state.) Once a party decides that the original problem has been resolved, it advances to the next era.

Each party maintains local buffers buf_{fast} and buf_{slow} for the fast path and slow path, respectively. When a party receives a transaction tx in its (real) buffer buf_{main}, it places tx in both buf_{fast} and buf_{slow}. As will be discussed further below, parties will also add additional, "symbolic" transactions to buf_{slow}.

Votes and certificates. Special messages called *votes* and *certificates* play a central role in the protocol. In general, a vote is associated with a unique era, index, and block, and is signed by the party who creates the vote using an $(n-t)$-out-of-n threshold signature scheme. Any $n-t$ votes (of the same type, for the same data) can be used to form a certificate. In the steady state, *confirmation votes/certificates* are used to acknowledge new blocks on the fast chain. In the recovery state, *recovery votes/certificates* are used to agree on what progress (if any) the fast chain made during the steady state. The format of these votes/certificates is as follows:

- A confirmation vote cv contains an era e, an index k, a fast-chain block B, and a partial signature σ_i on $h := H(B)$ by the party who created the vote.
- A confirmation certificate cc contains an era e, index k, hash digest h, and a full signature σ on h.
- A recovery vote rv contains an era e, index k, a confirmation certificate cc associated with e and k, and a partial signature σ_i on cc.
- A recovery certificate rc contains an era e, index k, a confirmation certificate cc associated with e and k, and a full signature σ on cc.

We use standard notation for values in votes and certificates, e.g., cc.idx denotes the index of confirmation certificate cc. We say a vote and/or certificate is *for* (e, k) if it contains era e and index k.

Certificates provide a form of what is sometimes called *external validity* for Π_{fast}. Typically, a protocol has external validity if it is possible for non-participants (such as clients) to verify the legitimacy of each block in the chain. In Abraxas, these certificates are used by the participants themselves to verify the legitimacy of a Π_{fast} block, even if their instance of Π_{fast} has become stuck. (In the interest of generality, we do not assume Π_{fast} is externally valid; however, when Π_{fast} is externally valid, the compiler can be simplified slightly as an optimization.)

We remark that using hash digests to reduce communication introduces a minor subtlety: an honest party who receives a confirmation certificate, recovery vote, or recovery certificate cannot derive the corresponding block from the associated digest. However, the digest is guaranteed to correspond to a real block, because honest parties will only send confirmation votes if they have seen the corresponding block B. Moreover, all honest parties will eventually receive B, because there must be some honest party who originally sent a confirmation vote for B.

To simplify the protocol description, we implicitly assume that parties ignore any votes/certificates with invalid or missing signatures (or signature shares); thus, whenever we say a party receives a vote/certificate, it is implied that the vote/certificate in question has a valid partial or full signature, as appropriate. An honest party who receives $n-t$ votes (by different parties) on the same data is assumed to automatically transform them into a certificate. Thus, we usually do not distinguish between sets of $n-t$ votes (with valid partial signatures) on the same data and certificates with a single full signature, and we often use the term "certificate" to refer to either object. (E.g., "receiving a certificate" means either receiving $n-t$ votes for the same data or receiving a certificate directly.)

When a party obtains a confirmation certificate or a recovery certificate, it adds that certificate to buf_{slow} as a special "symbolic" transaction.[3] This allows parties to share a consistent view of whether the protocol is running "quickly," as will be discussed further below. Throughout, when we say a party *sees* a certificate we mean it either received that certificate directly from another party or it observed that certificate as a symbolic transaction in a block of C_{slow}.

Resolving blocks. A central idea in Abraxas is to detect failure of the fast path so that parties can fall back to the slow path. Informally, we determine that the fast path has failed if it is not "keeping ahead" of the slow chain. In other words, the fast chain should be outputting new transactions and forming certificates quickly relative to the speed of the slow path. By inputting certificates for fast chain blocks to the slow chain, we create a global timestamp for the time those transactions were confirmed. If these timestamps start to run late (or don't appear at all), then there is a problem with the fast chain.

A slow chain block $C_{\text{slow}}[\ell]$ is considered *resolved* once every transaction in the block has had its confirmation certificates appear on the slow chain within the next λ blocks, i.e., by block $\ell+\lambda$, where λ should be tuned to the block rate of both chains. The intuition behind this rule is as follows. If the fast path is working as expected (and λ was set appropriately), then it should take at most λ slow chain blocks to 1) confirm all of these transactions on the fast path, i.e., to produce a confirmation certificate for them 2) have all of these certificates become part of one of these λ slow chain blocks. Thus, if a block on the slow chain remains unresolved, then parties agree that something has gone wrong on the fast path and collectively decide to fall back to the slow path.

The rules for resolving a block are described in detail below. In our code, we use Procedure ssTryResolve in Algorithm 2 to find the index of the most recent block that cannot be resolved. P considers a block $C_{\text{slow}}[\ell]$ *ready to be resolved* if $\text{len}(C_{\text{slow}}) \geq \ell+\lambda$ and if for every confirmation certificate cc for the current era that P sees in $C_{\text{slow}}[:\ell+\lambda]$, P has received a confirmation vote cv such that $H(\text{cv.block}) = \text{cv.hash}$. (Thus, P knows a block corresponding to every confirmation certificate in $C_{\text{slow}}[:\ell+\lambda]$.) If $C_{\text{slow}}[\ell]$ is ready to be resolved, then it is *resolved* if for every transaction $\text{tx} \in C_{\text{slow}}[\ell]$ the following are contained in $C_{\text{slow}}[:\ell+\lambda]$:

- A confirmation certificate cc (for the current era) corresponding to a block B that contains tx.
- For all $k \leq$ cc.idx, a confirmation certificate cc_k for the current era and index k.

[3]In the interest of generality we do not assume the slow path differentiates between symbolic transactions and real transactions; in practice, however, one might optimize the slow path to prioritize symbolic transactions in order to recover more quickly after falling back.

Algorithm 2: Steady state helper functions

```
8  define ssTryResolve(ℓ):
9      if C_slow[ℓ] is not ready to be resolved then
10         return (ℓ, false);
11     else
12         if C_slow[ℓ] can be resolved then
13             return ssTryResolve(ℓ + 1);
14         else
15             return (ℓ + 1, true);

16 define ssTryVote():
17     Let k* ≤ len(C^e_fast) be maximal s.t. for all k < k*, P_i has
         seen cc_k for (e, k);
18     for k = k^cvoted + 1, ..., k* do
19         multicast a confirmation vote for (e, k, C^e_fast[k]);
20     k^cvoted := k*;

21 define ssTryOutput():
22     Let k* be maximal s.t. for all k ≤ k*, P_i has seen
         cv_k, cc_k for (e, k);
23     if k* − 1 ≥ k^out + 1 then
24         for k = k^out + 1, ..., k* − 1 do
25             find cv_k, cc_k for (e, k) s.t.
                 H(cv_k.block) = cc_k.hash;
26             output C_main[k^offset + k] := cv_k.block;
27         k^out := k* − 1;
```

In this case, ssTryResolve(ℓ) returns $(\ell + 1, \text{true})$. Note that if further blocks beyond block ℓ can be resolved, ssTryResolve(ℓ) recursively calls itself to find the most recent one that can be resolved. Otherwise, we say $C_{\text{slow}}[\ell]$ *cannot be resolved.* In this case, ssTryResolve(ℓ) returns (ℓ, false). Each party also maintains a global variable ℓ^* indicating the lowest-indexed block on C_{slow} that is not (yet) resolved.

The fast path. We now describe the steady state of our protocol. (Our simplified description omits some technical details; refer to Algorithms 3 and 2 for full details.) Upon entering the steady state when a new era e begins, a party invokes a fresh instance of C_{fast}. It executes the code specified in Algorithm 3, which invokes the helper functions in Algorithm 2 at appropriate events. We begin by explaining Procedure ssTryOutput (Algorithm 2). Roughly speaking, when a party outputs the kth block to C_{fast}, it will attempt to write it to C_{main} once it has written all previous fast chain blocks it has output this era to the main chain (i.e., up and including block $k-1$).

To prevent parties from committing different blocks at the same position, a party outputs a new block only upon seeing a confirmation certificate for this block. Thus, parties attempt to gather such certificates by multicasting [4] a confirmation vote (with index k) for each new block k that is output on C_{fast}. To do so, parties use a second Procedure, ssTryVote (Algorithm 2). As in ssTryOutput, a party only casts a confirmation vote for a new block once it has confirmed all previous blocks and if it hasn't sent a vote for index k before.

Whenever a party sees a confirmation certificate cc, it multicasts cc and adds cc to buf_{slow}. It then attempts to confirm and output as many blocks as possible to C_{main} (as "regular blocks") for which it has seen a contiguous sequence of confirmation certificates. (This is achieved by again calling ssTryVote and ssTryOutput). Note that a party P_i may output a block B to C_{main} regardless of whether B was output on P_i's fast path or was received as part of a confirmation certificate.

In summary, what the above achieves is the following: whenever a party outputs the kth block to C_{main}, then it can be sure that all other parties eventually receive the information necessary to confirm all blocks up to and including block k. In addition, we can guarantee a set of $t+1$ honest parties *have already confirmed* block k and thus have also confirmed and output all previous blocks up to and including block $k-1$. These properties of our compiler are crucially exploited in case the parties need to fall back.

To implement these steps, a party keeps track of the following global variables which are set at the beginning of each new era (i.e., upon reentering the steady state).

- k^{offset} is set to the length of C_{main} at the start of each era, and is used to determine the correct position for new fast-chain blocks on the main chain. (This is necessary because each era spawns a new instance of C_{fast}).
- k^{out} is set to 0 at the start of each era and is incremented upon outputting a block to the main chain during the steady state.
- k^{cvoted} keeps track of the indices for which a confirmation vote has been sent, to prevent sending votes multiple times for the same block. k^{rvoted} plays the same role for recovery votes.

Falling back. When a party outputs a block on C_{slow}, or receives a confirmation vote, it checks whether additional blocks on C_{slow} can be resolved; it transitions to the slow path if it sees that a block cannot be resolved.

As explained above, the intuition for this rule is that in the optimistic case, blocks are regularly output on C_{fast}. Hence, as parties are able to quickly form confirmation certificates on those blocks, this allows those blocks to be quickly output to C_{main}. On the other hand, if confirmation certificates are appearing more slowly (or not at all), the parties will switch to the recovery state.

When entering the recovery state, a party P stops C_{fast} and multicasts a recovery vote for the highest index k^{cvote} it has confirmed (Again, this simplified description omits some technical details in the interest of simplicity; refer to Algorithms 4 and Procedure rsInitialize in Figure 5 for full details.) Whenever P receives a confirmation certificate cc, it multicasts a recovery vote for the highest index k^* such that it has already received confirmation certificates for all previous indices and has cast recovery votes for them. (See Procedure rsTryVote.) Whenever P receives a recovery certificate, P multicasts it and adds it to buf_{slow}. Finally, P waits to see the first recovery certificate rc for the current era appear on C_{slow}. Once that happens, P waits until it has *caught up* to rc, namely, until

[4] Multicasting means sending a message to all other parties; we use this term to distinguish from (Byzantine/reliable) broadcast.

Algorithm 3: Steady state

```
28 define steadyState():
29   on entering this state:
30     switch := false;
31     k^offset := len(C_main), k^cvoted := 0, k^out := 0;
32     begin running Π^e_fast (which writes to C^e_fast);
33     ssTryVote();
34     ssTryOutput();
35     (ℓ*, switch) := ssTryResolve(ℓ*);
36   on outputting a new block to C^e_fast:
37     ssTryVote();
38   on outputting a new block to C_slow or receiving a
       confirmation vote cv with cv.era = e:
39     (ℓ*, switch) := ssTryResolve(ℓ*);
40   on receiving a confirmation certificate cc with cc.era = e:
41     if cc is the first confirmation certificate received for
         (e, cc.idx) then
42       multicast cc and add cc to buf_slow;
43       ssTryVote();
44       ssTryOutput();
45       (ℓ*, switch) := ssTryResolve(ℓ*);
46   on setting switch to true:
47     recoveryState();
```

Algorithm 4: Recovery state

```
48 define recoveryState():
49   on entering this state:
50     switch := false;
51     rsInitialize();
52     rsTryVote();
53     (ℓ*, switch) := rsTryRecover(ℓ*);
54   on outputting a new block to C_slow or receiving a
       confirmation vote cv with cv.era = e:
55     (ℓ*, switch) := rsTryRecover(ℓ*);
56   on seeing a confirmation certificate cc s.t. cc.era = e:
57     if cc is the first conf. certificate seen for
         (e, k) = (e, cc.idx) then
58       multicast cc;
59       rsTryVote();
60       (ℓ*, switch) := rsTryRecover(ℓ*);
61   on seeing a recovery certificate rc s.t. rc.era = e:
62     if rc is the first recovery certificate seen for
         (e, k) = (e, rc.idx) then
63       multicast rc and add rc to buf_slow;
64   on setting switch to true:
65     e := e + 1;
66     steadyState();
```

P receives, for all $k \leq$ rc.idx, a confirmation certificate (and corresponding confirmation vote) for the current era and index k. P attempts to catch up by calling Procedure rsTryRecover whenever it receives new certificates of either type.

Once P has caught up, it outputs on C_{main} the corresponding block—which we also call a "regular block"—for each of those confirmation certificates. To maintain liveness, P also outputs an additional[5] "cleanup block" containing all the other transactions in C_{slow} that have not yet appeared on C_{main}; then P switches back to the steady state. Our careful design ensures that if a party outputs a block k in the steady state, all parties eventually catch up to k and agree on this index. As discussed above, this is because parties will eventually receive all the necessary confirmation certificates/votes and at least $t + 1$ honest parties have already confirmed block k previously. Thus, all parties will eventually send a recovery vote for index k and it is impossible to form a recovery certificate for any index lower than k.

4 SECURITY

Abraxas achieves the same security as Π_{slow}. Formally:

THEOREM 4.1. *Let $t < n/3$. If Π_{slow} is a t-secure SMR protocol in an asynchronous network, then so is Abraxas.*

In this section, we give an overview of the main ideas of the proof of Theorem 4.1. (Formal proofs can be found in Appendix A.) The proof proceeds in three parts, for consistency (Theorem A.5), completeness (Theorem A.12), and liveness (Theorem A.13).

[5]As described, the protocol allows blocks to contain arbitrarily many transactions; one can modify the protocol to use fixed-size blocks by deterministically mapping a large block to a sequence of smaller blocks.

For the proof of consistency, the main challenge is in the analysis of the recovery state. Within the steady state of some era, consistency follows straightforwardly from the fact that confirmation certificates for a given era and index are unique (even if the underlying fast chain protocol is not consistent!). However, parties may enter the recovery state having output different numbers of blocks. Thus, the crux of the proof is showing that all honest parties finish the recovery state having "caught up" to the party who was farthest ahead.

The proofs of completeness and liveness rely on two main ideas, which are developed over a series of lemmas. Informally, we first need to prove that given enough time, some honest party will eventually make progress (e.g., enter a new state, resolve an index ℓ, etc.) towards outputting new blocks. Then, we show that if *some* honest party makes progress, then *all* honest parties will eventually make progress.

Notably, Abraxas does not rely on any properties of Π_{fast} for security. In principle, Abraxas would still be secure even if Π_{fast} was instantiated with a bad protocol that always outputs inconsistent values, or never outputs anything. In other words, a bad choice of Π_{fast} can't break Abraxas's security. Of course, the choice of Π_{fast} does affect throughput and latency—this point is explored in detail through our experimental evaluation (see Section 6).

5 EFFICIENCY

Let L denote a (fixed) block size, and let $CC_f(L)$ and $CC_s(L)$ denote the per-block communication complexity of Π_{fast} and Π_{slow} for block size L, respectively. Similarly, let RC_f and RC_s denote the per-block round complexity of Π_{fast} and Π_{slow}, respectively. As before,

Algorithm 5: Recovery state helper functions

67 **define** rsInitialize():
68   terminate Π^{e}_{fast};
69   Let k^* be maximal s.t. for all $k \leq k^*$, P_i has seen cc with $(\text{cc.era} = e) \wedge (\text{cc.idx} = k^*)$;
70   find cc s.t. $(\text{cc.era} = e) \wedge (\text{cc.idx} = k^*)$; then multicast a recovery vote for $(e, k^*, \text{cc.hash})$;
71   $k^{\text{rvoted}} := k^*$;

72 **define** rsTryVote():
73   let k^* be maximal s.t. for all $k \leq k^*$, P_i has seen cc with $(\text{cc.era} = e) \wedge (\text{cc.idx} = k)$;
74   **for** $k = k^{\text{rvoted}}, \ldots, k^*$ **do**
75   find cc among seen certificates s.t. $(\text{cc.era} = e) \wedge (\text{cc.idx} = k)$;
76   multicast a recovery vote for $(e, k, \text{cc.hash})$;
77   $k^{\text{rvoted}} := k^*$;

78 **define** rsTryRecover(ℓ):
79   **if** $\text{len}(C_{\text{slow}}) < \ell$ **then**
80   return (ℓ, false);
81   **else**
82   **if** $\nexists\, \text{rc} \in C_{\text{slow}}[\ell]$ *s.t.* $\text{rc.era} = e$ **then**
83   return rsTryRecover($\ell + 1$);
84   **else**
85   let rc be the first recovery certificate for era e in $C_{\text{slow}}[\ell]$;
86   **if** P_i *has caught up to* rc **then**
87   rsOutput(rc.idx, ℓ);
88   return $(\ell + 1, \text{true})$;
89   **else**
90   return (ℓ, false)

91 **define** rsOutput(idx, ℓ):
92   **for** $k = k^{\text{out}} + 1, \ldots, \text{idx}$ **do**
93   find valid cv, cc for (e, k) s.t. $H(\text{cv.block}) = \text{cc.hash}$;
94   set $C_{\text{main}}[k^{\text{offset}} + k] := \text{cv.block}$;
95   $B^* := C_{\text{slow}}[:\ell] \setminus C_{\text{main}}[:k^{\text{offset}} + \text{idx}]$;
96   output $C_{\text{main}}[k^{\text{offset}} + \text{idx} + 1] := B^*$;

κ denotes a computational security parameter; we assume partial signatures, full signatures, and hashes to all have length $O(\kappa)$.

Recall that a confirmation vote consists of an era e, index k, threshold signature σ_i, and block B. Thus, the length of a confirmation vote is $O(\kappa + L)$. Confirmation certificates are length $O(\kappa)$, because they carry a full signature and a hash of a block. Recovery votes and recovery certificates are both length $O(\kappa)$.

Optimistic case. The case in which the protocol remains in the steady state without falling back is called the optimistic case. In the optimistic case, each party runs Π_{fast} and Π_{slow} in parallel, and multicasts one confirmation vote and one confirmation certificate for each new fast chain block. One block is added to the main chain for each block output by the fast chain.

The communication cost per block output by the main chain in the optimistic case can be expressed as a function of several variables, including the amortized communication complexity of the fast and slow chains and the overhead incurred by the compiler. Putting these pieces together, the communication cost of a single main chain block in the steady state is $O(CC_f(L)+CC_s(L)\cdot\rho+\kappa+L)$, where ρ represents the number of slow chain blocks output per fast chain block. In principle a network adversary can influence ρ—even in the steady state—by selectively delaying fast chain messages; however, in benign conditions, ρ is the ratio of the two chains' round complexities, i.e., $\rho = (RC_s/RC_f)$. For example, using Jolteon as the fast path and 2-chain VABA as the slow path, $\rho = 5/(21/2) = 10/21$.

The throughput (i.e., transactions per unit time) in the optimistic case is equal to the throughput of the fast chain.

Worst case. It is challenging to precisely characterize the worst-case performance in a fully black-box way; below, we examine performance at a high level.

In worst case conditions, the fast chain stalls and fails to keep up with the slow chain, and parties will fall back to the recovery state after λ slow chain blocks. (There is a period in which the protocol is in "limbo," i.e., network conditions are no longer good but parties have not yet fallen back. During this period, the communication complexity of a deterministic synchronous or partially synchronous protocol may be arbitrarily high. For example, if parties' clocks can run at arbitrary rates, then a party running a timeout-based protocol may send an arbitrarily large number of messages in the time it takes for the slow chain to output λ blocks.)

While in the recovery state, each party will multicast confirmation certificates, recovery votes, and recovery certificates. Using Lemma A.9, the number of distinct indices for which honest parties send confirmation certificates (and hence the number of recovery votes and recovery certificates) is bounded by $(\max_{i\in H}\{k_i^{\text{out}}\}+1) - (\min_{i\in H}\{k^{\text{out}_i}\})$. (A majority of honest parties will not send this many votes or certificates. At least $t+1$ honest parties must enter the recovery state with $k_i^{\text{out}} \geq k^{\text{high}} - 1$, and therefore those parties (1) already sent certificates up to $k^{\text{high}}-1$ and (2) will not send recovery votes for values lower than $k^{\text{high}} - 1$.) The last cost associated with the recovery state is the per-block communication complexity of the slow chain multiplied by the number of slow chain blocks until the first recovery certificate is output. A recovery certificate will be received by all honest parties within 2 asynchronous rounds of falling back (one for all parties to receive confirmation certificates, one for all parties to vote). If the underlying slow chain has a mechanism for assigning certain transactions a higher priority, the recovery certificate can be output within an additional RC_s rounds. If it does not have such a mechanism, liveness of the slow chain still ensures that the recovery certificate will eventually be output, but the time until it is output may depend on the volume of transactions already in the system.

The throughput in the recovery state is slightly lower than the throughput of the slow path on its own, owing to the overhead incurred by storing certificates on the slow chain. In other words, if the slow path has a throughput of x Tx/s, the compiler's actual throughput (literal transactions per second) is $x - O(\kappa)$ Tx/s, since

$O(\kappa)$ bits are taken up by symbolic transactions. Reducing this overhead to $O(1)$ bits is an interesting open question.

6 IMPLEMENTATION AND EVALUATION

In this section, we evaluate a version of Abraxas in which Π_{fast} and Π_{slow} are instantiated using the state-of-the-art Jolteon and (2-chain) VABA protocols, respectively. We refer to the resulting protocol as Abraxas*. Our results show that the performance of Abraxas* matches or exceeds that of Ditto, a state-of-the-art hybrid protocol, in terms of both throughput and latency (except in the case of exceedingly poor network conditions). Compared to VABA, a leading protocol for the purely asynchronous setting, Abraxas* has much better latency when the network is fast. We conclude that Abraxas* offers high performance under good network conditions, while remaining more performant than the best existing hybrid protocol under bad network conditions.

6.1 Implementation

We implement Abraxas* in Rust,[6] building on the implementations of Jolteon and VABA by Gelashvili et al. [13]. The lookback parameter λ in Abraxas* is fixed to 20 blocks. (That parameter was set to heuristically optimize performance of Abraxas* in our experiments.) An implementation of Abraxas* strictly following the description in Section 3 would run *independent* executions of Jolteon each time the fast path is restarted. Instead, we optimize things by making sure to rotate the leader (in a round-robin fashion) each time a new instance of Jolteon is invoked. That way, if the fast path was abandoned because the leader was faulty, the restarted fast path will use a new leader. As another optimization, Abraxas* directly uses the signed and certified blocks already generated by Jolteon protocol as confirmation certificates rather than computing them independently (which would be redundant). Our Abraxas wrapper requires about 850 additional lines of code on top of the code for Jolteon and VABA. We use tokio[7] for asynchronous networking, ed25519-dalek[8] for elliptic-curve signatures, threshold_crypto[9] for coin tossing, and rocksdb[10] for persistent data structures.

In our experiments we run various SMR protocols among a fixed set of n nodes, each of which receives transactions from an external client at a predefined fixed rate. We have nodes commit to 512-bit hashes of the transactions, rather than the transactions themselves. All our experiments were conducted using Amazon EC2, with each node run on an independent m5.8xlarge instance. Nodes were spread uniformly across 8 regions: N. Virginia (us-east-1), Ohio (us-east-2), N. California (us-west-1), Oregon (us-west-2), Stockholm (eu-north-1), Frankfurt (eu-central-1), Tokyo (ap-northeast-1), and Sydney (ap-southeast-2). Each instance had 32 vCPUs supported by 16 processors, and all cores sustained a Turbo CPU clock speed of up to 2.5GHz. The instances all had 128GB memory and ran Ubuntu 20.04 LTS, and were connected to a network with 10 Gbps bandwidth.

[6]Code is available to reviewers upon request, and will eventually be released as open source.
[7]https://tokio.rs/
[8]https://github.com/dalek-cryptography/ed25519-dalek
[9]https://docs.rs/threshold_crypto/0.4.0/threshold_crypto/
[10]https://rocksdb.org/

6.2 Evaluation

We compare Abraxas* to two existing protocols: 2-chain VABA, the sub-protocol used to implement the slow path in Abraxas*, and Ditto [13], a hybrid protocol that also combines Jolteon and 2-chain VABA. Throughout, we fix the timeout parameter for Ditto at 10 seconds.[11] When the network is synchronous and the leader is non-faulty, Ditto uses 5 rounds to commit a block. In contrast, 2-chain VABA (which also serves as the slow path in both Ditto and Abraxas*) commits blocks every 10.5 rounds in expectation.

We perform experiments to evaluate Abraxas* relative to the other protocols in three different scenarios:

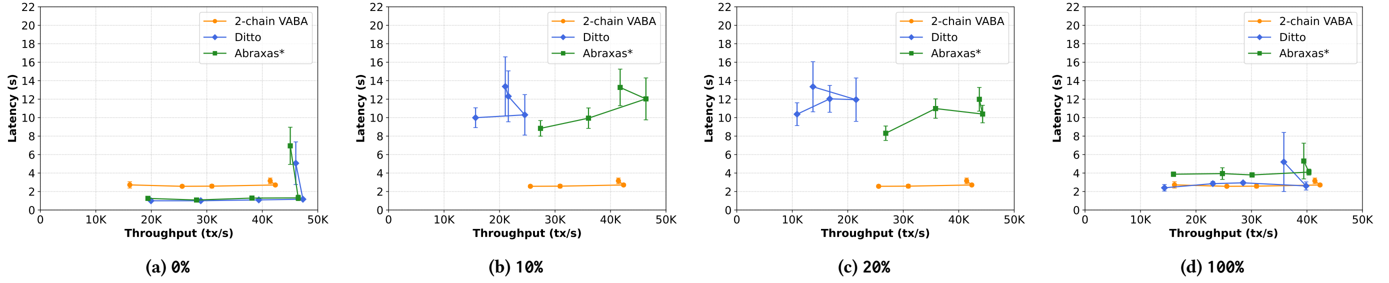
(1) **Non-faulty leader** (`0%`). Here the better choice is a partially synchronous protocol like Jolteon, and our aim is to understand how much worse (if at all) Abraxas* performs.
(2) **Always-crashed leader** (`100%`). This mimics both the effect of corrupted leaders as well as honest leaders in an asynchronous network where their messages are always delayed (e.g., if leaders are continually DDoS-ed by an adversary). In this case, an asynchronous protocol like 2-chain VABA would be the better choice, and we again want to understand how much worse Abraxas* performs.
(3) **Leader crashes randomly** (10–20%). This represents an intermediate point between the other two scenarios that should be best handled by an optimistic protocol. As crashes happen more often, however, any hybrid protocol (including Abraxas*) would repeatedly switch between the fast and slow paths and so this allows us to evaluate the overhead of such switching.

Our key performance metrics are the throughput and the end-to-end latency. Throughput is computed as the average number of transactions committed per second. Latency is the average time to commit a transaction, measured from the time a transaction is submitted by the client to all nodes until the time at which all nodes have committed the transaction. (The averages are taken over the duration of the experiment). Thus, the latency also includes the transaction queueing delay when the client submits a large number of transactions.

Figures 2–3 show our experimental results. Each experiment was run for approximately 3 minutes and was repeated 3 times; each data point in the graphs is the average of those trials with error bars representing one standard deviation.

Latency/throughput profiles. We first evaluate performance of the different protocols as a function of the system "load," i.e., the number of transactions injected by the client per unit time. That is, in each experiment we have the client send a fixed number of transactions per second, and we measure the obtained throughput and latency for a particular protocol. Regardless of the protocol being evaluated, we first observe an increase in throughput without any change in the latency as the load on the system increases. At some point, however, the system becomes "saturated" and as the load increases further the throughput remains the same while the latency increases due to an increase in the transaction queueing time. This saturation point differs for each protocol.

[11]Note that Ditto internally implements an exponential back-off that increases the timeout parameter as fallbacks occur.



(a) 0%   (b) 10%   (c) 20%   (d) 100%

Figure 2: Latency/throughput profiles, $n = 32$.

In Figure 2 we show the results of these experiments for $n = 32$ in different settings.[12] We observe:

- In the `0%` setting (Figure 2a) the maximum throughputs attained by Ditto and our protocol are similar since both are essentially running Jolteon on the fast path. 2-chain VABA, on the other hand, requires a larger number of rounds to commit in expectation and consequently has poorer throughput.
- In the `100%` setting (Figure 2d), both Ditto and our protocol eventually end up using 2-chain VABA on the slow path. Thus, after taking into account the time to switch to the slow path, the maximum throughputs of both protocols are similar but slightly worse than that of 2-chain VABA itself. Ditto has slightly better latency than Abraxas* here since Abraxas* continues to run the fast path even in this scenario. Ditto, on the other hand, uses an exponential back-off strategy that quickly confines it to the slow path entirely.[13]
- In the intermediate settings (Figures 2b–2c), the maximum throughput of our protocol is much better than that of Ditto and slightly better than that of VABA. Ditto spends a considerable amount of time switching between paths, during which no progress is made, whereas Abraxas* continually runs the slow path protocol in the background, thus not worsening throughput too much.
  On the other hand, the latency of our protocol is higher than that of VABA. This is expected since while Abraxas* does continue to create block certificates, those blocks are committed only when the protocol switches back to the steady state. In contrast, VABA commits blocks every fixed number of rounds (in expectation).

In the remaining experiments, each protocol's performance is measured at its saturation point, i.e., at the point where throughput is maximized but latency has not yet deteriorated.

Throughput over time. Figure 3 compares the number of transactions committed by the different protocols over time, still for $n = 32$. First note again that, in all cases, Abraxas* has throughput that is roughly on par with the throughout of VABA and is at least as high as—and sometimes much higher than—the throughput of Ditto. Moreover, if we focus on the slopes of the plots over time, we see that Abraxas* commits transactions at a fairly constant rate even under the least favorable conditions. This is in contrast to Ditto, which (in the `20%` and `100%` settings) has long periods during which no transactions are committed due to repeated switching between paths.

Throughput vs. number of parties. Finally, in Figure 4 we compare the throughputs of the different protocols for different numbers of parties $n \in \{16, 32, 64\}$. As expected, the throughput for each of the three protocols drops as the system size increases. We observe again that the throughput of Abraxas* is on par with, or better than, the throughput of the other two protocols; this is especially noticeable in the intermediate ranges.

[12] Instead of plotting latency and throughput vs. load, we plot latency vs. throughput. Note that, as just described, each plot begins with throughput increasing while latency remains roughly unchanged, and then at some point the latency increases while the throughput remains the same or even decreases slightly.

[13] It might be interesting to compare the performance of these protocols under varying timeout strategies (in Ditto) and lookback parameters λ (in Abraxas*). We leave such experiments for future work.

Acknowledgment

This work is supported in part by NSF Award 2237814.

REFERENCES

[1] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *IEEE Symposium on Security and Privacy*, pages 106–118. IEEE Computer Society Press, 2020.

[2] I. Abraham, D. Malkhi, and A. Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In P. Robinson and F. Ellen, editors, *38th ACM PODC*, pages 337–346. ACM, 2019.

[3] I. Abraham, K. Nayak, L. Ren, and Z. Xiang. Good-case latency of byzantine broadcast: A complete categorization. In *ACM PODC*, pages 331–341, 2021.

[4] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4), jan 2015.

[5] M. Ben-Or, B. Kelmer, and T. Rabin. Asynchronous secure computations with optimal resilience (extended abstract). In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '94, page 183–192, New York, NY, USA, 1994. Association for Computing Machinery.

[6] E. Blum, J. Katz, and J. Loss. Tardigrade: An atomic broadcast protocol for arbitrary network conditions. In *Theory of Cryptography Conference—TCC 2021*. Springer-Verlag, 2021.

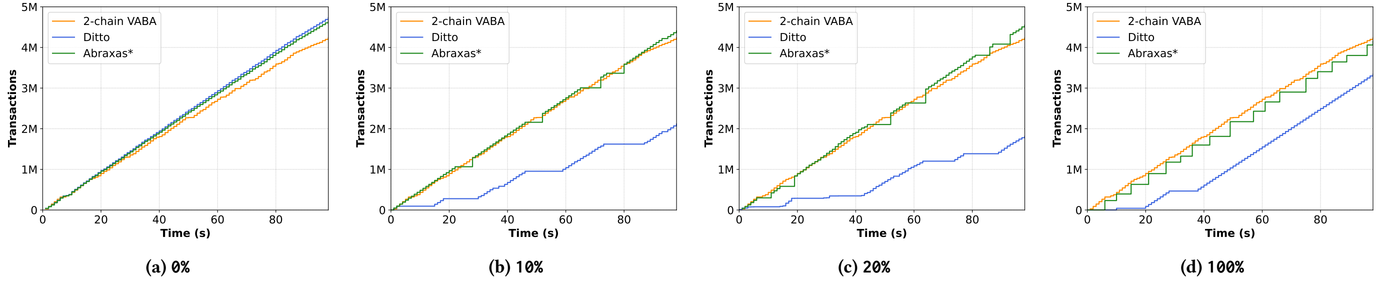
[7] E. Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

[8] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In J. Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 524–541, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[9] J. Camenisch, M. Drijvers, T. Hanke, Y.-A. Pignolet, V. Shoup, and D. Williams. Internet computer consensus. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 81–91, 2022.

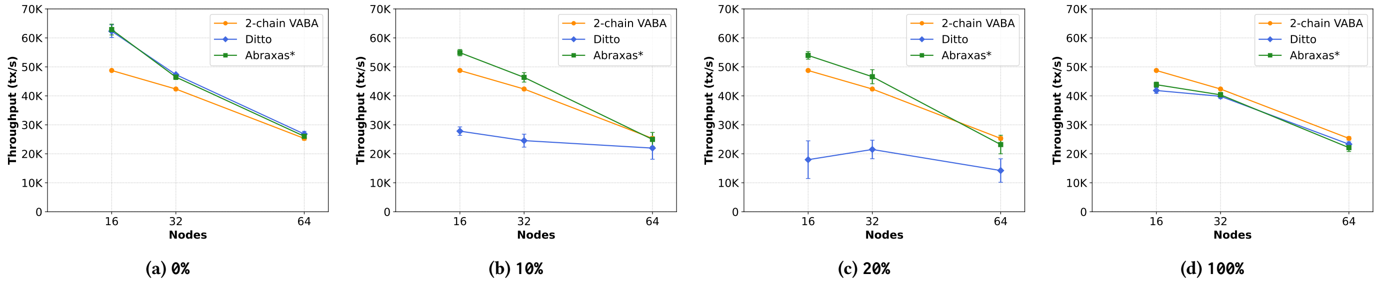
[10] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[11] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus. In *EuroSys*, pages 34–50, 2022.



(a) `0%` (b) `10%` (c) `20%` (d) `100%`

Figure 3: Throughput over time, $n = 32$.



(a) `0%` (b) `10%` (c) `20%` (d) `100%`

Figure 4: Throughput vs. system size.

[12] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, apr 1985.

[13] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang. Jolteon and Ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography*, 2022.

[14] N. Giridharan, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman. Bullshark: DAG BFT protocols made practical, 2022. Available at `https://arxiv.org/abs/2201.05677`.

[15] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, page 803–818, New York, NY, USA, 2020. Association for Computing Machinery.

[16] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman. All you need is DAG. In *Principles of Distributed Computing—PODC*, pages 165–175, 2021.

[17] K. Kursawe and V. Shoup. Optimistic asynchronous atomic broadcast. In L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors, *ICALP 2005*, volume 3580 of *LNCS*, pages 204–215, Lisbon, Portugal, July 11–15, 2005. Springer, Heidelberg, Germany.

[18] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[19] Y. Lu, Z. Lu, and Q. Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as pipelined BFT. *CoRR*, abs/2103.09425, 2021.

[20] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[21] R. Pass and E. Shi. Thunderella: Blockchains with optimistic instant confirmation. In J. B. Nielsen and V. Rijmen, editors, *Eurocrypt 2018, Part II*, volume 10821 of *LNCS*, pages 3–33, Tel Aviv, Israel, Apr. 29 – May 3, 2018. Springer, Heidelberg, Germany.

[22] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[23] H. V. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *OPODIS*, pages 88–102, 2005.

[24] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[25] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019*, pages 347–356, 2019.

A SECURITY PROOFS

When we say a party *sees* a certificate we mean it either received that certificate directly from another party, or observed that certificate in a block of $\mathcal{C}_{\text{slow}}$.

Lemma A.1. *If honest parties see confirmation certificates* cc, cc′ *with* (cc.era, cc.idx) = (cc′.era, cc′.idx) *then* cc.hash = cc′.hash.

Proof. Let (e, k) = (cc.era, cc.idx). Existence of cc implies that $n - t$ parties sent confirmation votes for era e, index k, and digest cc.hash; similarly, $n - t$ parties sent confirmation votes for era e, index k, and digest cc.hash′. Since honest parties send at most one confirmation vote per era and index, and $t < n/3$, we must have cc.hash = cc′.hash. (By collision-resistance of H, this also implies that the certificates are for the same underlying block.) □

Lemma A.2. *Suppose an honest party P_i enters the recovery state in some era, and let ℓ_i denote the value of ℓ^* held by P_i at the time it enters the recovery state. Then no other honest party ever holds $\ell^* > \ell_i$ in the steady state of that era.*

Moreover, if two honest parties call rsOutput *in era e, they do so using the same input (and so hold the same value of ℓ^* when entering the steady state of the next era).*

Proof. We prove the lemma assuming the honest parties hold the same value of ℓ^* at the beginning of the era. Since that is true at the beginning of the protocol, the lemma follows inductively.

Let ℓ_1 be the common value of ℓ^* held by the honest parties at the beginning of the era. An honest party who enters the recovery state does so while holding ℓ^* equal to the minimal value $\ell_2 > \ell_1$ such that $C_{\mathsf{slow}}[\ell_2 - 1]$ cannot be resolved. Consistency of Π_{slow} and collision-resistance of H imply that honest parties agree on this determination, and hence no honest party can ever hold $\ell^* > \ell_2$ while in the steady state.

An honest party who calls rsOutput from the recovery state does so while holding ℓ^* equal to the minimal value $\ell_3 \geq \ell_2$ for which $C_{\mathsf{slow}}[\ell_3]$ contains a recovery certificate for the current era. Thus, consistency of C_{slow} implies that any two honest parties who call rsOutput agree on the index ℓ_3 as well as the first such recovery certificate rc, and so call rsOutput on the same inputs. □

Lemma A.3. *If some honest party P ever outputs a regular block for index k and era e during the steady state and there is a recovery certificate for index k' and era e, then $k' \geq k$.*

Proof. We restrict attention to era e for the rest of the proof. Suppose that an honest party P outputs a regular block for index k. By the protocol description, P must have previously received a confirmation certificate for index $k + 1$. Thus, at least $t + 1$ honest parties sent confirmation votes for index $k + 1$.

Any honest party who sent a confirmation vote for index $k + 1$ must have received certificates for all indices up to and including k during the steady state. Thus, each of those parties would set k^{rvoted} to some value $k^* \geq k$ upon switching to the recovery state. Because honest parties send recovery votes for strictly increasing indices, this implies that there are at least $t + 1$ honest parties who will never send a recovery vote for any index $k' < k$, and so there cannot exist a recovery certificate for $k' < k$. □

Let $\mathcal{B}_i^e$ denote the sequence of blocks output on C_{main} by P_i in era e at some point in the protocol. (If P_i has not started running era e, or has started running era e but has not yet output any blocks, then $\mathcal{B}_i^e$ is the empty sequence.)

Lemma A.4. *Consider the chains $C_{\mathsf{main}}, C'_{\mathsf{main}}$ output by honest P_i, P_j (possibly with $i = j$) at arbitrary (possibly different) times during the protocol. Then for all eras e, the following hold: (1) one of $\mathcal{B}_i^e, \mathcal{B}_j^e$ is a prefix of the other (this includes the case $\mathcal{B}_i^e = \mathcal{B}_j^e$); (2) if both P_i and P_j had completed era e at the time they held their respective chains, then $\mathcal{B}_i^e = \mathcal{B}_j^e$.*

Proof. Fix an era e. If neither party had started running era e at the time they held $C_{\mathsf{main}}, C'_{\mathsf{main}}$, then the claim is trivially true; likewise, if P_i had started running era e but not P_j (or vice versa), then the claim is trivially true. Thus, consider the case where both P_i and P_j began running era e sometime before they held $C_{\mathsf{main}}, C'_{\mathsf{main}}$.

We prove the lemma assuming P_i, P_j hold the same value of k^{offset} at the beginning of era e; the lemma implies, in that case, that it continues to hold at the beginning of the next era. Since P_i, P_j hold the same value of k^{offset} at the beginning of the protocol, the lemma follows inductively.

The subchain $\mathcal{B}_i^e$ produced by an honest party P_i consists of 0 or more regular blocks followed by up to 1 cleanup block, with the cleanup block being added if/when they complete era e. Fix subchains $\mathcal{B}_i^e, \mathcal{B}_j^e$ and assume towards a contradiction that $\mathcal{B}_i^e[k] \neq \mathcal{B}_j^e[k]$ for some k. Assume (as the inductive hypothesis) that the lemma holds for all $e' < e$, and w.l.o.g. let k be the lowest index s.t. $\mathcal{B}_i^e[k] \neq \mathcal{B}_j^e[k]$. There are three possible cases:

1. Suppose both blocks are regular blocks. By the protocol description, if an honest party output a regular block $\mathcal{B}_i^e[k] = B$ (equivalently, $C_{\mathsf{main}}[k^{\mathsf{offset}} + k] = B$), then that block corresponds to a confirmation certificate it received for era e and index k. Because we assumed parties agree on the value of k^{offset}, this implies that P_i and P_j received confirmation certificates for the same (e, k) but different blocks. However, this violates Lemma A.1.
2. Suppose one block is a regular block and the other is a cleanup block. W.l.o.g., let $\mathcal{B}_j^e[k]$ be the cleanup block. Consider two subcases:
   (a) Suppose $\mathcal{B}_i^e[k]$ was output via ssTryOutput. This contradicts Lemma A.3, because the fact that $\mathcal{B}_j^e[k]$ is a cleanup block means that P_j must have received a recovery certificate for index $k - 1$ (cf. Algorithm 5, line 96), whereas Lemma A.3 states that no recovery certificates can exist for any $k' < k$.
   (b) Suppose the regular block $\mathcal{B}_i^e[k]$ was output via the for loop in rsOutput. Because the loop ranges over values from k^{out} to idx, P_i must have called rsOutput on some value of idx s.t. $k \leq \mathsf{idx}$. By the same token, P_j must have called rsOutput on a value of idx such that $k = \mathsf{idx} + 1$. However, by Lemma A.2, we know that both parties call rsOutput on the same arguments. This yields $k \leq \mathsf{idx} < \mathsf{idx} + 1 = k$, which is a contradiction.
3. Suppose both blocks are cleanup blocks. Recall from the protocol description that the contents of the cleanup block are computed deterministically from C_{slow}, C_{main}, and the arguments passed to rsOutput. Hence, if $\mathcal{B}_i^e[k] \neq \mathcal{B}_j^e[k]$, then either consistency of C_{slow} failed, Lemma A.2 was violated, or this is not actually the first index where P_i and P_j disagree; thus we reach a contradiction.

Together these show that for all $k \leq \min\{\mathsf{len}(\mathcal{B}_i^e), \mathsf{len}(\mathcal{B}_j^e)\}$ we have $\mathcal{B}_i^e[k] = \mathcal{B}_j^e[k]$, completing the proof of the first claim.

For the second claim, suppose both P_i and P_j have completed era e. If the subchains are the same length, the second claim follows immediately from the first; it remains to prove that the subchains must be the same length. Suppose towards a contradiction that this is not true. W.l.o.g., let P_i's subchain be the shorter one. Since both subchains have exactly one cleanup block, and that cleanup block is at the end, there must be some position k where P_j's subchain has a regular block and P_i's subchain has a cleanup block. As we argued in Case 3 above, this is not possible, so we reach a contradiction. This completes the proof of the second claim. □

Theorem A.5. *Abraxas satisfies consistency.*

Proof. Consider some honest parties P_i and P_j with respective chains C_{main} and C'_{main} at arbitrary (possibly different) times during the protocol, and let e, e' be the era associated with the highest block of $C_{\mathsf{main}}, C'_{\mathsf{main}}$ respectively. W.l.o.g., let $e \geq e'$. Because k^{offset} is set to the length of C_{main} at the start of each era, $C_{\mathsf{main}} = \mathcal{B}_i^1 || \ldots || \mathcal{B}_i^e$ and $C'_{\mathsf{main}} = \mathcal{B}_j^1 || \ldots || \mathcal{B}_j^{e'}$ (where $||$ denotes concatenation). Thus,

Lemma A.4 implies C'_{main} is a prefix of C_{main}, and so consistency is immediate. □

We now turn to proving liveness and completeness.

Lemma A.6. *For any era e and index k:*

- *If an honest party sees a confirmation certificate for (e, k), then for all $k' \le k$, all honest parties eventually see a confirmation certificate* cc *and a confirmation vote* cv *s.t.* $(\text{cc.era}, \text{cc.idx}) = (\text{cv.era}, \text{cv.idx}) = (e, k')$ *and* $H(\text{cv.block}) = \text{cc.hash}$.
- *If an honest party sees a recovery certificate for (e, k), then for all $k' \le k$ all honest parties eventually see confirmation certificates for (e, k').*

Proof. For part (1), suppose an honest party sees confirmation certificate cc_k for index k. It it directly received cc_k, then it would have multicast it, and so all honest parties eventually receive cc_k. Otherwise it observed cc_k in C_{slow} and so (by consistency and completeness of C_{slow}) all honest parties eventually observe it in C_{slow} as well. Furthermore, at least one honest party must have sent a confirmation vote for index k. That party must have received a confirmation certificate for all indices $k' < k$, and multicasts any confirmation certificate it receives; thus, every honest party will eventually receive a confirmation certificate for all indices $k' < k$ as well. For each $k' \le k$, the existence of a confirmation certificate $\text{cc}_{k'}$ for index k' implies that at least one honest party sent a confirmation vote for a block $B_{k'}$ with $H(B_{k'}) = \text{cc}_{k'}.\text{hash}$, and so all honest parties eventually receive such a vote.

For the proof of the second claim, suppose an honest party sees a recovery certificate for index k. Then at least $n - 2t > 0$ honest parties sent recovery votes for index k, and any such party must have received a confirmation certificate for index k, and so part (2) follows from part (1). □

Lemma A.7. *Let ℓ be an arbitrary index.*

1. *P_i eventually resolves ℓ (i.e., eventually P_i sets ℓ_i^* to a value higher than ℓ).*
2. *If honest parties P_i, P_j are in states φ_i, φ_j, respectively, immediately after resolving ℓ, then $\varphi_i = \varphi_j$.*

Proof. For the first claim, we will start by showing that if P_i sets $\ell_i^* = \ell$, then P_i eventually resolves ℓ. Suppose P_i sets $\ell_i^* = \ell$ at some time during the protocol. While ℓ is not resolved, P_i will continually call ssTryResolve or rsTryRecover (at a minimum, one of these is called whenever eventually a new block is output to C_{slow}, and liveness of C_{slow} ensures that this happens infinitely many times over the course of the protocol).

While P_i continues calling ssTryResolve, completeness of C_{slow} implies that eventually $\text{len}(C_{\text{slow}}) \ge \ell+\lambda$. Furthermore, by Lemma A.6, eventually P_i receives confirmation votes matching each confirmation certificate in $C_{\text{slow}}[\ell + \lambda]$. Once these conditions are satisfied, the innermost if/else block will be run and ℓ will resolve.

The case where P_i calls rsTryRecover proceeds similarly. By completeness of C_{slow}, eventually $\text{len}(C_{\text{slow}}) \ge \ell$. If $C_{\text{slow}}[\ell]$ is the first block to contain a recovery certificate rc for era e, then by Lemma A.6, P_i eventually receives confirmation certificates and matching confirmation votes for all $k \le \text{rc.idx}$, at which point ℓ resolves and P_i enters the steady state of the next era. Otherwise, $C_{\text{slow}}[:\ell]$ does not contain any recovery certificates for era e, and ℓ resolves immediately.

Since ℓ_i^* is initially set to 1 at the start of the protocol, the first claim follows by induction.

For the second claim, assume P_i and P_j are in the same state φ immediately before resolving $\ell^* = \ell$. Let $\varphi = (e, \text{ss})$. In order to resolve ℓ, both parties must have $\text{len}(C_{\text{slow}}) > \ell+\lambda$, and must have received confirmation certificates for all proofs in $C_{\text{slow}}[\ell+\lambda]$. By consistency of C_{slow} and uniqueness of confirmation certificates (Lemma A.1), P_i and P_j must take the same branch, and so their states after resolving ℓ will be the same. Similarly, suppose $\varphi = (e, \text{rec})$ for some e. In this case, both parties must have $\text{len}(C_{\text{slow}}) \ge \ell$, and therefore agree on whether $C_{\text{slow}}[\ell]$ contains the first recovery certificate. As before, this ensures they will (eventually) take the same branch, and so their states after resolving ℓ will be the same.

Since all honest parties begin the protocol in the same state, the second claim follows by induction. □

Lemma A.8. *If some honest party P_i switches to state (e, st) at some point during an execution, then every honest party switches to (e, st) at some point during that execution.*

Proof. Suppose P_i switches to state (e, st) upon resolving some index ℓ. The first part of Lemma A.7 implies that all honest parties eventually resolve ℓ, and the second part implies that after doing so they must also switch to (e, st). □

Lemma A.9. *During any era e, if no honest party receives a confirmation certificate for index k prior to calling* recoveryState(e)*, then no honest party sends a recovery vote for an index $k' \ge k + 1$.*

Proof. Fix an era e and an index k, and assume no honest party receives a confirmation certificate for index k prior to calling recoveryState(e). If no honest party ever calls recoveryState(e) then the claim is trivially true, so suppose that some honest party calls recoveryState(e). For each such party P_i, let k_i^{high} denote the index carried by P_i's initial recovery vote, and let $k^{\text{high}} = \max k_i^{\text{high}}$. Since we have assumed none of these parties has received a confirmation certificate for k upon calling recoveryState(e), we see that $k^{\text{high}} < k$.

Now, suppose towards a contradiction that some honest party sends a recovery vote for $k' \ge k + 1$. That party must have received a confirmation certificate for k'. This would imply that at least $t + 1$ honest parties sent confirmation votes for k' during the steady state. Each of those parties must have previously received a confirmation certificate for all $k'' \le k' - 1$ during the steady state, so their initial recovery vote would have been for a value $k_i^{\text{high}} \ge k' - 1$; this implies $k_i^{\text{high}} \ge k' - 1 \ge k > k^{\text{high}}$, a contradiction. □

Lemma A.10. *If all honest parties call* recoveryState(e) *(for the same e), then eventually some honest party calls* steadyState$(e + 1)$.

Proof. Suppose each honest party eventually enters state (e, rec). By Lemma A.7, there is some index ℓ such that every honest party switches to state (e, rec) upon resolving ℓ.

Let T_0 be the time when the last honest party enters state (e, rec). If any honest party has already entered state $(e + 1, \text{ss})$ by time T_0, the claim follows from Lemma A.8. Otherwise, all honest parties

are still in state (e, rec) at time T_0, and moreover each honest party has already sent at least one recovery vote. For each honest party P_i, let k_i^{high} be the index carried by P_i's first recovery vote, and let $k^{\text{high}} = \max_{i \in H} k_i^{\text{high}}$. At least one honest party must have received confirmation certificates for all $k \leq k^{\text{high}}$, and so by Lemma A.6 all honest parties eventually receive these certificates.

As before, if any honest party has left the recovery state prior to receiving all of these certificates we are done; so, suppose each honest party P_i is still in the recovery state when it receives these certificates. There are three possibilities: (1) P_i sends a recovery vote for k^{high}, (2) P_i previously sent a recovery vote for k^{high}, or (3) P_i previously sent a recovery vote for some $k > k^{\text{high}}$ (without ever sending a recovery vote for k^{high}). If either (1) or (2) holds for at least $2t+1$ honest parties we are done, because some honest party will eventually form a recovery certificate and forward it to all other parties, at which point liveness of C_{slow} and Lemma A.6 ensure that eventually some honest party sees that recovery certificate output on the slow chain and receives all the necessary confirmation votes and certificates to proceed to the steady state of era $e+1$.

Conversely, if there are fewer than $2t+1$ honest parties for which (1) or (2) holds, then there is some honest party for which (3) holds, i.e., who has never voted for k^{high} but has voted for some $k > k^{\text{high}}$. Call this party P_j. The definition of k^{high} and Lemma A.9 imply that k is at most $k^{\text{high}} + 1$, so combining these inequalities, k must be exactly $k^{\text{high}} + 1$. P_j must have received a confirmation certificate for $k^{\text{high}} + 1$, so by Lemma A.6, all honest parties eventually receive this certificate. If any honest party has already left the recovery phase upon receiving this certificate, we are done. Otherwise, for each honest P_i, one of the following is true when P_i receives the certificate: (1) P_i recovery votes for $k^{\text{high}} + 1$, or (2) P_j has already recovery voted for $k^{\text{high}} + 1$. (As noted above, it is not possible for P_j to have voted for a higher value.) By the same argument used in the previous case, liveness of C_{slow} and Lemma A.6 ensure that some honest party enters era $e+1$. □

Lemma A.11. *If there exists an era e such that no honest P_i enters era e during this execution, and transactions continue to be input to the system, then there exists an era $e^* < e$ such that $\forall k$, all honest parties eventually receive a confirmation certificate* cc *such that* $\text{cc.idx} = k$ *and* $\text{cc.era} = e^*$.

Proof. Suppose there is an era e such that no honest party P_i ever enters era e during this execution. Let $e^* < e$ be the highest era any honest party enters during this execution. By Lemma A.8, all honest parties eventually enter state (e^*, ss). Furthermore, Lemma A.10 implies that no honest party ever enters state (e^*, rec). This means that there is some global time after which all honest parties have entered state (e^*, ss) and will remain there forever.

Fix an index k^*. We will show that all honest parties eventually receive a confirmation certificate cc such that $\text{cc.era} = e^*$ and $\text{cc.idx} = k^*$.

Case 1: $k^* = 1$. Let T_0 be a point in the execution after each honest party enters era e^*. Choose a transaction tx that was created after time T_0 and input to all honest parties. By liveness of C_{slow}, this transaction is eventually output in some block $C_{\mathsf{slow}}[\ell]$. Furthermore, by completeness of C_{slow}, eventually each honest party outputs all blocks up to and including $C_{\mathsf{slow}}[\ell+\lambda]$. Since we have assumed that all honest parties remain in this era forever, $C_{\mathsf{slow}}[\ell]$ must be resolved, and in particular there must be a confirmation certificate $\text{cc} \in C_{\mathsf{slow}}$ that corresponds to a fast-chain block B containing tx. By Lemma A.6, all honest parties eventually receive this confirmation certificate.

Case 2: $k^* > 1$. Let T_0 be a point in the execution after some honest party has received a confirmation certificate for $k^* - 1$. Choose a transaction tx that was created after time T_0 and input to all honest parties. By liveness of C_{slow}, this transaction is eventually output in some block $C_{\mathsf{slow}}[\ell]$. Furthermore, by completeness of C_{slow}, eventually each honest party outputs all blocks up to and including $C_{\mathsf{slow}}[\ell+\lambda]$. Since we have assumed that all honest parties remain in this era forever, $C_{\mathsf{slow}}[\ell]$ must be resolved. This implies that there is a confirmation certificate in the chain prefix $C_{\mathsf{slow}}[:\ell+\lambda]$ that corresponds to a fast chain block B containing tx. Let $k' = \text{cc.idx}$. If $k' < k^*$, then clearly B cannot contain tx, since tx was created at a point when certificates for each $k' \leq k^* - 1$ already existed, and by Lemma A.1, there is at most one confirmation certificate per index. Therefore, $k' \geq k^*$. Using Lemma A.6, we conclude that all honest parties eventually receive a confirmation certificate for k^*. □

Theorem A.12. *The protocol is complete, i.e., for all k, any honest P_i eventually outputs $C_{\mathsf{main}}[k]$.*

Proof. If P_i enters era $k+1$, it must have exited the recovery phase for all eras $k' < k+1$. This implies that P_i has output at least k cleanup blocks to C_{main}, and therefore has output a block at index k.

If P_i never enters era $k+1$, let e^* be the highest era that some honest party enters during execution of the protocol. By Lemma A.8, eventually all honest parties enter era e^* (including P_i), and so $e^* < k+1$. Furthermore, by Lemma A.10, the honest parties remain there for an infinitely long period of time. Let k^{offset} be the length of C_{main} at the time P_i enters era e^*. If $k \leq k^{\text{offset}}$, we are done, so assume $k^{\text{offset}} > k$. By Lemma A.11, P_i eventually receives confirmation certificates for all $k' \leq k - k^{\text{offset}}$. At that point, if there are any blocks $C_{\mathsf{fast}}^{e^*}[k']$ for $k' \leq k - k^{\text{offset}}$ that P_i has not already output to C_{main}, P_i will do so now via ssTryOutput, and we are done. □

Theorem A.13. *The protocol is live, i.e., if* tx *is added to each honest party's buffer, then eventually* tx *will be output to the main chain of every honest party.*

Proof. Suppose a transaction tx has been added to each honest party's buffer by some global time T_0. It suffices to show that some honest party eventually removes tx from their buffer, because then consistency and completeness imply that all honest parties eventually do the same.

Consider a time $T_1 \geq T_0$ such that no honest party has yet removed tx from their buffer. (If no such T_1 exists, some honest party has already removed tx from their buffer, and we are done.) By liveness of C_{slow}, some honest party P_i will eventually output tx in some block $B = C_{\mathsf{slow}}[\ell]$. Furthermore, by completeness of C_{slow}, eventually P_i will have output all blocks in the prefix $C_{\mathsf{slow}}[:\ell+\lambda]$.

By Lemma A.7, P_i eventually resolves ℓ. We consider two cases. For the first subcase, suppose P_i resolves ℓ during the recovery state of some era e. By Lemma A.10 and A.8, P_i eventually switches to the steady state of era $e + 1$. Before it leaves the recovery state, it appends one or more cleanup blocks to the main chain. At this point, if tx was not already output to C_{main}, it will be included in a cleanup block.

For the second case, suppose P_i resolves ℓ during the steady state of some era e. There are two subcases. If P_i remains in the steady state upon resolving ℓ, then $C_{\mathsf{slow}}[\ell]$ must have been resolved. In particular, P_i must have received confirmation certificates for a contiguous prefix of C^e_{fast}, up to and including some block containing tx. This means that when P_i called ssTryOutput just prior to calling ssTryResolve, either P_i had already output tx to the main chain, or it does so then. For the second subcase, suppose P_i switches to the recovery state as a result of resolving $C_{\mathsf{slow}}[\ell]$. As in the first case, P_i will eventually recover and so every transaction in $C_{\mathsf{slow}}[\ell]$ will be output in a cleanup block (if it was not output already). ◻