Charlotte: Reformulating Blockchains into a Web of Composable Attested Data Structures for Cross-Domain Applications

ISAAC SHEFF, Heliax, USA
XINWEN WANG, Cornell University, USA
KUSHAL BABEL, Cornell Tech, USA
HAOBIN NI, Cornell University, USA
ROBBERT VAN RENESSE, Cornell University, USA
ANDREW C. MYERS, Cornell University, USA

Cross-domain applications are rapidly adopting blockchain techniques for immutability, availability, integrity, and interoperability. However, for most applications, global consensus is unnecessary and may not even provide sufficient guarantees.

We propose a new distributed data structure: *Attested Data Structures* (ADS), which generalize not only blockchains, but also many other structures used by distributed applications. As in blockchains, data in ADSs is immutable and self-authenticating. ADSs go further by supporting application-defined proofs (*attestations*). Attestations enable applications to plug in their own mechanisms to ensure availability and integrity.

We present *Charlotte*, a framework for composable ADSs. Charlotte deconstructs conventional blockchains into more primitive mechanisms. Charlotte can be used to construct blockchains, but does not impose the usual global-ordering overhead. Charlotte offers a flexible foundation for interacting applications that define their own policies for availability and integrity. Unlike traditional distributed systems, Charlotte supports heterogeneous trust: different observers have their own beliefs about who might fail, and how. Nevertheless, each observer has a consistent, available view of data.

Charlotte's data structures are interoperable and *composable*: applications and data structures can operate fully independently, or can share data when desired. Charlotte defines a language-independent format for data blocks and a network API for servers.

To demonstrate Charlotte's flexibility, we implement several integrity mechanisms, including consensus and proof of work. We explore the power of disentangling availability and integrity mechanisms in prototype applications. The results suggest that Charlotte can be used to build flexible, fast, composable applications with strong guarantees.

CCS Concepts: • **Computer systems organization → Peer-to-peer architectures**; **Distributed architectures**; *Availability*; *Redundancy*; • **Information systems → Data structures**; **Distributed storage**; *Distributed database transactions*; *Electronic commerce*; Remote replication; Web data description languages; • **Software and its engineering → Organizing principles for web applications**; • **Security and privacy → Distributed systems security**; *Trust frameworks.*

Authors' addresses: Isaac Sheff, Heliax, 905 Elmwood Ave., Buffalo, NY, 14222, USA, isaac@heliax.dev; Xinwen Wang, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, xinwen@cs.cornell.edu; Kushal Babel, Cornell Tech, Bloomberg Center, New York, NY, 10044, USA, babel@cs.cornell.edu; Haobin Ni, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, haobin@cs.cornell.edu; Robbert van Renesse, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, rvr@cs.cornell.edu; Andrew C. Myers, Cornell University, Gates Hall, Ithaca, NY, 14853, USA, andru@cs.cornell.edu.

Permission to make digital or hard copies of all or part 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 components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2023 Association for Computing Machinery.
XXXX-XXXX/2023/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Isaac Sheff, Xinwen Wang, Kushal Babel, Haobin Ni, Robbert van Renesse, and Andrew C. Myers. 2023. Charlotte: Reformulating Blockchains into a Web of Composable Attested Data Structures for Cross-Domain Applications . 1, 1 (September 2023), 51 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Blockchains are not only a way to manage cryptocurrencies [26, 58, 69] but also a more general mechanism for storing data with strong guarantees of integrity, availability, and durability. These properties are already commercially valuable for applications such as supply-chain tracking [42], but they have many other potential applications, such as long-term archival of the scientific record [1, 47, 61, 76] and managing real estate transactions [43].

The appeal of blockchain guarantees has led to a proliferation of blockchain systems with a variety of tradeoffs between performance and security. Unfortunately, there does not seem to be any one-size-fits-all design. Traditional blockchain designs such as Bitcoin [69] are slow, expensive, inefficient [20], and even considered unsafe by some [48, 68]. Traditional blockchains use a massive, expensive, global consensus to totally order everyone's transactions. The global consensus is intended to be so trustworthy that all services can trust the chain for availability and integrity. For many services, such as Ethereum's cryptokitties [90], blockchains are overkill, while for others, they are not enough. For example, some major banks do not trust the Bitcoin blockchain to ensure payments aren't rolled back [48].

We introduce Charlotte, a principled system that offers a framework for construction and interaction of a variety of blockchain-like systems, as applications within Charlotte. Charlotte is not a blockchain; instead, it deconstructs blockchains into a lower-level system in which one can *build* blockchains as Charlotte applications, including efficient versions of proof-of-work blockchains like Nakamoto [69] and Prism [6], permissioned blockchains like HyperLedger [12], as well as more specialized decentralized data structures. Charlotte is not a library; like the Web, it is a *system* that can be extended incrementally in a decentralized way with new applications. Blockchain applications implemented on Charlotte can refer to each other's blocks, and their transactions can even be made part of multiple blockchains. Crucially, Charlotte offers clear semantic guarantees when different blockchain applications interact.

Charlotte enables developers to build systems beyond the conventional, linear, totally ordered ledger that the term "blockchain" suggests. We view a blockchain as an instance of a more general construction, a distributed data structure in which data blocks reference each other with cryptographic hashes. While a blockchain is such a structure, so too are distributed hash tables as in CFS [21], the core data structures of distributed version-control services like Git [91], file distribution services like BitTorrent [18], and Public Key Infrastructure services [40]. Similarly, the Charlotte data structure is connected by hashes, but in addition, the Charlotte data structure includes *attestations* of application-specific integrity and availability properties. We refer to such a data structure as an *Attested Data Structure* (ADS). Together, all Charlotte ADSs form the *blockweb*, an attested directed acyclic graph (DAG) [60] of all Charlotte data, divided into *blocks* that reference each other by hash.

Charlotte applications can achieve high levels of integrity without traditional drawbacks by using the flexibility of attestations to achieve *entanglement* of different block histories. Traditional blockchains work because entanglement of different transactions means that no application's actions can be undone without rolling back others. Our insight is that global consensus mechanisms are an unnecessary and expensive way to achieve high entanglement. Charlotte makes it possible to instead build applications that exploit *opportunistic entanglement*, a lighter-weight way to bolster each application's integrity. Charlotte applications can follow the *least ordering principle*: only that

which must be ordered, should be ordered. As the database community has known for decades [11], unnecessary ordering is expensive.

The appeal of blockchains is that they offer assurance of integrity and availability even in the presence of strong adversaries, who can cause failures such as crashing or corrupting host nodes, or adding contradictory blocks. Charlotte can offer similar assurances despite similarly strong adversaries. In Charlotte, any *observer* can specify their own threat model: the failures they believe the system should securely tolerate. Each user is an observer, but observers can include any system or entity that observes states of ADSs. Observers' failure specifications can encompass a wide variety of threat models, such as those used in proof-of-work systems or those of more traditional, trusted quorum systems, or even mixtures of the two.

Charlotte enables new kinds of services that share multiple ADSs across multiple trust domains. New services can even be created by composing multiple existing services. To illustrate the power of interacting services, we built several independently useful example services, including version control, authenticated storage, timestamping, and a permissioned blockchain. We then *composed* these services to produce *DarXiv*, a secure disaggregated version of arXiv [29], which can take advantage of the integrity and availability properties of its component services (section 8.5).

We believe Charlotte takes a first step toward a new ecosystem of interoperable services that fulfill the promises of blockchains, while avoiding their drawbacks. Charlotte itself occupies a low layer of the application stack: even lower than existing blockchains, since it can be used to build them. As a new abstraction layer, Charlotte specifies rules for formatting blocks, referencing blocks, and transmitting blocks across the network, and it also offers a formal model for reasoning about data integrity and availability. Each service can choose its own data structures and its own mechanisms for ensuring integrity and availability; Charlotte makes it easy for services to share these mechanisms while offering a common understanding of the resulting integrity and availability guarantees. A prototype implementation of Charlotte is publicly available at https://github.com/isheff/charlotte-public [80].

1.1 Roadmap and Contributions

- We expand on this motivation in section 2 and give an overview of Charlotte's design in section 3.
- Charlotte provides an extensible type system for blocks, and a standard API for communicating them (section 4).
- We present a general mathematical model for ADSs (section 5) with diverse properties expressed in terms of observers, a novel characterization of different failure tolerance for each participant. Our model features a general way to compose ADSs and their properties.
- To demonstrate its usefulness, expressiveness, and ability to compose services, we describe how various interesting services can be built using Charlotte (section 6).
- Further, we show that Charlotte generalizes blockchains while allowing separation of availability and integrity duties, and also generalizes linearizable transactions over distributed objects (section 7).
- We have implemented a prototype of Charlotte along with proof-of-concept implementations of ADSs for various services (section 8).
- Our evaluation shows Charlotte's performance overheads are reasonable (section 9).

2 MOTIVATION

Modern applications rely on a variety of services that can be viewed as ADSs. A single modern application might use timestamping services, version tracking, public key infrastructure, distributed storage, and blockchains or other ordering services. There is unfortunately no uniform framework

for addressing data from all such services or sharing data between them. As a result, applications often unnecessarily duplicate data in their own formats, rather than contributing to an interoperable web of data that anyone can use.

Most distributed applications rely on data structures maintaining a variety of guarantees. Availability guarantees describe which data is safely stored and can be retrieved. Integrity guarantees can specify invariants over the possible states of an ADS. Until now, there has not been a formal, composable model for expressing the integrity or availability guarantees of ADSs. With Charlotte, applications can express guarantees for their ADSs in terms of which possible states can be constructed. Users can express their assumptions about server behavior (including possible failures), thereby defining which ADS guarantees they expect to hold and which they do not.

Furthermore, when applications share data, it is possible for one application to boost the availability and integrity of another's ADS. For instance, suppose ALICE publishes a document on her server, and BOB wants to make the document available even if ALICE's server fails. With Charlotte, he can duplicate it onto his own server, and clients transparently understand that these are the *same document*, now made strictly more available. This is a unique feature of the blockweb. With more traditional web infrastructure, data identity is tied to data location (often specified as a URL). Conflating data identity with location unnecessarily limits its availability and integrity.

2.1 Contrasting with Blockchains

A key novelty of Charlotte is its generality; it is not service-specific. Unlike other systems that build Directed Acyclic Graphs (DAGs) of blocks, Charlotte does not implement a cryptocurrency [55, 72, 75, 84, 85], require a universal "smart contract" language for all services [36, 45, 92], have any distinguished "main chain" [71, 95], or enforce the same integrity requirements across all ADSs in the system [14, 23, 51, 59, 96].

Indeed, since popular blockchains limit the integrity of any application to that of the global consensus, they actually *limit* the integrity of all data structures stored on them. For example, an attacker with 51% of the Ethereum stake could revert any transaction on any Ethereum smart contract, even if the service using the smart contract has other integrity mechanisms it trusts. This is why JPMorgan runs their cryptocurrency in-house: they trust their own integrity service more than they trust Ethereum's [48].

Instead, Charlotte distills ADSs down to their essentials, allowing Charlotte to serve as a more general *ADS framework*, in which each service can construct an ADS based on its own trust assumptions and guarantees—yet all of these heterogeneous ADSs are part of the same blockweb. We show that Charlotte is flexible enough to simultaneously support a variety of services, including Git-like distributed version control, timestamping, a disaggregated arXiv, and blockchains based variously on agreement, consensus, and proof-of-work.

Charlotte services can create an ordering on blocks, but blocks are by default only partially ordered. Charlotte ADSs can *intersect*, or share blocks. Not only can services benefit from each other's storage, but importantly, they can atomically commit information to multiple ADSs simultaneously. By contrast, services on traditional blockchains, such as Ethereum's smart contracts, obtain composable data structures and atomic commits only by storing all data on one big chain that by requiring a global consensus is inherently more expensive, even with the recent move toward Proof of Stake.

2.1.1 Concurrency in Banking.

Although Charlotte is not limited to totally ordered blockchains, the more general DAG data structures it supports have advantages even for conventional blockchain applications. In particular, a partially ordered DAG structure can be updated in parallel, improving throughput. To see the potential benefit, consider the Bitcoin payment network. Bitcoin keeps track

		Unaltered	2 Accounts
linearized	longest chain	6,953,512	24,129,215
	time	**3.72 years**	12.91 years
parallelized	longest chain	110,787	244,163
	time	21.63 days	**47.68 days**

Table 1. Theoretical advantages of Charlotte-style parallelization in the Bitcoin payment network for the first 200,000 blocks.

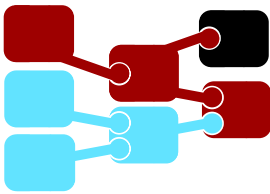

Fig. 1. Composing data structures. Blocks are represented as rectangles, and references from one block to another are shown as circles. The pale blue blocks form a tree, whereas the darker red blocks form a chain. The rightmost red block references a blue block, so together the union of the red and blue blocks forms a larger tree. The black block also references a red block.

of money in terms of Unspent Transaction Outputs, or UTXOs. Each Bitcoin transaction consumes a set of input UTXOs and generates new output UTXOs. Thus, Bitcoin transactions form a graph, with transactions as vertices and UTXOs as directed edges [69]. Note that transactions that do not depend on each other can happen in parallel. This is one of the reasons that research in side-chains is so popular: they increase parallelism. However, whereas side-chains tend to ask "what is the least parallelism we can get away with?" we prefer to ask "what is the least ordering we actually need?" With minimal ordering, even if each transaction required a consensus mechanism exactly as slow as the one Bitcoin uses, all the transactions of the first 200,000 blocks could be confirmed 63× faster in a Charlotte-based ADS without unnecessary ordering (table 1).

A Bitcoin "account," where a user owns one UTXO at a time, is simply a path in the Bitcoin transaction graph. In Bitcoin, it improves anonymity and performance to combine many small transfers of money into big ones, with many inputs and many outputs. In the real financial system of the USA, however, all monetary transfers are from one account to another. They are all exactly two-chain transactions. We describe how to construct such a banking system in Charlotte in section 7.4.1. For Bitcoin data, we can simulate this limitation by refactoring each transaction as a butterfly network of transactions with logarithmic depth (appendix A).

Even in a financial system limited to pairwise transactions (as we describe in section 7.4.1), Charlotte could still achieve a 28× speedup by exploiting the power of parallelism (table 1).

2.2 Composability

Charlotte provides a common framework for data structures from separate services to reference each other. For example, a timestamping service can stamp data from any other ADS, using a common reference format. A blockchain might "endorse" specific commits from a version-control repository as representing an "official release." This is exactly how our DarXiv example application (section 8.5) is structured: a permissioned blockchain represents editors designating official releases of a document.

Charlotte data structures are naturally *composable*: the *union* of two data structures is itself a data structure. For example, the red and blue blocks in fig. 1 are separate data structures whose union forms a data structure. More concretely, a Git-style version-control repository is a union of many "branch" data structures. Sharded database or blockchain designs form one big database as the union of many "shard" databases. Timestamps created by a timestamping service create a

"timestamped data" ADS consisting of all ancestors of the timestamp in the blockweb, potentially incorporating any number of other ADSs.

The *intersection* of data structures is also meaningful. Intuitively, the intersection of two data structures comprises the data that is part of both structures. We can think of cross-shard transactions appended to a sharded blockchain ADS as data in the intersection of multiple shard ADSs.

With Charlotte, we want to be able to naturally express both union and intersection ADSs, and to reason about their properties in terms of the properties of the component ADSs. For instance, in our DarXiv example application (section 8.5), documents gain availability by being in the union of many authenticated storage data structures, and integrity through being in the intersection of version control, timestamping, and permissioned blockchain data structures. The DarXiv as a whole is the union of many individual document data structures.

2.3 Availability and Integrity Properties

An ADS does not automatically possess all properties a service might need. An ADS might fail to ensure availability, because a reference to data does not guarantee it can be retrieved. It might even fail to ensure integrity, because an ADS might be extended in inconsistent, contradictory ways—for example, multiple new blocks could claim to be the 7th in some blockchain.

Therefore, an ADS commonly incorporates additional mechanisms to ensure availability and integrity in the presence of malicious adversaries. Some systems rely on gossip and incentive schemes to ensure availability, and consensus or proof-of-work schemes to ensure integrity.

Presently, only observers that believe in a set of service-specific failure assumptions can use each service. Historically, this has been a core reason ADS services haven't been composable. Blockchains like Bitcoin [69] and Ethereum [26] lose integrity if the adversary controls a majority of the hash power, while Chord loses availability if an adversary controls enough consecutive nodes [89].

Charlotte provides a theoretical framework for formalizing the availability and integrity properties of extremely diverse ADSs. Charlotte characterizes availability in terms of the conditions under which data might not be available, but since the system is decentralized, different observers might have different opinions about these conditions. Integrity is similar: each observer characterizes each ADS in terms of which states it believes are possible.

These properties can be composed just as the ADSs can: For example, the intersection of two ADSs (section 5.7.2) has, in a strong sense, the integrity of both. If two different timestamping services both claim a block existed before a certain time, that proof stands so long as at least one timestamping service is honest.

3 DESIGN OVERVIEW

3.1 Blocks

In Charlotte, blocks are the smallest unit of data. Traditional blockchain systems are motivated to increase block size to increase throughput. Because Charlotte does not mandate a global ordering, it has no such design tension, so Charlotte clients do not fetch "block headers," or other partial blocks, as in some other systems [26]. Therefore, Charlotte services ideally use small blocks. For instance, to build something like Ethereum in Charlotte, it would be sensible to represent the Merkle tree [64] structure found within each Ethereum block using many small Charlotte blocks. Using small blocks makes it easier to divide up storage duties and to fetch and reference specific data. Since blocks are small, Charlotte's server protocol allows for streaming arbitrary collections of blocks. Efficient applications may want to disseminate groups of related blocks. We discuss some useful ways of grouping blocks in section 4.4.

3.2 Observers

To capture the formal guarantees offered by Charlotte, we characterize an *observer* in a distributed system as an entity with a set of assumptions concerning the possible ways that the system can fail. For example, an observer can represent a user: they expect a data structure to retain certain properties, so long as certain assumptions about the relevant servers remain true. In Lamport's Paxos terminology, *learners* are examples of observers: they observe data, and demand agreement properties on their observations [54]. Charlotte explicitly considers active adversaries who are trying to corrupt ADSs; observers can characterize their assumptions regarding these adversaries in terms of crashed and/or Byzantine servers. We call an observer *accurate* if its failure assumptions are correct.

Given a set of assumptions about who can fail and how, and the desired integrity properties of each ADS, each observer may choose to ignore any portions of the blockweb that lack adequate attestations (section 3.3). What remains is the observer's *view* of the ADS: the set of blocks it believes are available and part of the state of the ADS.

Each observer's view of an ADS is guaranteed to remain available and to uphold any integrity properties the observer has chosen so long as the observer is accurate (i.e., its failure assumptions hold). Further, portions of the blockweb that feature attestations satisfying two observers are guaranteed to remain in both observers' views, once both have observed all the relevant blocks. Of course, in practice, servers take time to download relevant blocks, and in an asynchronous system there is no bound on the time this may take.

3.3 Attestations

Some blocks are *attestations*: they prove that an ADS satisfies properties beyond those inherent to a DAG of immutable blocks. References to blocks are accompanied by a set of references to attestations describing the properties of that block, specifying where the block can be found and what data structure it is part of.

For instance, if a server signs an attestation stating that it will store and make available a specific block, it means the block will be available as long as that server functions correctly. Such an attestation functions as a kind of proof premised on the trustworthiness of the signing server. In other words, if the property guaranteed by the attestation fails to hold, the signing server must not be trustworthy. Servers should sign attestations only when they can be sure that the property is enforced.

3.3.1 Specifying Attestations. Charlotte deliberately does not prescribe a language or syntax of attestations: that would limit what developers could do with Charlotte. Instead, applications provide their own interpretation of the meaning of their attestations: how an attestation is interpreted is up to observers, since how much an observer *believes* any specific attestation is up to that observer. We provide a flexible semantic framework in section 5.8 that applications can use to precisely specify the guarantees offered by attestations. It allows observers to reason about these guarantees either in isolation or when composed with other attestations, possibly made by other applications. When applications define the semantics of their attestations within that framework (as we do for all examples in this paper), application guarantees naturally compose: all properties of all attestations hold when all conditions are met (section 5.7).

Attestation types are *pluggable*: Charlotte servers can define their own subtypes, which prove nothing to observers who do not understand them. Although there is no fixed language for expressing attestations, each type is specified with a unique hash, so observers cannot accidentally

parse the same attestation in different ways. Service-defined attestation types can represent different consensus mechanisms (including Paxos and Nakamoto), different ADS types, and different availability strategies.

Applications that require guarantees which our semantic framework cannot express can still build on Charlotte's libraries and servers: they simply do not get the guarantees from our semantic model. Future work could introduce concrete syntaxes for attestations, and implement such attestations atop Charlotte. Such a syntax is at a higher level of abstraction than Charlotte: because the Charlotte system does not need to implement it, and it can be built above Charlotte, Charlotte does not implement it.

Although attestations can express a wide variety of properties about an ADS, we classify them as either *availability attestations* or *integrity attestations*.

3.3.2 Availability Attestations. Availability attestations prove that blocks will be available under certain conditions by specifying how to obtain those blocks. One example of an availability attestation would be a signed statement from a server promising that a given block will be available as long as the signing server is functioning correctly. Attestations may make more complex promises. For example, proofs of retrievability [13] might be used as availability attestations. Availability attestations are not limited to promises to store forever: they might specify any conditions, including time limits or other conditions under which the block is no longer needed. Availability attestations generalize and can model features found in many existing distributed data systems:

- In existing blockchains, clients wait for responses from many full nodes, to be sure their transaction is "available."
- In BitTorrent, a seeder tells a tracker that it can provide certain files to leechers.
- Many databases inform clients that their transaction has been recorded by a specified set of replicas.

In general, a reference to a block includes references to associated availability attestations that describe how to find the block. However, there are no availability attestations for availability attestation references, because that would be useless.

3.3.3 Integrity Attestations. An ADS often requires some kind of permission to add a block to its state. For example, a blockchain typically requires some set of servers ("miners") to decide that a particular block uniquely occupies a given height in the chain. Integrity attestations determine which blocks belong in which ADSs. For instance, servers maintaining a blockchain might issue an integrity attestation stating that a given block belongs on the chain at a specific height; the server promises not to issue any integrity attestation indicating that a different block belongs on the chain at that height. Timestamps are another integrity attestation type: they define an ADS consisting of all blocks a specific server claims existed before a specific time.

We call servers that issue availability attestations *Wilbur servers*[1] and servers that issue integrity attestations *Fern servers.*[2] Fern servers generalize ordering or consensus services. In blockchain terminology [69], they correspond to "miners," which select the blocks belonging on the chain. Figure 2 illustrates possible states of a Bitcoin-style blockchain Attested Data Structures. Under the assumption that the length of the branches is less than 3, correct Fern servers would not give integrity attestations to both block s and c simultaneously and thus prevent the system state from diverging. A single process can be both a Wilbur and a Fern server: it simply provides both types of interfaces for clients (section 4).

[1]after the *Charlotte's Web* character whose objective is to stay alive.
[2]after the *Charlotte's Web* character who decides which piglets belong.

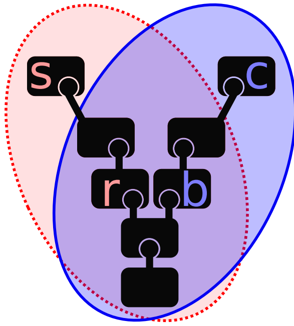

Fig. 2. A Bitcoin-style blockchain ADS with branches of length < 3. All of the blocks are present in the blockweb. The red dotted oval and the blue solid oval represent two possible *alternative* states of the ADS.

```
message AnyWithReference {
  google.protobuf.Any any;
  Reference typeBlock;
}
message Hash {
  oneof hashalgorithm_oneof {
    AnyWithReference any;
    bytes sha3; // technically unnecessary
  }
}
message Reference {
  Hash hash;
  repeated Hash availabilityAttestations;
  repeated Reference integrityAttestations;
}
message Block {
  oneof blocktype_oneof {
    AnyWithReference any;
    string protobuf;
  }
}
```

Fig. 3. Core Types of Charlotte: this (slightly simplified) proto3 code describes how blocks, references to blocks, and generic data are safely marshaled and unmarshaled in Charlotte.

3.3.4 Incentives. Each observer may want a different subset of servers to generate different kinds of attestations. Charlotte does not have a universal framework for incentivizing servers to issue attestations. We imagine that for many applications, observers will pay servers for these availability and integrity services, with attestations serving as formal receipts. Since Charlotte can be used to implement various different blockchains, suitable on-chain currencies can be used to compensate servers. Presumably, services that are more difficult to implement, such as a high degree of storage replication or a strong integrity guarantee, would command higher fees. This is not so very different from the way light client proofs are used in Ethereum: you get no money back if the chain maintainers are dishonest and the chain forks, but the maintainers are incentivized to stay honest so that they can continue offering their service for a fee.

4 CHARLOTTE API

Charlotte defines protocols through which clients and servers interact, specified as an open-source API [80]. The API is defined using gRPC [33], a popular network service specification language

```
message SendBlocksResponse {
  string errorMessage;
}
service CharlotteNode {
  rpc SendBlocks(stream Block) returns (stream SendBlocksResponse) {}
}
```

Fig. 4. All Charlotte servers implement the CharlotteNode service.

based on Protocol Buffers [73]. A language-independent description of the API is desirable, so we use simplified Protocol Buffers (protobuf) syntax to describe the Charlotte protocols.[3]

Figure 3 presents the core types used by Charlotte protocols, in protobuf syntax. Charlotte is built around four core types:

- **`Block`**: can contain any protobuf [73] data type, or the block itself can be a protobuf type definition. `Attestation` is a subtype of `Block`.
- **`Hash`**: represents the hash of a block. For convenience, we support SHA-3 by default, but arbitrary other hash subtypes can be added.
- **`Reference`**: is used by one block to reference another; it contains the `Hash` of the referenced block, along with zero or more references to attestations (section 3.3). These attestation references allow an observer to learn about the availability and integrity (ADS membership) of the referenced block, even before they fetch it.
- **`AnyWithReference`**: Anyone can add their own subtypes of `Block`, `Hash`, or `Attestation`, which any server can safely marshal and unmarshal. It contains a reference to the block where the type description can be found (as proto3 [73] source code), and marshaled data.[4]

We provide some useful example subtypes of `Hash` (e.g., `sha3`) and `Block` (e.g., `Attestation`).

In our API, all Charlotte servers must implement the `SendBlocks` RPC (fig. 4), which takes in a stream of blocks and can return a stream of responses that may contain error messages. We define subtypes of attestation for *Availability* and *Integrity*, and show how to construct and observer who trusts specific types, or even quorums of types (sections 5.8.1 and 5.8.2).

4.1 Wilbur

Charlotte servers which issue availability attestations are called Wilbur servers. Wilbur servers host blocks, providing *availability*. In blockchain terminology [69], Wilbur servers correspond to "full nodes," which store blocks on the chain. In more traditional data store terminology, Wilbur servers are key–value stores for immutable data. The Charlotte framework is intended to be used for building both kinds of systems.

In our API, Wilbur servers are Charlotte servers that include the `RequestAvailabilityAttestation` RPC (fig. 5), which accepts a description of the desired attestation, and returns either an error message, or a reference to a relevant availability attestation. For most applications, we imagine that an availability attestation does not itself require attestations, so this returned `Reference` will simply be a hash.

4.2 Fern

[3]For simplicity, our specifications omit the indices of the various fields. The actual source code is also slightly more complicated, for extensibility [80].

[4]The proto3 `Any` type itself features a URL string meant to reference the type definition, but Charlotte uses a block reference because it is self-verifying.

```
message AvailabilityPolicy {
  AnyWithReference any;
}
message RequestAttestationResponse {
  oneof requestattestationresponse_oneof {
    string errorMessage;
    Reference reference;
  }
}
service Wilbur {
  rpc RequestAvailabilityAttestation(AvailabilityPolicy)
    returns (RequestAttestationResponse) {}
}
```

Fig. 5. Wilbur Service Specification.

```
message IntegrityPolicy {
  AnyWithReference any;
}
service Fern {
  rpc RequestIntegrityAttestation(IntegrityPolicy)
    returns (RequestAttestationResponse) {}
}
```

Fig. 6. Fern Service Specification.

Charlotte servers which issue integrity attestations are called Fern servers. Integrity attestations define the set of blocks in a given ADS. Among other things, integrity attestations can be proofs of work, or records demonstrating some kind of consensus has been reached. One simple type of integrity attestation, found in our prototype, is a signed pledge not to attest to any other block as belonging in a specific slot in an ADS. Fern servers generalize ordering or consensus services. In blockchain terminology [69], Fern servers correspond to "miners," which select the blocks belonging on the chain.

In our API, Fern servers are Charlotte servers that include the `RequestIntegrityAttestation` RPC (fig. 6), which accepts a description of the desired attestation, and returns either an error message or a reference to a relevant integrity attestation. A Fern server might also request attestations for the new integrity attestation itself. For instance, it might store the integrity attestation on a Wilbur server (possibly itself, if the Fern server is also a Wilbur server), and get an availability attestation stating that the integrity attestation itself is available. When the Fern server returns a `Reference`, it can include attestation references, so that, for example, anyone seeking to look up the integrity attestation would know where it is stored.

4.3 Life of a Block

Figure 7 illustrates the API calls used during the life of a block. A client first starts a `CharlotteNode` service to communicate with the servers. The client then mints a block, including data and references to other blocks. To ensure the block remains available, the client sends it to Wilbur servers via the `RequestAvailabilityAttestation` API. The Wilbur servers store the block and return availability attestations, demonstrating the availability of the block.

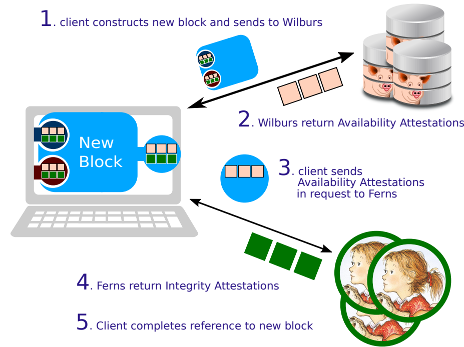

Fig. 7. Life of a block. A client mints a new block, and wants to add it to an ADS. The block, as drawn, includes two references to other blocks. The client acquires availability attestations from Wilbur servers, and integrity attestations from Fern servers. Then it can create a reference (drawn as a circle) to the block, so anyone observing the reference knows the block is in the ADS.

The client then uses the API call `RequestIntegrityAttestation` to submit a reference to the block to a collection of *Fern* servers, which maintain the integrity of a specific ADS. Since Fern servers may not want to permanently add a block to their ADS if that block is going to become unavailable, the client may also send earlier obtained availability attestations. If the block does not violate the integrity of the ADS, Fern servers return signed integrity attestations that prove the block is part of the ADS.

The client includes all of these attestations in references to the block, so that whenever an observer sees a reference to the block, they know how available it is, and what ADSs it belongs to. Over time, more attestations may be issued, so a block can become more available or increase in integrity, perhaps joining more ADSs.

Charlotte is flexible: blocks do not have to follow this precise life cycle. For example, a service might reduce the burden on a client for adding a block to a specific ADS by co-locating its Fern and Wilbur servers, and preemptively issuing integrity attestations and availability attestations as soon as it receives a relevant block from a client.

4.4 Practices for Additional Properties

In order to understand a `Reference` object within a block (how available the referenced block is, and data structures it is in), an observer reads attestations referenced within the reference object.[5] For example, without the content of the availability attestations, it is not clear where to look to retrieve the referenced block. As a rule of thumb, before one server sends a block to another, it should ensure the recipient has any attestations or type blocks (section 4) referenced within that block. This ensures the recipient can, in a sense, fully understand the blocks they receive. In fig. 8, for instance, when sending block *a*, the sender should be sure the recipient has received everything in the dashed rectangle. Our example applications follow this practice. It is possible, however, that for some applications, servers may be certain the recipient doesn't care about some attestations or type blocks, and therefore might leave those out.

When servers follow this practice, it is useful for availability attestations to attest to groups of blocks likely to be requested together. In fig. 8, for instance, an availability attestation that attests

[5]We considered making references contain full copies of attestations, but this made blocks large, and since many blocks may reference the same block (and attestations), blocks were full of redundant information.

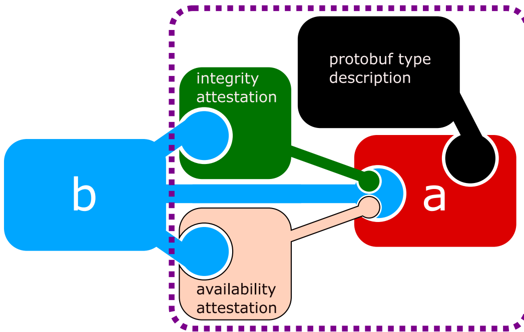

Fig. 8. Generic illustration of a block a, and the types of references it can include. Block a references block b, and that reference carries attestations. Block a also references a type description block, for unmarshaling data in a. In general, when sending block a to a server or client, the sender should be sure the recipient has received *all* the blocks in the dashed purple rectangle, so the recipient can fully understand block a and the properties of its references.

to everything in the dashed rectangle would be more useful than just attesting to block a. Our example applications' availability attestations are generally designed this way.

Availability failures can cause available states of ADSs to become disconnected subgraphs (if the blocks that connect them are forgotten). For most data structures, this isn't a particularly interesting concern. However, there *are* simple rules for building an ADS that will always remain connected:

- Availability attestations that attest to a block should also attest to the availability attestations referenced within that block.
- Whenever a block x references a block y, and block y references block z, if y isn't at least as available as z, then x should reference z as well.

Here, "y is at least as available as z" means that the availability attestations in references to y guarantee that y will be available in every circumstance under which the availability attestations in references to z guarantee that z will be available. In general, references need not explain where to find availability attestations, so a best practice is to store and transmit any attestations referenced (especially availability attestations) along with each block that references them (see fig. 8).

5 FORMALISM

We now turn to the theoretical model that defines guarantees for ADSs built using the Charlotte framework, and the assumptions of its observers. We first formalize the key concepts — ADSs, observers, observers' beliefs and the universe induced by these beliefs. We then show how observers refine their beliefs and calculate incontrovertible states of an ADS in light of new blocks being observed. We then formally define the notion of composability of ADS and show that Charlotte enables intuitive composition of ADSs and their guarantees. Finally, we formally define how integrity and availability attestations influence observer calculations about ADSs.

5.1 Formalizing ADSs and States

A *state* is a set of blocks, and an *ADS* is a set of valid states. For instance, consider the Bitcoin blockchain as an ADS. Every block (except the genesis) in every state features both a proof of work and a reference to a parent block. A Bitcoin state is a tree with an arbitrarily long main chain, and shorter branches. The Bitcoin ADS consists of all such possible states. We illustrate two possible states of such a blockchain in fig. 2.

As another example, consider a simple ADS R representing a permissioned blockchain managed by one Fern server. It has one genesis block, and each successive block must be stored on a Wilbur server and approved by the Fern server. Each state of R features a linear, unbranching chain of blocks, each referencing the previous, back to the genesis block. In each state, each block b in the

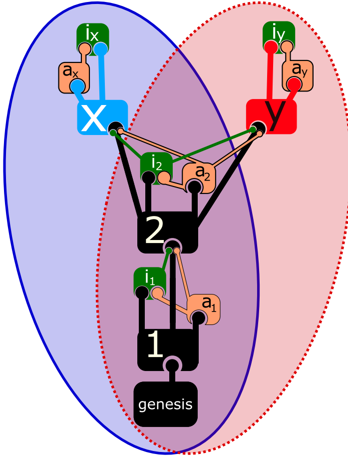

Fig. 9. Two conflicting states in a permissioned blockchain R, in which each chain block b (other than the genesis block) has an integrity attestation i_b and an availability attestation a_b. Note that integrity attestations can themselves reference availability attestations, if, for example, the Fern server wants to show it did not attest to something unavailable (section 4.2). The red dotted oval and the blue solid oval represent two possible *alternative* states of the ADS. Since block x and block y have the same height, a correct Fern server would never create both i_x and i_y. So long as the Fern server is correct, only one of these states can ever be part of the blockweb.

chain is accompanied by an integrity attestation i_b, from the Fern server, and at least one availability attestation a_b. We illustrate two possible states of such a chain in fig. 9.

5.2 Observers and Adversaries

Observers represent principals who use the system. An observer receives blocks from servers and in so doing learns about the current and future states of ADSs in the system. Observers may correspond (but are not limited) to servers, clients, or even people. Formally, an observer is an agent that *observes* an ordered sequence of blocks from the blockweb. On an asynchronous network, different observers may see different blocks in different orders.

Observers define their own failure assumptions, such as who they believe might crash or lie (section 5.3). These assumptions, combined with evidence, in the form of blocks they have observed so far, induce an observer's *belief*: what they think is true about the blockweb now and what is (still) possible in the future.

The failure-tolerance properties of any distributed system are relative to assumptions about possible failures, including possible actions taken by adversaries. Charlotte makes these assumptions explicit for each observer. An observer who makes inaccurate assumptions may not observe the properties they expect of some ADSs. For instance, if more servers are Byzantine than the observer thought possible, data they believed would remain available might not. Alternatively, data structures might lose integrity, such as when two different blocks both appear to occupy the same height on a chain.

5.3 Observers' Beliefs

We characterize a belief α as a set of possible *universes*. This set bounds the believed powers of the adversary: the observer assumes this set includes all possible universes that might occur under the influence of the adversary. Figure 10 illustrates an observer holding a belief, and some universes in that belief.

Each observer has an *initial belief*: the belief it holds before it observes any blocks. For example, an observer who trusts one Fern server to maintain the permissioned blockchain R does not have any universes in its initial belief in which that server has issued two integrity attestations for different blocks at the same height. This belief encodes the observer's assumption that the server's failure is not tolerable. The observer in fig. 10 has such a belief: no universe features two integrity attestations for equal height blocks in R (shown as green squares labeled i_x or i_y).

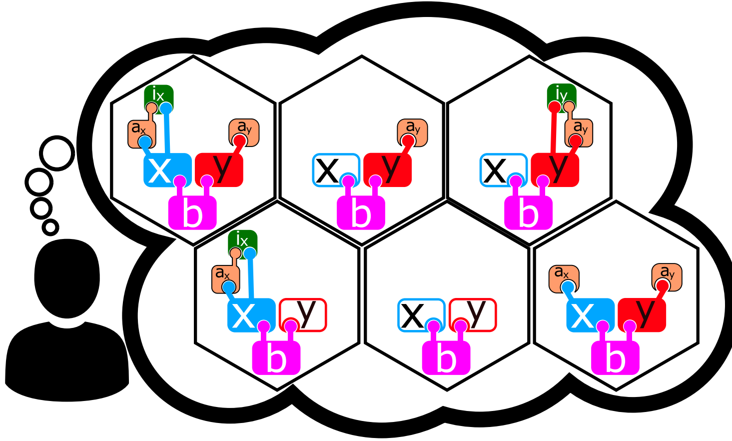

Fig. 10. An *observer* holds a *belief*, which is a set of *universes*, depicted here as hexagons. In each universe U, some blocks in *exist*(U) are shown, with the blocks in *avail*(U) filled in. Note that no block has an availability attestation (drawn as orange squares labeled a) unless it is filled in (meaning it is available): this observer trusts these availability attestations. Here we can imagine block b as part of a chain (such as block 1 or 2 in fig. 9), and blocks x and y as alternative possibilities for b's successor.

In a traditional failure-tolerant system, an observer usually assumes that no more than f participants will fail in some specific way (e.g., crash failures or Byzantine failures). We model such an observer's initial belief as the set of all universes in which no more than f participants exhibit failure behaviors (in the form of blocks issued).

As an observer observes more blocks, it can eliminate universes it no longer believes possible. For instance, observing any block rules out any universes in which that block doesn't exist. If an observer believes in certain attestations, they might rule out some universes after they observe those attestations. In our permissioned blockchain example R, our observer rules out any universes featuring i_y once it has observed i_x, since it does not believe the Fern server would issue integrity attestations for different blocks of the same height on the chain.

5.4 Formalizing Universes

We propose a general model for universes, which offers a great deal of expressive power for representing the beliefs of observers, and is sufficient for all examples in the paper. A universe U has the following components:

(1) A set of blocks that can *exist*, written *exist*(U). These are the blocks that either have already been observed or ever *can* be observed by any observer.
(2) A *strict partial order* $\sqsubset_U$ on *exist*(U). Every observer is assumed to observe blocks in an order consistent with the universal partial order $\sqsubset_U$.
(3) The set of blocks that *are available*, written *avail*(U). These are the blocks that can be retrieved from some server. Any available block must also exist: *avail*$(U) \subseteq$ *exist*(U).

The set *exist*(U) constrains the blocks any observer will observe. It does not model time: an observer's initial belief contains universes representing all possible futures, with all blocks that are possible in each.

Since we are modeling asynchronous systems, the model does not explicitly include the time when blocks are observed, but the ordering $\sqsubset_U$ constrains the times at which different observers

can observe blocks, implicitly capturing a temporal ordering on blocks. This ordering is useful for blockchains like Bitcoin, where observers traditionally do not believe in any universe U unless there is a *main chain* in which each block b is ordered (by $\boxed{\underline{U}}$) before any equal-height block with which b does not share an ancestor fewer than security parameter k (usually 6) blocks away. Further, the *main chain* must forever outpace any other branch. In fig. 2, this belief (with $k = 3$) implies that if a Bitcoin observer believes in a universe U in which both blocks s and c exist, they must be ordered by $\boxed{\underline{U}}$. If Bitcoin's security assumptions are accurate, any two observers must see s and c in the same order.

We make the simplifying assumption in each of our example services that the only availability of interest is permanent: asserting blocks will *forever* be available. We leave more nuanced availability policies to future work.

5.5 Updating Beliefs

An observer starts with some *starting belief*. Recall that if an observer's failure assumptions are correct, we say that the observer is *accurate*: the real universe (the set of blocks that can ever exist, paired with the set that will be available) is in the observer's starting belief. As an observer observes blocks created by Charlotte services, it updates its belief by whittling down the set of universes it considers possible. It can only eliminate universes inconsistent with its observations: those in which blocks it has observed do not exist, or the order in which it has observed the blocks is impossible. For instance, if an observer with belief α observes a block b, clearly b *can exist*, so the observer refines its belief. It creates a new belief α', filtering out universes in which b is impossible:

$$\alpha' = \left\{ \; U \;\middle|\; b \in \mathit{exist}(U) \;\wedge\; U \in \alpha \; \right\}$$

If the observer in fig. 10 were to observe i_x, it would update its belief, retaining only universes U with $i_x \in U$. Of the universes shown, only the leftmost three would remain.

An observer also refines its belief by observation order: If an observer with belief α observes blocks B in total order $<_B$, its new belief becomes

$$\mathit{Possible}(\alpha, B, <_B) \triangleq \left\{ U \;\middle|\; \begin{array}{l} \quad \forall b \in B \,.\, \forall b' \boxed{\underline{U}}\, b \,.\, b' \in B \;\wedge\; b' <_B b \\ \wedge\, B \subseteq \mathit{exist}(U) \\ \wedge\, U \in \alpha \end{array} \right\}$$

An observer making no assumptions believes in all possible universes. Most interesting observers have other assumptions. For example, the observer in fig. 10 trusts that only one integrity attestation for ADS R will be issued for each chain height, so if it observes i_x and removes all universes U without $i_x \in \mathit{exist}(U)$, then no universes with i_y will remain.

5.6 Observer Calculations

An observer with belief α knows a set of blocks B are *available* if they are made available in all possible universes:

$$\forall U \in \alpha .\, B \subseteq \mathit{avail}(U)$$

For example, the observer in fig. 10 trusts availability attestations a_x and a_y (orange squares): it believes in no universe where such attestations reference an unavailable block.

Likewise, an observer with belief α knows a state S of an ADS D is *incontrovertible* if no *conflicting* state S' can exist in any possible universe. Recall that a state of an ADS is a set of blocks (section 5.1). Two states conflict if they cannot be merged to form a valid state: observing one precludes ever observing the other. So a state S of an ADS D is incontrovertible if:

$$\forall U \in \alpha, S' \in D .\, (S \cup S' \in D) \;\vee\; (S' \not\subseteq \mathit{exist}(U))$$

For example, the observer in fig. 10 trusts that only one integrity attestation for ADS R will be issued for each chain height. It does not believe in any universes with both i_x and i_y (shown as green squares). Therefore, if it observes i_x, it knows the state with x has the highest block (shown in blue in fig. 9) is incontrovertible: no conflicting state (such as any including block y) exists in any universe in its belief.

The state of ADS D that an observer with belief α sees as available and incontrovertible is therefore:

$$View(\alpha, D) \triangleq \bigcup \left\{ S \;\middle|\; \begin{array}{l} \quad \forall U \in \alpha, S' \in D.\ (S' \cup S \in D) \vee (S' \not\subseteq exist(U)) \\ \wedge\ \forall U \in \alpha.\ S \subseteq avail(U) \\ \wedge\ S \in D \end{array} \right\}$$

We call this the observer's *view* of the ADS: Charlotte's natural notion of the "current state." So long as an observer's assumptions are accurate, new observations can only cause its view to grow. For example, if the observer in fig. 10 observes both a_x and i_x, then it believes the blue state from fig. 9 is available and incontrovertible. Its view of the permissioned blockchain ADS R features x occupying height 3, and so long as its assumptions are accurate, this will never change.

As another example, suppose a blockchain uses a simple agreement algorithm: a quorum of servers must attest to a block being at a specific height. States consist of a chain of blocks, each with integrity attestations from a quorum. An observer's view will not include any blocks lacking sufficient attestations. The observer assumes that no two blocks with the same height both get a quorum of attestations, so the chain it has viewed must be a prefix of the chain in any future view.

One observer can calculate what another observer's view of an ADS would be, given a set of observations, and communicate by sharing blocks they've observed. Because new observations can only cause a view to grow, one observer can know (at least part of) another communicating observer's view. This what we mean when we say views in Charlotte are *consistent:* two observers can know the other's view of the same data structure, and so the state of a data structure can be, in a sense, global.

5.7 Composability

Recall that a *state* is a set of blocks, and an ADS is a set of states (section 5.1). ADSs in Charlotte have two natural notions of composition: *union* ($\uplus$) and *intersection* ($\Cap$). For illustrating composability, we use a running example: an atomic swap of two cryptocurrencies, maintained by two respective blockchains D and D'.

5.7.1 Union. Intuitively, the union of two ADSs D and D' is all the data in either ADS. As states are sets of blocks (section 5.1), their union is simply the traditional union of sets. Thus, the union ADS is composed of unions of states:

$$D \uplus D' \triangleq \left\{ S \cup S' \;\middle|\; S \in D \ \wedge\ S' \in D' \right\}$$

As a result, given an observer's failure assumptions, its view of the union of two ADS is simply the union of its views of the ADSs:

Theorem 1.

$$\forall \alpha, D. View(\alpha, D \uplus D') = View(\alpha, D) \cup View(\alpha, D')$$

Proof. Follows from the definitions of *View* and $\uplus$. □

In our atomic swap example, all pairs of valid states from each blockchain D and D' are valid states of the new ADS $D \uplus D'$. For an observer, the incontrovertible state of ADS $D \uplus D'$ is the union of the incontrovertible state of each blockchain, as discussed in section 5.6.

5.7.2 Intersection. Intuitively, what we call the *intersection* of two ADSs D and D' is more *trustworthy* than either component ADS: observers need only trust that there won't be conflicts in at least one component data structure. As a result, an observer who believes conflicting states for each component might exist (they do not trust the integrity mechanism of either ADS) might still believe that no conflicting states exist in the intersection. While our notion of *intersection* may not match some readers' intuitions, these intersections are useful because they allow us to build trust from less trustworthy sub-structures.

Each state of the intersection data structure includes exactly one state from one component ADS, and may include multiple states from the other component ADS. Intuitively, this means that the intersection data structure includes states where *one* component ADS has failed its integrity guarantees, and has multiple contradictory states. The integrity properties intersection structures offer are weaker statements, more strongly enforced: two states of the intersection conflict only when they contain conflicting states from *both* component ADSs.

In our blockchain example, this means that the intersection of two blockchains D and D', written $D \Cap D'$, includes states where either D has forked, or D' has forked, but not both. The weaker property (that at least one chain has not forked) is enforced more strongly than the forking properties of D or D': so long as *either* D's or D''s remains unforked, $D \Cap D'$ keeps its integrity.

In general, for ADSs D and D':

$$D \Cap D' \triangleq \left\{ \; S \cup \bigcup W \;\middle|\; (S \in D \land W \in \mathcal{P}(D')) \lor (S \in D' \land W \in \mathcal{P}(D)) \; \right\}$$

where $\mathcal{P}(X)$ is the powerset of X.

As a result, given an observer's failure assumptions, its view of the intersection of two ADS will always contain its view of each component ADS.

Theorem 2.

$$View(\alpha, D) \cup View(\alpha, D') \subseteq View(\alpha, D \Cap D')$$

Proof. Follows from the definitions of *View* and $\Cap$. □

Within each state $S \in D \Cap D'$, a (belief independent) sub-set of the blocks each make up a state of D and a state of D'. The intersection of these can be thought of as *in both ADSs*:

$$InBoth(S, D, D') \triangleq \left\{ \; b \;\middle|\; b \in W \in D \land b \in W' \in D' \land W \cup W' \subseteq S \; \right\}$$

The other blocks in S serve, in a sense, to prove which blocks belong in $InBoth$.

For our running example, the blocks that are part of both chains (*InBoth*) represent transactions atomically committed to both ledgers. These are the natural place to put *cross-chain transactions*: trades involving atomic swap of the two crypto-currencies. Thus, the intersection of the two blockchains is the sequence of cross-chain transactions.

The blocks in $InBoth(S, D, D')$ share the properties of the intersected ADSs. In particular, if an observer trusts both data structures, the blocks it views in *InBoth* will be precisely the blocks it views in both structures.

Theorem 3. *If a belief α contains no universes with conflicting states of D or D', then:*

$$InBoth(View(\alpha, D \Cap D'), D, D') = View(\alpha, D) \cap View(\alpha, D')$$

Proof. We defer the proof to appendix C. □

In our blockchain example, cross-chain blocks remain totally ordered by the blockweb so long as either component blockchain remains totally ordered by the blockweb (a traditional integrity property of blockchains). Furthermore, cross-chain blocks remain available if the blocks of either component blockchain remain available, as discussed in section 5.6.

5.8 Attestation Semantics

Observers use attestations to determine which states of the ADSs they care about are available and incontrovertible. To formally describe the guarantees offered by an attestation x, we give it an interpretation $[\![x]\!]$ that is a *belief*: a set of universes in which x's guarantees are inviolate (section 5.2). Any attestation with a corresponding belief fits into our semantic model, making it extremely flexible. However, this flexibility comes with some cost: since we have no formal syntax for attestations to encode their semantics, it is up to observers to know what the attestations they care about mean. Future work could introduce types of attestations with a formal syntax, and plug them directly into the existing framework. Ultimately, however, attestations are about *trust*: merely encoding certain semantics doesn't guarantee anyone will believe in them.

We divide attestations into two subtypes: availability attestations, which help observers to determine which blocks are sufficiently available (section 3.3.2), and integrity attestations, which help observers to determine which blocks belong in a data structure (section 3.3.3). In principle, and attestation can be in both subtypes, but none of our examples are.

5.8.1 Availability Attestations. Availability attestations are issued by Wilbur servers (section 4.1), and guarantee some blocks are available in some universes. For instance, consider the availability attestation subtype $\tau_{AliceProvides}$. Attestations of this type are blocks of the form $aliceProvides(b)$ (where b is another block). Intuitively, each value states that `Alice` (a Wilbur server) promises to make the specified block b available forever. Thus, all universes U in which $aliceProvides(b)$ exists also have b available:

$$[\![aliceProvides(b)]\!] \triangleq \left\{ U \;\middle|\; aliceProvides(b) \in exist(U) \implies b \in avail(U) \right\}$$

5.8.2 Integrity Attestations. Integrity attestations (section 3.3.3) are issued by Fern servers (section 4.2), and represent proofs guaranteeing commitments to specific states. Fundamentally, this means committing exclusively: integrity attestations guarantee the *non-existence* of specific other integrity attestations, under certain circumstances. While this definition may seem counter-intuitive, it generalizes the notion of *conflict* or *exclusivity* in ADSs. For example, in our permissioned blockchain ADS R, all the integrity attestations for blocks at the same height in any state of R are mutually exclusive. The existence of a state including one integrity attestation disproves all conflicting states, which puts the attestation, and the block it references, in the view of any observer with an appropriate belief.

An integrity attestation guarantees some other blocks *cannot exist* in some universes. For example, consider $\tau_{BobCommits}$, a subtype of integrity attestation with values that are blocks of the form $bobCommits(b)$, which intuitively indicates that `Bob` (a Fern server) promises never to commit to any block other than b. ($\tau_{BobCommits}$ is specific to some slot in some ADS, such as a specific height of a specific blockchain.)

Thus, all universes U in which $bobCommits(b) \in exist(U)$ don't feature $bobCommits(c)$ for any $c \neq b$:

$$[\![bobCommits(b)]\!] \triangleq \left\{ U \;\middle|\; \forall c \neq b.\ bobCommits(c) \notin exist(U) \right\}$$

5.8.3 Composition. Defining attestations this way makes it easy to define observers' beliefs based on which attestations, attestation types, or even participants they trust. For instance, if an observer trusts all attestations with type τ, we define that observer's belief:

$$\alpha = \bigcap_{x:\tau} [\![x]\!]$$

This provides a straightforward definition for what it means to *believe in* a type; it means trusting all attestations of that type.

$$[\![\tau]\!] \triangleq \bigcap_{x:\tau} [\![x]\!]$$

We can also define beliefs that trust only combinations of attestations. For example, if an observer believes a block will be available only if it has observed appropriate attestations of both type τ *and* type σ, we define that belief α as $\alpha = [\![\tau]\!] \cup [\![\sigma]\!]$. Similarly, an observer who believes b is committed only after receiving an attestation of type τ and an attestation of type σ would believe $\alpha = [\![\tau]\!] \cup [\![\sigma]\!]$. Likewise, a more trusting observer who believes b is committed after receiving an attestation of either type τ or σ would believe $\alpha = [\![\tau]\!] \cap [\![\sigma]\!]$. In this way, we can even build up quorums of attestations or attestation types (e.g., $([\![\tau_1]\!] \cup [\![\tau_2]\!]) \cap ([\![\tau_2]\!] \cup [\![\tau_3]\!]) \cap ([\![\tau_1]\!] \cup [\![\tau_3]\!])$).

It is also possible to combine integrity and availability attestation types to define a belief. An observer who trusts attestations of type τ to commit blocks, and attestations of type ρ to ensure their availability would believe: $\gamma = [\![\tau]\!] \cap [\![\rho]\!]$. In this way, we can even define quorums of trusted types.

5.9 Monotonicity

The semantics of attestations are somewhat restricted in that they must be *monotonic*: adding attestations never weakens what statements can be proved: Any belief α must satisfy this monotonicity condition concerning availability attestations:

$$\forall U, W \in \alpha.\ exist(U) \subseteq exist(W) \quad \Rightarrow \quad avail(U) \subseteq avail(W)$$

Since integrity attestation semantics are about what *cannot exist*, the definition of $Possible([\![\tau]\!], B, <_B)$ (from section 5.2) guarantees integrity attestations are *monotonic*: adding more attestations never proves weaker statements:

$$C \subseteq B \Rightarrow Possible(\tau, B, <_B) \subseteq Possible(\tau, C, <_B)$$

6 USE CASES

Charlotte is well-suited to a wide variety of services. Using the semantics from section 5, each service can offer robust and composable guarantees. In addition to various examples discussed earlier, we outline some particularly suitable uses of the Charlotte framework.

6.1 Verifiable Storage

Our Wilbur specification provides a common framework for verifiable storage. Because ADS references include hashes, it is always possible to check that data retrieved was the data referenced. Different Wilbur servers might offer different levels of durability (e.g. RAM, Hard Drive), or different locations while providing exactly the same API. Furthermore, availability attestations are a natural framework for proofs of retrievability [13].

Wilbur servers can even be layered: one server might offer a service wherein it erasure-codes blocks and stores shares on specific other servers, and then combines each of their availability attestations into one attestation for the block, proving that it remains available so long as a quorum of the shares remain available. Such layering does not require the share-storing servers to be aware of the erasure coding.

6.1.1 Queries. In addition to `SendBlocks` and `RequestAvailabilityAttestation`, Wilbur servers may offer other interfaces. Service designers may wish to implement query systems for retrieving relevant blocks. We created one such example interface, the `WilburQuery` RPC (fig. 11). Given a

```
message WilburQueryInput {
  oneof wilburquery_oneof {
    Reference reference;
    Block fillInTheBlank;
  }
}
message WilburQueryResponse {
  oneof wilburqueryresponse_oneof {
    string errorMessage;
    repeated Block block;
  }
}
service WilburQuery {
  rpc WilburQuery(WilburQueryInput) returns (WilburQueryResponse) {}
}
```

Fig. 11. WilburQuery Specification.

`Hash` as input, `WilburQuery` returns the block with that hash. If the server does not know of such a block, our example implementation waits until one arrives.

`WilburQuery` also provides a kind of fill-in-the-blank match: if sent a block with some fields missing, `WilburQuery` returns all stored blocks that match the input block in the provided fields. For example, we might query for all blocks with a field marking them as a member of a certain ADS.

6.2 Timestamping

Timestamps are a subtype of integrity attestation. We implemented a Signed Timestamp type, wherein the signer promises that they have seen specific hashes before a specific time. Our timestamping Fern servers can use *batching*: they wait for a specific (configurable) number of new requests to arrive before issuing a timestamp block referencing all of them. In fact, since hash-based references represent a *happens-before* relationship [53], timestamps are transitive: if timestamp a references timestamp b, and b references c, then a effectively timestamps c as well.

We recommend that batch timestamp blocks themselves should be submitted to other Timestamping Fern servers. This allows the tangled web of timestamp blocks to very quickly stamp any block with timestamps from exponentially many servers, making them very high-integrity.

6.3 Conflict-Free Replicated Data Types

Charlotte, and ADSs in general, work well with CRDTs, especially Operation-Based Commutative Replicated Data Types (CmRDTs) [78]. CmRDTs are replicated objects maintained by a group of servers. Whenever a new operation originates at any server, all known operations on that object on that server are said to *happen before* it. Then the operation asynchronously propagates to all other servers. Thus, the set of operations known to any particular server are only partially ordered. The *state* of a CmRDT object is a deterministic function of a set of known operations (and their partial order). For example, a CmRDT implementation of an insert-only Set might feature the `insert` operation, and its state would be the set of all arguments to known `insert` operations.

In Charlotte, CmRDT operations can naturally be expressed as blocks, with *happens-before* relationships expressed as references. Since references are by hash, it is impossible for an adversary to insert a cycle into the graph of operations. The states of a CmRDT can be formally expressed as all possible sets of operations with all possible partial orderings.

Aside from whatever credentials one needs to authorize an operation, CmRDTs do not need integrity attestations. Availability attestations are still useful.

The blockweb as a whole is a CmRDT: Its state is the DAG of all blocks, and every block is an operation adding itself to the state. Other than the blockweb itself, however, we have not implemented any interesting CRDTs yet.

6.4 Composition

Charlotte makes ADSs easy to compose (section 5.7). Representing references as hashes means blocks in one ADS can reference blocks in another. Charlotte's hash-based references become the new URLs. For instance, a Timestamping server might maintain a chain of timestamp blocks, which reference other blocks people want timestamped (Git commits, payments, documents, etc.). A Git-style repository might (like Git *submodules* [17]) reference earlier commits in another repository. (either because one is a fork of the other, or one has merged in code from another) without having to copy all of the data onto both servers. A blockchain could reference a Git commit as a smart contract, instead of hosting a separate copy of the code [26]. A single block of data, stored on some highly available servers, could be referenced without duplication from Torrent-style filesharing services, Git-style repositories, and blockchains.

Attestations allow building high-integrity ADSs out of lower-integrity ones (section 5.7). For instance, the blocks appended to two chains will themselves form a linear order that can only fork if *both* component chains fork. Users may want to put especially important blocks on many different chains, the way they want many different witnesses for important legal transactions.

Likewise, we can build low-integrity ADSs out of higher integrity ones (section 5.7). If a set of blockchains each manage independent tokens, and sometimes share blocks (for atomic trades of tokens), then together all the chains form a DAG. If *any* chain in the DAG is corrupted, then the supply of that token may not be conserved: the DAG as a whole is lower integrity than any one chain. The "integrity of the marketplace" is distinct from the integrity of any one token.

6.5 Entanglement

Some attestations, such as timestamps (section 6.2), and proofs of work (section 7.2.1), implicitly lend integrity to everything in a block's ancestry. When many ADSs reference each other's blocks, these recursive attestations can make some forms of fraud very difficult. For example, if many services regularly reference past timestamps, and many services request timestamps from a variety of servers, it quickly becomes infeasible to falsely claim a block *did not* happen before a given time, since that would require corrupting many services. Charlotte promotes this useful entanglement.

7 BLOCKCHAINS AS ADSS

Charlotte is an ideal framework for building new blockchains and related services. In the simplest sense, a blockchain is any path through the blockweb. However, most existing blockchain-based services are much more complex.

Like all ADSs, a blockchain needs *integrity* and *availability*. Here, integrity means that an observer's view (section 5.2) always features a *main chain*, in which no two blocks ever have the same *height*. Availability means that once an observer observes a main-chain block at a height, that block remains available for download indefinitely. Existing blockchain systems already effectively provide integrity and availability attestations, phrased as proofs of work, proofs of stake, etc. Charlotte makes these proofs more explicit, without limiting the attestation types a service can use.

7.1 Separating Availability and Integrity

With few exceptions [63], existing blockchain systems require all integrity servers (e.g., miners, and consensus nodes) to store all blockchain data. This total replication is fundamentally inefficient. For example, traditional Byzantine consensus tolerating f failures needs $3f+1$ participants, while a storage system tolerating f failures needs only $f+1$ participants. If blockchain systems separated storage and consensus duties, they could store about 3 times as much, with the same failure assumptions.

Charlotte makes it easy to separate availability from integrity. Servers issuing availability attestations must store the data, but those issuing integrity attestations need not.

For example, if one were to build something like Ethereum in Charlotte, what Ethereum calls *block headers* would themselves be integrity attestations, and the Merkle root in each would instead be a reference (or collection of references) to blocks stored on Wilbur servers. This makes it natural to search and retrieve block headers and portions of state, without splitting apart blocks or downloading the whole chain.

7.2 Integrity Mechanisms

Different blockchains have used a variety of mechanisms to maintain the integrity of the chain [15, 58, 62, 69]. To demonstrate the flexibility of Charlotte, we have implemented a few example mechanisms in small-scale experiments.

7.2.1 Nakamoto (Proof of Work). We can represent a Bitcoin/Ethereum-style blockchain as an ADS D whose states are trees of proof-of-work blocks. Observers assume that a main chain will forever outpace other forks: that is, become longer, faster (see section 5.4).

7.2.2 Agreement. Some blockchain-based services only require agreement: they lose liveness if two *potentially* valid blocks are proposed for the same height [55, 75, 84]. For instance, if a chain represents a single bank account, and potentially valid blocks represent transactions signed by the account holder, then honest account holders should never sign two transactions unordered by the blockweb.

Agreement Fern servers are simple to implement. When a server attests to a block, it promises never to attest to any conflicting block. The semantics of each agreement attestation do not feature any universes where two conflicting blocks both have an attestation from the issuing server. Observers can construct quorums of trusted servers, as in section 5.8.2. A block appears in an observer's view when the observer has observed enough attestations: committing a conflicting block would require too many parties to break their promises.

7.2.3 Heterogeneous Paxos. Heterogeneous Paxos (HetPax) is our own consensus algorithm [81] generalizing Leslie Lamport's Byzantine Paxos [54]. HetPax allows each pair of observers (*learners* [54]) to specify the set of universes (section 5.2) in which they must agree. They define these universes in terms of which Fern servers (*acceptors* [54]) are *safe* (not Byzantine) and *live* (not crashed) in each. In this way, we support heterogeneous servers, heterogeneous (or "mixed" [77]) failure models, and heterogeneous observers. In the symmetric case, when all observers have the same failure tolerance, HetPax reduces to regular Byzantine Paxos [81].

For each observer, HetPax forms quorums of participants whose attestations are necessary to put a block in the observer's view. Generally speaking, two observers will agree so long as all their quorums intersect on a safe participant.

7.3 Blocks on Multiple Chains

In general, nothing prevents a single block from being part of multiple chains. It simply requires the integrity attestations for each chain. For example, if one blockchain represents records of events that have happened to a specific vehicle (crashes, repairs, ...), and another represents repairs a specific vendor has performed, it makes sense to append the record of a specific repair to both chains. The record (a block) could reference the previous blocks on each chain, and the next blocks on each would in turn reference it. Each chain's integrity mechanism would have to attest to the block, and references to the block could carry both sets of attestations to let readers know it is in both ADSs.

7.3.1 Atomicity. Sometimes, such block appends need *atomicity.* For example, suppose one blockchain represents the cryptocurrency RedCoin, and another represents the cryptocurrency BlueCoin. Alice wants to give Bob one RedCoin in exchange for one BlueCoin. This represents two transactions: one on each chain. It is crucial that either *both* happen, or *neither* do. Otherwise, Alice might give Bob a RedCoin and get nothing in return. We want to commit both transactions together, *atomically.*

Formally, we say that an attestation i_x *commits* a block x to an ADS D for an observer with starting belief α, who has observed blocks B in order $<_B$, if the observer sees x in its view of D after observing i_x, but not before:

$$commits(i_x, D, \alpha, B, <_B) \triangleq x \notin View(Possible(\alpha, B, <_B), D) \land x \in View(Possible(\alpha, B \cup \{i_x\}, <_B : i_x), D)$$

Where $<_B : i_x$ is the total order $<_B$ extended with i_x as the last element.

7.3.2 Mutual Subtypes of Attestations. To atomically commit one block to multiple ADSs, an observer requires a single integrity attestation which represents a simultaneous commitment of all of them: before observing the attestation, the block is not in the observer's view of any of the ADSs, but after observing the attestation, the block is in the observer's view of all of the ADSs involved. We often think of the set of attestations that can commit a specific block to a specific data structure as *types.* Viewed this way, an attestation that can commit to multiple data structures is a *mutual subtype.*

For instance, suppose committing a block to data structure D requires an attestation signed by every member of some quorum of servers Q. Committing the same block to data structure D' requires an attestation signed by every member of some quorum of servers Q'. It would then be reasonable for an attestation signed by every member of $Q \cup Q'$ to be sufficient to commit b to both D and D'.

Not all pairs of ADSs have integrity attestations that can atomically commit to both. However, we created mutual subtypes for our HetPax blockchains. The quorum necessary for an attestation with mutual subtype is the union of one quorum from each component type. In other words, to make an attestation sufficient for an observer decide to commit to both chains, you need all the participants it would take to make an attestation for each of the chains. With this construction, we can atomically commit a single block onto multiple HetPax chains.

7.4 Linearizable Transactions on Objects

It can be useful to model state as a collection of stateful *objects*, each of which has some availability and integrity constraints [74]. We can model objects as a chain of blocks, defined by availability and integrity attestations upholding these constraints. For instance, if an object must be consistent and available so long as 3 of a specific 4 servers are correct, each block should have "store forever" availability attestations from 2 servers, and integrity attestations from 3 stating that they'll never attest to any other block in that slot.

Each block represents a state change for each of the objects represented by chains of which the block is a part. In other words, the blocks are *atomic* (or *ACID*) *transactions* in the database sense [34]. A collection transactions is guaranteed to have a consistent, serial order so long as the chains maintained for each of the objects they touch are consistent. For a given observer, the transactions involving objects which that observer assumes to be linearizable have a serial order so long as that observer's assumptions are accurate. Furthermore, two accurate observers can never see two transactions oppositely ordered.

This gives programmers a natural model for atomic transactions across object-chains with different integrity and availability mechanisms, which would be useful for services from banking to supply chain tracking. Transactions can involve any set of objects, so long as their integrity mechanisms have a mutual subtype for their attestations (section 7.3).

7.4.1 Banking. We can imagine bank accounts as linearizable objects, with state changes being deposits and withdrawals to and from other bank accounts, signed by appropriate parties. We can model such accounts in Charlotte. Each bank maintains some integrity mechanism (Fern servers) to ensure accounts' state changes are totally ordered, which prevents double-spending. Likewise, each bank maintains some Availability mechanism (Wilbur servers), ensuring transactions relevant to their customers' accounts aren't forgotten. Each transaction is thus a block shared by two chains, and must be committed atomically onto both chains.

When considering how "trustworthy" the money in an account is, what matters is the integrity of the ADS featuring the full ancestry of all transactions in the account. To ensure the trustworthiness of their accounts, banks may issue their own integrity attestations for all transactions in the causal past of transactions involving that bank. This requires checking that ancestry for any inconsistencies with anything to which the bank has already attested. This ensures any observers trusting the bank's attestations have consistent view (section 5.2), but cannot guarantee that observers trusting different banks have the same view.

An "attest to the complete history" approach is analogous to auditing the full finances of everyone with whom you do business for every transaction. In reality, much of the time, banks effectively trust each other's attestations. This allows much faster transaction times, with weaker guarantees.

7.4.2 Supply Chain Tracking. Much like bank accounts, we can imagine each good in a supply chain as a linearizable object. Transactions may involve decreasing or destroying some goods to increase or create others. For example, a transaction might feature destroying 10 kg from a case of grapes to add 9 kg to a vat of juice, and 1 kg to a bin of compost. As with banking, each good is only as "trustworthy" as the ADS featuring its complete ancestry, and audits attesting to past transactions can increase this trustworthiness.

8 IMPLEMENTATION

Our full open-source Charlotte spec, with all example types and APIs, is 298 lines of gRPC (mainly protobuf) [33]. We implemented proof-of-concept servers in 3833 lines of Java [32] (excluding comments and import statements), with a further 2282 lines of unit tests and experiments. Our code is on Github [80].

8.1 Wilbur Servers

By default, our example Wilbur servers store all blocks received in RAM forever. They are not meant to be optimal, but they are usable for proof-of-concept services. The only type of availability attestation we have implemented is one in which the Wilbur servers promise to store the block indefinitely. This attestation proves that the block is available as long as the Wilbur server is functioning correctly. Any observer can choose which crash failures they believe possible, and

```
1  message PublicKey {
2    message EllipticCurveP256 {
3      bytes byteString;
4    }
5    oneof keyalgorithm_oneof {
6      AnyWithReference any;
7      EllipticCurveP256 ellipticCurveP256;
8    }
9  }
10 message CryptoId {
11   oneof idtype_oneof {
12     AnyWithReference any;
13     PublicKey publicKey;
14     Hash hash;
15   }
16 }
17 message Signature {
18   message SHA256WithECDSA {
19     bytes byteString;
20   }
21   CryptoId cryptoId;
22   oneof signaturealgorithm_oneof {
23     AnyWithReference any;
24     SHA256WithECDSA sha256WithEcdsa;
25   }
26 }
```

Fig. 12. Signature Specification. We include `Any` types for extensibility, as well as default built-in types, like `Sha256WithECDSA`. Note that the `message` keyword defines a type in the local scope.

consider only those blocks which will survive on at least one server "available." For instance, if an observer believes that at least one server of $\{A, B, C\}$ will survive, and it sees attestations from each of A, B, and C for block b, then it believes block b is available. This is how our experiments use multiple Wilbur servers for increased availability.

Our Wilbur servers can be configured with a list of known peers, to whom they will relay any blocks they receive and any attestations they create. This is easy to override: servers can be made to relay blocks to any collection of peers.

Each of our Wilbur servers implement the WilburQuery service (section 6.1.1), which allows clients to request and download blocks. Our Wilbur servers can do fill-in-the-blank pattern matching on all implemented block types. When not under load, the WilburQuery service imposes no overhead on other services.

8.2 Version Control

We implemented a simple simulation of Git [91]. Our servers are not fully functional version-control software, as they do not implement file-diffs and associated checks, which are irrelevant for our purpose.

The types for our version-control ADS are described in fig. 13. We created a block subtype, `SignedGitSimCommit`, representing a specific state of the files tracked. Each block features a signature, comment, and hash of the state. It can be an *initial* commit, in which case it has no parents, but does include bytes representing the full contents of the files being tracked. Alternatively, it can have some number of parent commits, each with a reference and a file diff.

```
message SignedGitSimCommit {
  message GitSimCommit {
    message GitSimParents {
      message GitSimParent {
        Reference parentCommit;
        bytes diff;
      }
      repeated GitSimParent parent;
    }
    string comment;
    Hash hash;
    oneof commit_oneof {
      bytes initialCommit;
      GitSimParents parents;
    }
  }
  GitSimCommit commit;
  Signature signature;
}

message Block {
  oneof blocktype_oneof {
    AnyWithReference any;
    string protobuf;
    SignedGitSimCommit signedGitSimCommit;
  }
}

message IntegrityAttestation {
  message GitSimBranch {
    google.protobuf.Timestamp timestamp;
    string branchName;
    Reference commit;
  }
  message SignedGitSimBranch {
    GitSimBranch gitSimBranch;
    Signature signature;
  }
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    SignedGitSimBranch signedGitSimBranch;
  }
}
```

Fig. 13. Git Simulation integrity attestation Specification. We include Any types for extensibility, and provide types like SignedGitSimBranch as options. Note that the message keyword defines a type in the local scope, and that the Signature type is defined in the full Charlotte spec [80].

```
1  message IntegrityAttestation {
2    message TimestampedReferences {
3      google.protobuf.Timestamp timestamp;
4      repeated Reference block;
5    }
6    message SignedTimestampedReferences {
7      TimestampedReferences timestampedReferences;
8      Signature signature;
9    }
10   oneof integrityattestationtype_oneof {
11     AnyWithReference any;
12     SignedTimestampedReferences sigTimeRefs;
13   }
14 }
```

Fig. 14. Timestamping integrity attestation Specification. We include `Any` types for extensibility, and provide `SignedTimestampedReferences` as an option. Note that the `message` keyword defines a type in the local scope, and that the `Signature` type is defined in the full Charlotte spec [80].

A Version Control Fern server tracks the current commit it associates with each *branch* (strings), and issues integrity attestations that declare which commits are on which branches. A correct Fern server should never issue two such attestations for the same branch, unless the commits they reference are ordered by the blockweb. In other words, each new commit on a branch should follow from the earlier commit on that branch; it cannot be an arbitrary jump to some other files. Formally, in all states S of a branch ADS B, all commit blocks are totally ordered by the blockweb. Our example servers enforce this invariant [80].

Furthermore, version control `SignedGitSimBranch` integrity attestations prove that a given server, and by extension whomever controls it, approves of a given commit. Formally, we can define a data structure V consisting of only commits with attestations from appropriate authors. For instance, every state in the *Harry Potter* branch might include an integrity attestation signed by J. K. Rowling for every commit. An observer's view of the data structure would exclude any possibly spurious commits featuring, say, additional chapters of text, until they observed an appropriate attestation signed by Rowling.

Fern servers can have other reasons to reject a request to put a commit on a branch. Perhaps they accept only commits signed by certain keys. When a client issues a request, they can include attestation references. A Fern server can demand that clients prove a commit is, for instance, stored on certain Wilbur servers before it agrees to put it on a branch. The Wilbur servers need not even be aware of the Git data types.

Our version control implementation can use the same Wilbur servers as any other service. In fact, separating out the storage duties of Wilbur from the branch-maintaining duties of Fern allows our Charlotte-Git system to divide up storage duties for large repositories, much like git-lfs [30].

8.3 Timestamping

Timestamps are a subtype of integrity attestation. Each timestamp includes a collection of references to earlier blocks, the current clock time [44], and a cryptographic signature.

Our Timestamping Fern servers timestamp any references requested, using the native OS clock. By default, they issue a timestamp immediately for any request, and do not need to actually receive the blocks referenced. Because references contain hashes, the request itself guarantees the block's existence before that time.

Our Timestamping Fern servers also implement *batching*. Every 100 (configurable at startup) timestamps, the Fern server issues a new timestamp, referencing the blocks it has timestamped since the last batch. Each server then submits its batch timestamp to other Fern servers (configurable at startup) for timestamping. Since timestamps are transitive (if a timestamps b, and b references c, then a also timestamps c), blocks are very quickly timestamped by large numbers of Fern servers. This allows services to quickly gather very strong timestamp integrity.

Formally, an observer with initial belief α can define a data structure T_α consisting of blocks with timestamps they trust. For example, if an observer believes that at least one timestamping server of A, B and C is honest, then any trio of timestamps (one signed by A, one signed by B, and one signed by C) coupled with their mutual ancestors in the blockweb form a valid state of the observer's time-stamped data structure. The observer believes that all the (non-timestamp) blocks in that state happened before the latest timestamp.

8.4 Blockchains

In principle, any path through the blockweb is a blockchain (section 7). Blockchain attestations prove that blocks in the chain are totally ordered, so long as some integrity mechanism is correct. Formally, each blockchain is a data structure C with each state being a totally ordered sequence of blocks, and a set of integrity attestations serving as evidence for that ordering. An observer with initial belief α_C that trusts in the integrity mechanism used does not believe in any universes in which two blocks are committed to the same height. Therefore, once such an observer has observed all the blocks from some state in C, it doesn't believe in any universes where conflicting states (featuring different blocks committed to the same height) exist.

We implemented Fern blockchain servers using three very different integrity mechanisms (section 7.2). We used some of these servers to demonstrate the advantages of separating integrity and availability mechanisms (section 7.1), and blockchain composition: we put blocks on multiple chains (section 7.3).

8.4.1 Agreement. Our Agreement Fern servers keep track of each blockchain as a *root* block, and a set of *slots*. Each slot has a number representing distance from the root of the chain.

Our Agreement Fern servers use the `SignedChainSlot` subtype of integrity attestation (fig. 15) to indicate that they have committed a block. The attestation features a cryptographic signature, and references to a chain's *root*, a slot number, and the block in that slot. This serves as a format for both requests and attestations. Each request is simply an `IntegrityAttestation` with some fields (like the cryptographic signature) missing. While it is possible to encode this request format in the `IntegrityPolicy`'s `any` field, we provide the `fillInTheBlank` option as a convenience.

The Agreement Fern servers are configured with parameters describing which requests they can accept, in terms of requirements on the reference to the proposed block and its parent. Once a correct Agreement Fern server has attested that a block is in a slot, it will *never* attest that a different block is in that slot. For instance, to configure a blockchain using quorums of 3 Agreement Fern to approve each block, we require that each request's `parent` Reference include 3 appropriate integrity attestations.

Our Agreement Fern servers make it easy to separate integrity and availability duties (section 7.1). To ensure that a block is available before committing it to the chain, we require a `block` Reference to include specific availability attestations from Wilbur servers. The number of Wilbur servers used must be more than the number of failures tolerated.

8.4.2 Nakamoto. Nakamoto, or *Proof-of-Work* Consensus is the integrity mechanism securing Bitcoin [69] (modeled formally in section 5.4). In Bitcoin, *miners* create proofs of work, which are stored by *full nodes*. With the Simplified Payment Verification protocol, *clients* submit a transaction,

```
message IntegrityAttestation {
  message ChainSlot {
    Reference block;
    Reference root;
    uint64 slot;
    Reference parent;
  }
  message SignedChainSlot {
    ChainSlot chainSlot;
    Signature signature;
  }
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    SignedChainSlot signedChainSlot;
  }
}
message IntegrityPolicy {
  oneof integritypolicytype_oneof {
    AnyWithReference any;
    IntegrityAttestation fillInTheBlank;
  }
}
```

Fig. 15. Agreement integrity attestation Specification. We include `Any` types for extensibility, and provide `SignedChainSlot` as an option. Note that the `message` keyword defines a type in the local scope, and that the `Signature` type is defined in the full Charlotte spec [80].

```
message IntegrityAttestation {
  message NakamotoIntegrityInfo {
    Reference block;
    Reference parent;
  }
  message NakamotoIntegrity {
    NakamotoIntegrityInfo info;
    uint64 nonce;
  }
  oneof integrityattestationtype_oneof {
    AnyWithReference any;
    NakamotoIntegrity nakamotoIntegrity;
  }
}
```

Fig. 16. Nakamoto integrity attestation Specification. We include `Any` types for extensibility, and provide `NakamotoIntegrity` as an option. Note that the `message` keyword defines a type in the local scope. [80].

and retrieve the *block headers* (proofs of work and Merkle roots) of each block in the chain from full nodes [69]. Each client can use these to verify that its transaction is in the chain (has integrity).

We implement miners as Fern servers, which produce integrity attestations bearing proofs of work, taking the place of block headers. Wilbur servers take the place of full nodes, and store blocks, including integrity attestations. For simplicity, our implementation assumes one transaction per block, so clients generate blocks, and request attestations. When a client receives an integrity attestation (fig. 16), it can retrieve the full chain from Wilbur servers.

With Simplified Payment Verification (SPV) [69], clients traditionally try to collect block headers until they see their transactions buried "sufficiently deep" in the chain. For simplicity, our Fern servers delay responding to the client at all until the client's block has reached a specified (configurable) depth. Regardless, clients can collect integrity attestations from Wilbur servers until they are satisfied with the offered integrity.

Our implementation of Nakamoto consensus offers a more precise availability guarantee than Bitcoin does. Nakamoto Fern servers demand availability attestations with any blocks submitted, ensuring that before a block is added to the chain, it meets a (configurable) availability requirement.

Formally, the states of a Nakamoto style blockchain D are trees of proof-of-work integrity attestations, each with a root at a specific origin block.

The height of a block $height(b)$ is simply its graph distance from the origin block. To encode an observer's belief α that a Nakamoto style blockchain D will not fork by more than k blocks, we require that in every universe U in α, there is some "main chain" $m(U)$ of blocks which will always be observed before any other blocks of equal height, unless they share an ancestor within k blocks. These blocks store (and totally order) the transactions representing the state of the chain. $m(U)$ must be totally ordered to prevent, for instance, double-spending cryptocurrency. Specifically, the main chain has a block at each height in any state that exists:

$$(U \in \alpha \ \wedge \ b \in S \in D \ \wedge \ S \subseteq exist(U)) \Rightarrow \exists a \in m(U) \cap S.\ height(a) = height(b)$$

The main chain has exactly one block at each height:

$$a, a' \in m(U) \wedge a \neq a' \Rightarrow height(a) \neq height(a')$$

With the exception of the origin (height 0) block, each block a in the main chain specifies a unique predecessor a'. We write this $a \prec a'$. The main chain is all connected via this relationship:

$$a \in m(U) \wedge a' \in m(U) \wedge height(a) = height(a') + 1 \Rightarrow a \prec a'$$

To specify that a' is an ancestor of a (via the predecessor relationship), within a maximum distance of k, we write $a \prec^{\leq k} a'$. Every block either within k blocks of the main chain, or observed after a main-chain block of equal height:

$$(U \in \alpha \ \wedge \ b \in S \in D \ \wedge \ S \subseteq exist(U)) \Rightarrow \exists a \in m(U).b \prec^{\leq k} a \ \vee \ height(a) = height(b) \wedge a \boxed{U}\ b$$

Within any α that meets these constraints, no universe exists in which a fork of more than k blocks ever out-paces the main chain. Two correct observers with starting belief α will always agree on the main chain in their views (although one may have more recent information, and thus be a superset of the other). This precisely models the assumptions Bitcoin uses to prevent double-spending the same cryptocurrency.

8.4.3 Prism. The Prism [6] blockchain protocol uses Nakamoto-style longest-chain consensus as a building block. It achieves near physical-limit performance for throughput and latency while maintaining security in presence of Byzantine actors. The central idea is the observation that a Bitcoin block serves two purposes: First, it proposes a set of transactions and second, it adds integrity to its ancestors. Prism makes these roles explicit by having $n+1$ proof-of-work chains—one

```
message NakamotoIntegrityInfo {
  Reference block;
  Reference parent;
  VoteInfo voteInfo;
}
message VoteInfo {
  int32 chain = 1;
  repeated Hash endorsedProposals = 2;
}
```

Fig. 17. Prism integrity attestation Specification. [80].

proposal chain for mining blocks that propose transactions and n voter chains for mining blocks that endorse proposal blocks. The ledger is constructed by taking into account the proposal blocks with maximum votes at each level in the proposal chain.

In Charlotte, attestations and beliefs (representing policies for issuing attestations) allow for a natural encoding of Prism. We implement Prism by building on top of our Nakamoto implementation (section 8.4.2). The integrity attestation generated by Fern servers for the proposal chain is similar to the Nakamoto implementation. Like before, the Fern servers demand (configurable) availability attestations before issuing these integrity attestations. For mining on the voter chains, the Fern servers issue integrity attestations with additional information about the endorsed proposals (fig. 17). Here, the candidate blocks for receiving endorsements are only the proposal blocks for which an integrity attestation has been issued previously.

The states of a Prism data structure D feature a state for each of the n voter chains $V = \{V_1, \dots, V_n\}$:

$$S \in D \Rightarrow \forall 1 \leq i \leq n \,.\, \exists S_i \in V_i \,.\, S_i \subseteq S$$

Furthermore, each state S features a state from a "proposer" tree P:

$$S \in D \Rightarrow \exists S' \in P \,.\, S' \subseteq S$$

P is a tree, so other than the root block, each block b specifies a unique parent b' in the tree, written as $b \prec b'$. Some other properties of P being a tree include:

$$b \in S \in P \wedge b \prec b' \Rightarrow b' \in S$$

$$S, S' \in P \implies (S \cup S') \in P$$

Leaves of P are the *plurality* vote winners of the voter chains for each height of the proposer tree. We can formalize whether a block p wins a plurality vote, and thus is part of the main PRISM chain $m(D)$ as:

$$\left(\begin{array}{l} \quad S \in D \\ \wedge\ L \subseteq [1,\dots,n] \\ \wedge\ \forall i \in L \,.\, m_i(S) \text{ votes for } p \\ \wedge\ \nexists p' \neq p; L', L'' \subseteq [1,\dots,n].\left(\begin{array}{ll} & height(p') = height(p) \\ \wedge & |L' \cup L''| \geq |L| \\ \wedge & \forall i \in L'.m_i(S) \text{ votes for } p \\ \wedge & \forall i \in L''.m_i(S) \text{ does not vote at } height(p) \end{array}\right) \end{array}\right) \Leftrightarrow p \in m(D)$$

(where $m_i(S)$ represents the main chain for voter chain i in state S, and a chain "votes for" a value if a vote for that value appears prior to any other vote for any other value of the same height.)

For an observer with belief α to believe their view of a Prism data structure D is incontrovertible, they must define which election outcomes they believe are still possible based on their observations.

Furthermore, we must ensure that two observers who believe subsets of α will agree on the ordering of any proposal blocks in their view. These involve a complex notion of which blocks they believe will be observed before which others.

To begin with, we use $\alpha_k(V_i)$ to refer to a belief that voter chain V_i will not fork by more than k blocks (as discussed in section 8.4.2). An observer likely believes that no voter chain will ever fork by more than some large bound k_Ω blocks, so we can limit our beliefs to:

$$\alpha \subseteq \bigcap_{V_i \in V} \alpha_{k_\Omega}(V_i)$$

This is sufficient for an observer who is content to wait for every vote to be fully determined before deciding which proposer blocks win an election. However, part of Prism's speed comes from the ability to do better than that: it is possible to call the winner of an election when *enough* votes are known.

Let $\alpha_k^j(V_i)$ be the belief that voter chain V_i will not fork by more than k blocks at height j (building on our definition in section 8.4.2). That is, if a branch diverges from the main chain at height j, then the main-chain blocks of height $j+k$ and higher must always be observed before their branch counterparts of equivalent height.

Let $P(\beta)$ be the probability our observer assigns to belief β being true. Note that in general, we have $k' > k \Rightarrow P\left(\alpha_{k'}^j(V_i)\right) > P\left(\alpha_k^j(V_i)\right)$.

Suppose an observer has observed blocks B in order $<_B$, including the main chain of each V_i up to height $height(i, B)$. Consider an election outcome in which each $j(i, h, B)$ is the height within each voter chain V_i that it casts its vote concerning which proposal block should have height h in the proposer chain. The observer believes the probability of this election outcome among voter chains $Q \subseteq V$ (no votes overturned) is:

$$P(h, B, <_B, Q) \triangleq \prod_{V_i \in Q} P\left(\alpha^{j(i,h,B)}_{height(i,B)-j(i,h,B)}\right)$$

However, the only thing that matters for Prism is which block wins each election, not a specific outcome. For the same observer, we can define a probability that the observed winner will not be overturned. Let $w(h, B)$ be the plurality winner at height h of main voter chains in blocks B, with the second-place candidate winning $c(h, B)$ votes.

$$P_{win}(h, B, <_B) \triangleq \sum_{Q \subseteq V} (\text{if } \leq (c(h,B) + |V| - |Q|) \text{ chains in } Q \text{ vote for } w(h,B) \text{ then } 0 \text{ else } P(h, B, <_B, Q))$$

The key requirement for a Prism observer's belief, then, is that for some probability threshold ϵ, no possible universe should allow for an observation in which a proposal wins with probability $\geq \epsilon$, but loses in the light of further observations.

$$\left(\begin{array}{ll} & U \in \alpha \\ \land & S \in D \\ \land & B \subseteq S \subseteq exist(U) \\ \land & <_B \text{ consistent with } \boxed{U} \\ \land & P_{win}(h, B, <_B) \geq \epsilon \end{array}\right) \Rightarrow w(h, B) \in S$$

This constraint on α is sufficient to prove the ordering properties of Prism. So long as such a belief α is not violated, committed proposal blocks b permanently become part of an observer's view, and no conflicting proposal blocks are committed. Furthermore, two observers believing in α will not arrive at conflicting views:

Prism Ordering Property So long as α is accurate (the real universe is in α), if an observer with starting belief α satisfying these constraints observes p at height h in its view of the proposer chain, then for all observers with any starting belief α' satisfying the same constraints, no block other than p has height h.

Proof Sketch: Assume belief α is accurate: the "real universe" U_r is indeed in α. Suppose an observer s with starting belief α observes blocks $S \subseteq exist(U_r)$ (where $S \in D$), in order $<_S$ consistent with $\boxed{U_r}$, and so now its belief is β, where $U_r \in \beta \subseteq \alpha$. Consider any proposal block p, with height h, in the view of observer s: $p \in View(\beta, D)$.

Furthermore, consider another observer s', with starting belief α. Suppose s' observes blocks $S' \subseteq exist(U_r)$ (where $S' \in D$) in order $<_{S'}$ consistent with $\boxed{U_r}$, and so now its belief is β', where $U_r \in \beta' \subseteq \alpha$. Consider any proposal block p', with height h (the same height as p), in the view of observer s': $p' \in View(\beta', D)$.

To ensure that accurate observers observe the same order of proposal blocks, it suffices to show that $p = p'$. Given our definitions above, $p = w(h, S)$, and $p' = w(h, S')$.

By the definition of the states of the Prism data structure D, there must exist some state $S'' \in D$ which includes all the blocks in all the *voter chains* of both views, and in which some block has won the plurality election for height h:

$$\forall X, i \,.\, X \in V_i \land (X \subseteq S \lor X \subseteq S') \Rightarrow X \subseteq S''$$

An observer who observes the blocks $S \cup S' \cup S''$ must observe a winner $p'' = w(h, S \cup S' \cup S'')$. By the constraints on α, any winners that can be determined (by P_{win}) by observing a subset of $S \cup S' \cup S''$ (in an order consistent with $\boxed{U_r}$) must also be in S'', and since only one winner exists for each height in each state, we have $p = p' = p''$.

As a result of this ordering property, correct observers with a starting belief α will always have consistent views: two observers can never disagree on the proposal at each height. When, in an observer's view, the proposal chain is filled up to some height (there is a block at every lesser height), the transactions referenced in the proposal blocks are fully committed and totally ordered. These constraints on α model the assumptions under which correct observers will never disagree on transaction ordering.

This formulation says little about the actual probabilities of voter chains diverging. However, it provides an abstraction barrier separating the statistical analysis from proofs about chain properties, which we believe is useful. For instance, the complex proofs from the original Prism paper [6] could be broken in two: statistical support for the definitions of $P(\alpha)$, and proofs that the belief α sustains the necessary chain properties.

8.4.4 Heterogeneous Paxos. We implemented a prototype of HetPax (section 7.2.3) as a Fern service. We use Charlotte blocks as messages in the consensus protocol itself, so attestations can reference messages demonstrating that consensus was achieved.

HetPax inherits Byzantine Paxos' minimum latency of 3 message delays. In our implementation, clients do not participate in the consensus: they merely request an integrity attestation from a Fern server. Including receiving a request from and sending an attestation to the client, the process has a minimum latency of 5 messages (fig. 18).

In our implementation, quorums representing trust configurations are encoded as blocks. Each HetPax blockchain includes a reference to such a block in its root, ensuring everyone agrees on the configuration. To append a block to the chain, a client requests an integrity attestation for some observer, specifying proposed block and height. To propose one block be appended to multiple chains, a client can request an integrity attestation that is the mutual subtype (section 7.3) of the

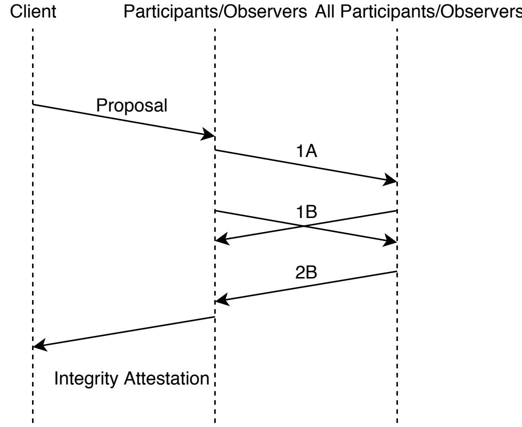

Fig. 18. Heterogeneous Paxos Message Latency

integrity attestations needed for both chains. The Fern servers then run a round of consensus in which each quorum includes one quorum of the consensus necessary for each chain.

For the purposes of demonstrating the Charlotte framework, our experiments with HetPax are *symmetric*: all observers want to agree under the same conditions. For instance, observers might trust 4 Fern servers to maintain a chain, expecting no more than one of them to be Byzantine.

8.5 DarXiv (Disaggregated arXiv)

To show the composability of Charlotte, we build DarXiv, the disaggregated version of arXiv [29], a document hosting service. DarXiv is built on a composition of services: Wilbur, Version Control, Timestamping, and a blockchain service. While the existing arXiv has a single point of failure [5], DarXiv can tolerate as many faults as the services it is composed of. We implemented DarXiv in 420 LoC based on the Charlotte framework and existing services.

8.5.1 Goals of DarXiv. Like the original arXiv, DarXiv's goal is to publicly and permanently host academic LaTeX and PDF documents with high-integrity timestamps and versioning. Formally, for an observer with initial belief α, DarXiv provides:

- **Availability**: Within each DarXiv ADS all commits c must be available:

$$c \in S \in D \ \wedge \ S \subseteq exist(U) \wedge U \in \alpha \ \Rightarrow \ c \in avail(U)$$

- **Timestamping**: For a DarXiv ADS D, all commits c must be adequately timestamped in all the valid states.

$$c \in S \in D \ \Rightarrow \exists S' \in T_\alpha . c \in S' \subseteq S$$

(Where T_α is the ADS representing adequately timestamped blocks under belief α.)

- **Versioning**: The documents in DarXiv can have different versions. These versions must be ordered. Each version is supposed to update the previous version. For a DarXiv ADS D, there must not exist universes where two states S and S' have different commits at the same version number.

$$S, S' \in D \ \wedge \ S, S' \in exist(U) \wedge U \in \alpha \Rightarrow (S \cup S') \in D$$

8.5.2 Implementation. To explain the composition, we consider DarXiv as a client that interacts with existing services: a Version Control service (section 8.2), a blockchain service (section 8.4.1)

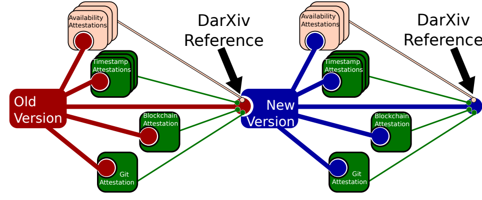

Fig. 19. Two versions (red and blue) of a document in DarXiv. Each has a collection of availability attestations (peach), as well as integrity attestations (green) from timestamping, git, and a blockchain service. Document references including these attestations provide availability, timestamps, and versions.

that provides a *total order*, one or more Wilbur services that provide availability, and one or more timestamping services (section 8.3).

To publish a new version of a document, DarXiv roughly follows these steps:

- **Bolster availability.** DarXiv interacts with the Wilbur services to establish availability of the new version by obtaining the corresponding availability attestations (peach blocks in fig. 19). The DarXiv blockchain Fern servers can be configured to check for any kind of availability threshold before committing, ensuring that any documents in DarXiv meet a global minimum availability.
- **Acquire timestamps.** DarXiv then requests timestamps from timestamp services by collecting integrity attestations.
- **Decide version.** DarXiv interacts with both the Version Control service and the blockchain service to ensure this new version is a proper "update" to the previous version and to resolve the conflict with any other simultaneous additions to the same document.
- **Commit to Blockchain.** If DarXiv successfully commits to the version-control repository, it then seeks to establish that version as the canonical next entry on the blockchain for that document. The blockchain is in essence like a multi-party Version Control Fern server: it is less likely to fail and fork the repository, because it uses agreement between many parties.
- **Generate reference** If DarXiv succeeds in collecting integrity attestations in the previous step, it generates a reference containing all the attestations for this version of document.

8.5.3 Compositional Properties. By composing the existing services, each DarXiv ADS has the flexibility to use different, unrelated Wilbur, Timestamping, or Git services. DarXiv takes advantage of the composition of the component services' properties to create the properties of arXiv as a whole. In our DarXiv implementation, each document is represented as a series of commits from some Version Control repository, signed by appropriate authors, made sufficiently available and timestamped, and totally ordered by a blockchain.

Formally, for an observer with belief α, we can express a DarXiv ADS as an intersection of ADSs: $V \Cap B \Cap T_\alpha \Cap C_\alpha$. Here, V is an ADS restricted to blocks signed by the document's authors, B a version-control branch, T_α a (belief-specific) ADS representing timestamped blocks, and C_α a (belief-specific) blockchain using HetPax. In fact, the commit blocks representing DarXiv versions will be precisely $InBoth(View(\alpha, DarXiv), B, V, T_\alpha, C_\alpha)$ (section 5.7.2). Observers need not agree on availability thresholds or trustworthiness of timestamp servers. However, attestations are monotonic (section 5.8.1), so adding more attestations never weakens what can be proven. Once an accurate observer views a document in DarXiv, it will remain forever available.

Following directly from the definitions of V and B in section 8.2, T_α in section 8.3, and C_α in section 8.4, a DarXiv ADS meets the formal properties laid out in section 8.5.1.

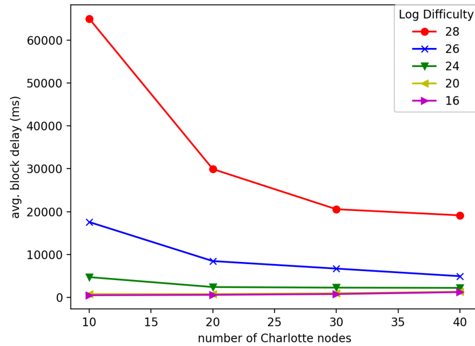

Fig. 20. Mean block delay of Nakamoto on Charlotte, with bars showing standard error. Difficulty is represented in log_2 of the number of hashes expected to mine a new block.

9 EVALUATION

To evaluate the performance of Charlotte, we ran instances of each example service. Except as specified, experiments were run on a local cluster using virtual machines with Intel E5-2690 2.9 GHz CPUs, configured as follows:

- Clients: 4 physical cores, 16 GB RAM
- Wilbur servers: 1 physical core, 8 GB RAM
- Fern servers: 1 physical core, 4 GB RAM

To emulate wide-area communication, we introduced an artificial 100 ms communication latency between VMs. Each Wilbur server stores blocks in RAM forever; to estimate latency for servers using more durable storage, one could add the marginal latency increase for the chosen storage type. However, this storage latency is typically low compared to network communication.

9.1 Blockchains

Since blockchains are an obvious application of Charlotte, we evaluate the performance, scalability, and composability of various blockchain implementations.

9.1.1 Nakamoto. To compare performance of our Nakamoto implementation to Bitcoin's, we use multiple ($n = 10, 20, 30, 40$) Charlotte nodes and measure the mean delay (across 100 consecutive blocks) until a client received an integrity attestation for a block with fixed security parameter $k = 1$. All clients and servers have one physical core, and 4 GB RAM. Figure 20 shows the results of our tests with various difficulty values (expected number of hashes to mine a block).

When difficulty is low, the delay for an integrity attestation is dominated by the communication overhead (200 ms). When, more realistically, the difficulty is high, delay is dominated by the cost of mining. Figure 20 shows that latency increases with difficulty and decreases with the inverse of the number of Charlotte servers (total computational power). Charlotte indeed scales suitably for blockchain implementations.

In fact, Bitcoin has about 2×10^{11} times the hash power [24], and 10^{14} times the difficulty that we had in our experiment, and it achieves an average block latency of 10 min. Our experiments indicate that with compute power scaled appropriately, our implementation would achieve comparable performance: about 5 minutes per block.

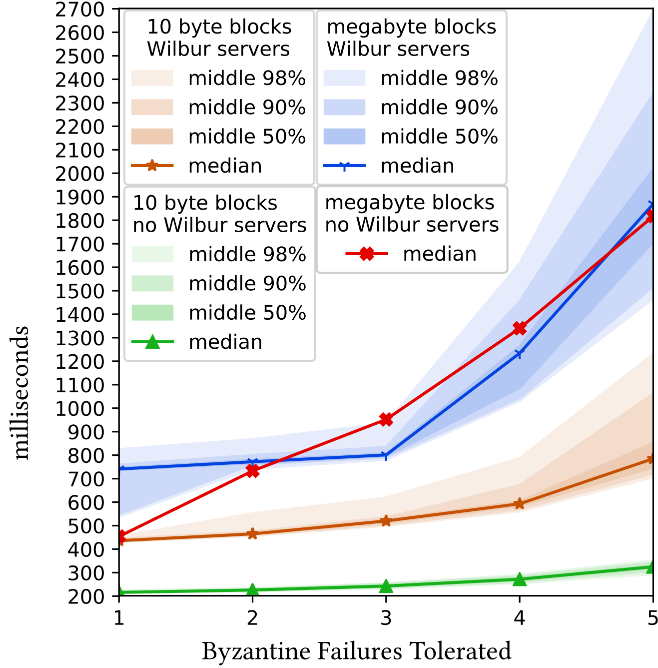

Fig. 21. Time to commit blocks in Agreement chains with various numbers of servers. The shaded zones cover the middle percentile of blocks, so the top of the lightest zone represents the 99th (slowest) percentile, and the bottom represents the 1st (fastest) percentile. The distribution for the megabyte-block, no-wilbur-server experiment is in fig. 22.

9.1.2 Agreement. To evaluate the bandwidth advantages of separating integrity and availability services, we built Agreement Chains (section 8.4.1) tolerating up to 5 Byzantine failures, both with and without Wilbur servers. To tolerate f Byzantine failures, a chain needs $3f + 1$ Fern servers, and, if it relies on Wilbur servers for availability, $f + 1$ Wilbur servers. We tested the latency and bandwidth of our chains, with some experiments using 10 byte blocks, and some using 1 MB blocks. In each experiment, a single client appends 1000 blocks to a chain, with the first 500 excluded from measurements to avoid warm-up effects. Each experiment ran three times.

In the simple case, without Wilbur servers, all Fern servers receive all blocks, similarly to the traditional blockchain strategy [69]. The theoretical minimum latency is 2 round trips from the client to the Fern servers, or 200 ms.

We also built chains that separate the Fern servers' integrity duties from Wilbur servers' availability duties. In these chains, Fern servers would not attest to any reference unless it included $f + 1$ different Wilbur servers' availability attestations. For these chains, the client sends blocks to Wilbur servers, waits to receive availability attestations, and then sends references to Fern servers. These two consecutive round trips incur a minimum latency of 400 ms.

Latency. Figure 21 and fig. 22 show the median latency to commit a block for each of our Agreement chain experiments (section 9.1.2). Theoretical minimum latency is 4 message sends (round trip from the client to the Wilbur servers, and then from the client to the Fern servers), or 400 ms. For chains with small blocks, latency remains close to the 200 ms and 400 ms minimums.

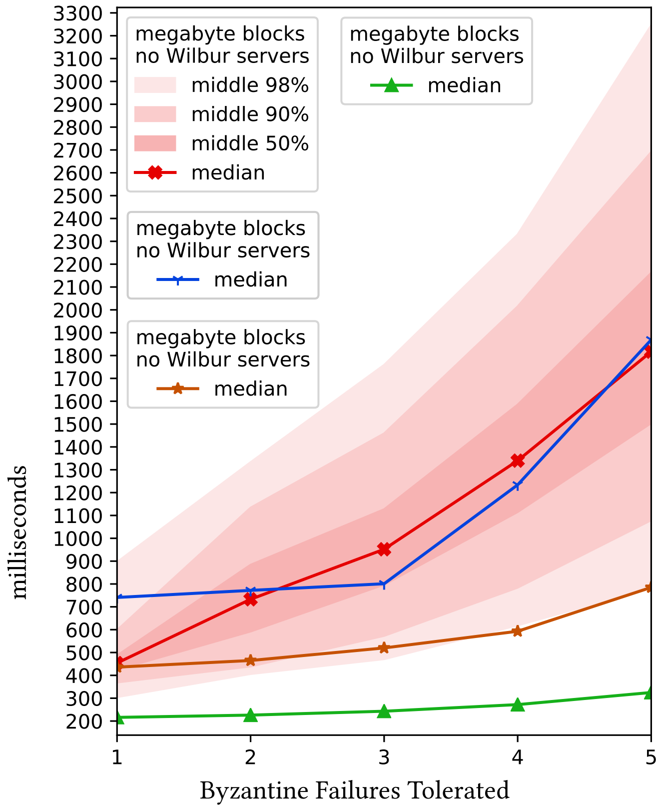

Fig. 22. Time to commit blocks in Agreement chains with various numbers of servers. The distribution for the megabyte-block, no-wilbur-server experiment is shown.

For chains with 1 megabyte blocks, experimental setup has significant slowdowns, likely due to bandwidth limitations.

Bandwidth. Separating availability and integrity concerns (section 7.1) has clear benefits in terms of bandwidth. Because it sends large blocks to just $f + 1$ Wilbur servers instead of $3f + 1$ Fern servers, our client uses much less bandwidth in the large-block experiments with Wilbur servers than without them (fig. 23). In theory, committing a block with Wilbur servers requires bandwidth for $f + 1$ blocks, and without Wilbur servers requires $3f + 1$ blocks. The overhead inherent in the additional communication with Wilbur servers and the attestations issued is small compared to the savings, but visible in fig. 23 for larger numbers of Byzantine failures tolerated.

9.1.3 Heterogeneous Paxos. To evaluate the feasibility of consensus-based blockchains and multi-chain blocks in Charlotte, we built several chains with HetPax (section 7.2.3), and ran 5 types of experiments. With our artificial network latency, the theoretical lower bound on consensus latency is 500 ms, and maximum throughput per chain is 2 blocks/second. Each experiment recorded the latency clients experience in appending their own blocks to the chain, as well as system-wide throughput. All HetPax experiments used single-core VMs with 8 GB RAM, except as noted.

Single Chain. In these experiments, a client appends 2000 successive blocks to one chain. Mean latency is 527 ms for a chain with 4 Fern servers and 538 ms for 7 Fern servers. Since the best

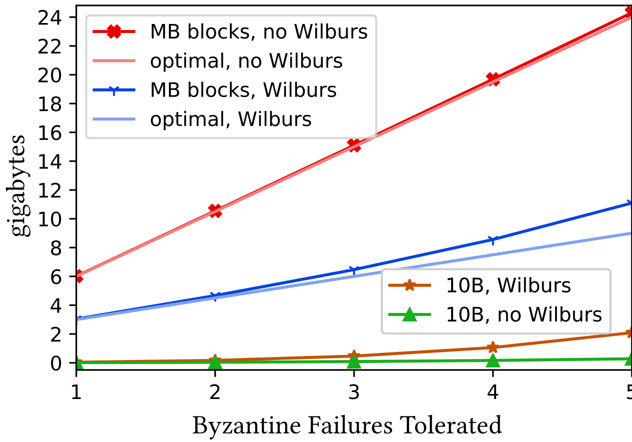

Fig. 23. Total bandwidth used by a client appending 1500 blocks to Agreement-based chains. With 10 B blocks and higher numbers of Byzantine failures tolerated, we can see the overhead inherent in the additional communication with a larger number of servers when using separate Wilbur and Fern servers. For 1 MB blocks, this overhead is much smaller than the bandwidth saved.

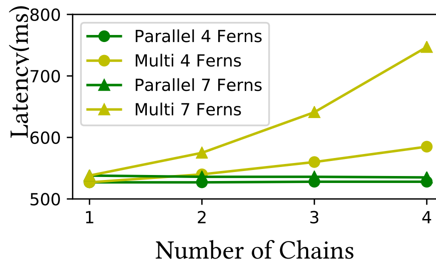

Fig. 24. HetPax Multichain and Parallel experiments. In Parallel experiments, each chain operates independently (and has its own client). In Multichain experiments, one client tries to append all blocks to all chains. Optimal latency is 500 ms.

possible latency is 500 ms, these results are promising. Overheads include cryptographic signatures, verification, and garbage collection.

Parallel. As the darker green lines in fig. 24 show, independent HetPax chains have independent performance. In these experiments, we simultaneously ran 1–4 independent chains, each with 4 or 7 Fern servers. In each experiment, a client appends 2000 successive blocks to one chain. There is no noticeable latency difference between a single chain and many chains running together. Throughput scales with the number of chains (and inversely to latency). This scalability is the fundamental advantage of a blockweb over forcing everything onto one central blockchain.

Multichain shared blocks. Shared (joint) blocks facilitate inter-chain interaction (section 7.3). In these experiments, a single client appends 1000 shared blocks to 2–4 chains, each with 4 or 7 Fern servers. As the yellow lines in fig. 24 show, latency scales roughly linearly with the number of chains.

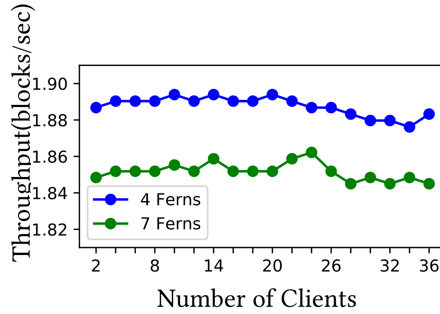

Fig. 25. Throughput of HetPax under contention. 2–36 clients try to append 2000 blocks to just one chain. Optimal throughput is 2 blocks/sec.

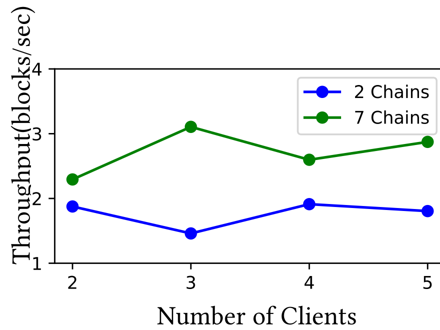

Fig. 26. Throughput of HetPax mixed-workload experiment (4 Fern servers).

Contention. In these experiments, all clients simultaneously contend to append 2000 unique blocks to the same chain. We measured the blocks that were actually accepted into slots 500–1500 of the chain. We used 2–36 clients, and chains with 4 or 7 Fern servers, configured with 2 GB RAM. Like Byzantized Paxos [54], HetPax can slow under contention and occasionally requires a dynamic timeout to automatically trigger a new round. Chain throughput is shown in fig. 25. Our chains, on average, achieved 1.88 blocks/sec throughput for 4 Fern servers and 1.85 blocks/sec for 7, not far from the 2 blocks/sec optimum. Throughput does not decrease much with the number of clients.

Mixed. These experiments attempt to simulate a more realistic scenario by including all 3 types of workload. Each experiment has 2–5 clients, and either 2 or 7 chains, each with 4 Fern servers. On each block, a client tries to append a shared block to two random chains with probability 10% and otherwise tries appending to a random single chain. The results are in fig. 26. Throughput can be over 2.0 blocks/sec because multiple clients can append blocks to different chains in parallel. Mean throughput is 1.8 blocks/sec and 2.7 blocks/sec for 2 and 7 chains respectively, which is expected because the 2-chain configuration has more contention.

HetPax scales well horizontally with multiple chains running in parallel. Furthermore, throughput does not decrease much with more clients involved. This gives us ability to make progress even with lots of clients connecting to the same chain concurrently.

However, the number of Fern servers does play a major role in determining latency. With a small group of Fern servers, HetPax can almost reach 500 ms, which is optimal for our network. Although latency increases linearly with the number of Fern servers, for some services, it is possible

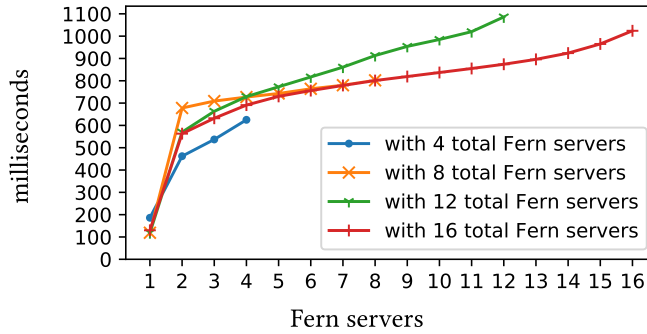

Fig. 27. Mean time for a block to be timestamped by x Fern servers, in experiments featuring 4, 8, 12, and 16 total Fern servers.

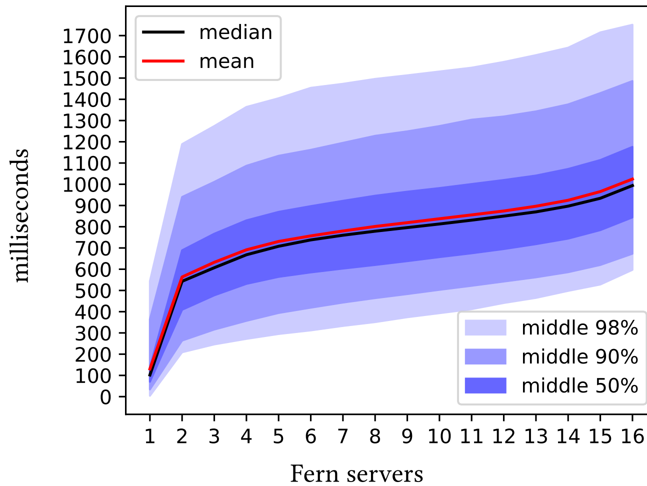

Fig. 28. Time for a block to be timestamped by x Fern servers, in an experiment featuring a total of 16 Fern servers. Shaded zones cover the middle percentile of blocks, so the top of the lightest zone represents the 99th (slowest) percentile, and the bottom represents the 1st (fastest) percentile. Also shown are mean and median block times (very similar).

to break down big shared blocks into a set of small shared blocks, each of which may require fewer servers to commit. For example, in section 2.1.1, we discuss how to break a k-chain transaction into a $\log k$-depth graph whose nodes are small two-chain transactions. By following the same strategy, the latency would be reduced to $t \times \log k$, where t is the average latency for completing a 2-chain block. Since our HetPax implementation is just a prototype, we believe that with further optimization effort, average latency performance will improve.

9.2 Timestamping

To evaluate performance, composability, and entanglement (section 6.5) with a non-blockchain service, we ran experiments with varying numbers of Timestamping Fern servers (section 8.3). All client and server VMs had 4 GB RAM. For each experiment, a single client requested timestamps

for a total of 100,000 blocks. For each block, it requested a timestamp from one server, rotating through the Fern servers.

For each 100 timestamps a Fern server issued, it would create a new block referencing those 100 timestamps, and request that all other Fern servers timestamp this block. Since timestamps are transitive (if c is a timestamp referencing b, and b references a, then c also timestamps a), every block was soon timestamped by all Fern servers.

Our results, in fig. 27 and fig. 28, show that blocks are timestamped by one server at nearly network speed, and then (after a delay while a batch is produced) extremely quickly timestamped by many other timestamping Fern servers.

We also calculated the time it took blocks to accrue different Fern servers' timestamps. As fig. 27 shows, the Fern servers quickly timestamp each request. Blocks get 1 timestamp very close to the 100 ms network latency minimum. There is a delay between 1 and 2 timestamps because it takes a little while for the Fern servers to collect 100 timestamps and to create their own block. After that, blocks accrue timestamps very quickly, since each Fern Server requests timestamps from all other Fern servers. These experiments suggest that entanglement (section 6.5) can be a fast, efficient, and compositional way to lend integrity to large ADSs.

Not all blocks took exactly the same amount of time to accrue the same number of timestamps. fig. 28 shows the distribution of times for blocks in the experiment with 16 Fern servers. The scale is the same as in fig. 27. In general, each data point (time for blocks to accrue x timestamps in an experiment with n Fern servers) was approximately Poisson distributed.

9.3 Composition

9.3.1 Timestamping and Blockchains. To explore Charlotte's composability, we also composed our (1- or 2-failure-tolerant) Agreement chains with our Timestamping Fern servers. We saw no statistically significant change in chain performance: the overhead of timestamping was unmeasurably small. Each block was timestamped quickly by directly requested Timestamping servers, but entanglement (section 9.2) was limited by the chain rate.

9.3.2 DarXiv. DarXiv (section 8.5) illustrates the easy composition of Charlotte services, since only 420 lines of code needed to be written above the reused services. For evaluating its performance, the DarXiv Blockchain service accepts commits from a Version Control service carrying at least one attestation from Wilbur and Timestamping services each. DarXiv document creation takes an average of 32 ms per document, and update and fetch operations take 40 ms on average. This performance is easily sufficient to handle the load on the actual arXiv system, which currently handles about 640 new submissions, 700 update requests, and a million downloads per day [4, 35].

DarXiv Fern servers check for a global minimum availability threshold before committing a document. They could be configured to require, for example, attestations from arbitrary quorums of Wilbur servers using arbitrary availability mechanisms. Submitting the document concurrently to multiple redundant Wilbur servers does significantly increase latency, although it does of course require more total bandwidth.

10 RELATED WORK

10.1 Addressing by Hash

Many other distributed systems reference content by hash and form ADSs. Authenticated data structures [66, 70] such as Merkle Trees [64] often reference data by hash without any additional attestations, just to ensure the client can check that data fetched is correct. Most hash-based reference systems, however, only work within a specific service. For instance, Git uses hashes to reference and request commits stored on a server [91]. Git-lfs can track and request large files

on separate servers with hash-based identifiers [30]. Similarly, PKI systems reference keys and certificates by hash, and maintain groups of availability servers [16, 37, 37, 39, 40, 52, 79]. Distributed Hash Tables, such as CFS [21] ultimately maintain Availability servers, and ensure integrity by referencing data via Hash.

HTML pages can reference resources using the `integrity` field [94] to specify a hash, and the `src` field to specify a server, like an availability attestation without formal guarantees. Likewise, BitTorrent's Torrent files [18] and Magnet URIs [19] reference a file by hashes of various kinds, and can specify "acceptable sources" from which to download the file. Charlotte's references aim to be extensible in terms of the hash algorithms used, and generic over all types of data. Uniquely, Charlotte bundles references to data with references to attestations, which can offer precise formal guarantees.

In concurrent work, Protocol Labs' IPLD [46] is a multi-protocol format for addressing arbitrary content by hash. Like Charlotte's `AnyWithReference` (section 4), Multiformats [67] offers an extensible format for self-describing data including protobufs [73]. Both IPLD and Multiformats are developed closely with IPFS [10], a peer-to-peer file distribution system. Future work might fruitfully combine these technologies with Charlotte-style attestations.

10.2 BlockDAGs

Other projects have explored DAGs of blocks in a blockchain context. Many, such as Iota [72], Nano (also known as RaiBlocks) [55], Avalanche [75], Spectre [84], Phantom, and Ghostdag [85] are tailored to cryptocurrency. Wang, Yu, Chen, and Xiang surveyed a variety of such projects in 2023 [93]. Each defines its own currency, and they do not compose.

Some projects, such as æternity [36], alephium [92], Qubic [45], and Plasma [71] enable general-purpose computation on a BlockDAG by way of smart contracts. However, they ultimately rely on a single global consensus mechanism for the integrity of every service. Concurrent work has also explored using DAGs of blocks as part of mempools and consensus protocols to more efficiently create serializable transaction orderings for blockchains [7, 22, 50, 87]. These projects describe a single chain, rather than an open system of interconnected structures.

Sharded blockchains, including Omniledger [51], Elastico [59], RapidChain [96], RSCoin [23], and Ethereum 2.0 [14] are a form of BlockDAG. Most still require that all services have essentially the same trust assumptions.

Other sharded blockchain projects, such as Aion [88], Cosmos [27], and Polkadot [95], envision heterogeneous chains with inter-chain communication. Polkadot features a single Relay Chain trusted by all parachains (parallelizable chains), although it does allow parachains to proxy for outside entities, including other blockchains. Most similarly to our multi-chain transactions (section 7.3), Aion uses Bridges, consensus mechanisms trusted by multiple chains, to commit a transaction to each. All of these blockchain projects operate at a higher level of abstraction than Charlotte, but might benefit from implementing on top of Charlotte. For example, whereas Cosmos' Inter-Blockchain Communication [27] and Aion's Transwarp Conduits [38] require chains to read and validate each other's commits, we offer a unified framework for such protocols: integrity attestations.

10.3 Availability attestations

Although storage services are widely available [2, 31, 65], availability attestations make Wilbur servers unique. However, there is a great deal of work on reliable storage [25, 28] and proofs of retrievability [13, 49, 82] that could be used to make useful availability attestations.

10.4 Integrity attestations

In some ways, attestations resemble the labels of distributed information flow control systems [57, 98], and implement a kind of endorsement [97] as additional attestations are minted for the same block. In other ways, integrity attestations generalize ordering services for traditional distributed systems [41] or blockchains [86]. These services maintain a specific property of a ADS (ordering), much like our blockchain integrity attestations. However, integrity attestations generalize over many possible properties: timestamps, provenance, etc.

Future integrity attestation subtypes might take advantage of technologies like authentication-logic proofs or other artifacts representing assurances of data provenance [3, 83].

10.5 Distributed Data Structures

Prior work such as Tango [9] successfully built distributed data structures based on a shared log abstraction [8]. The shared log records updates to the data structures and allows clients to build a view of the data structure by playing back these operations. While this design has some similarities to Charlotte, Tango is not a fully decentralized system; it does not consider data structure integrity, it relies on a centralized sequencer, and it requires a totally ordered log.

The Fabric system [56] maintains persistent, distributed, decentralized data structures whose use is governed by explicit integrity and confidentiality labels. These labels are used to mediate access to the data structures by different clients, with both static and dynamic information flow control used to prevent client code from violating these policies. Fabric references are not hash-based, so they do not have the built-in integrity properties that derive from that mechanism. Fabric does not explicitly enforce availability properties, and it does not provide a semantic framework for interpreting labels.

11 CONCLUSION

Charlotte offers a decentralized framework for composable Attested Data Structures with well-defined availability and integrity properties. Together, these structures form the blockweb, a novel generalization of blockchains. Charlotte addresses many of the shortcomings of existing blockchain systems by enabling parallelism and composability. Charlotte is flexible enough to enable services patterned after existing ADS while offering rigorous guarantees through attestations that can be given precise semantics.

With Charlotte, heterogeneous observers can use heterogeneous services across heterogeneous participants with well-defined failure tolerance. By embracing Least Ordering and heterogeneity, new services will save time and resources, and serve broader audiences. Charlotte is available as open source [80].

Charlotte provides a basic abstraction layer for decentralized distributed computing, and future research could explore building higher-level services on top of Charlotte. Charlotte makes new kinds of inoperable applications feasible. For example, the DarXiv application could be expanded into a general service for tracking trustworthy versioned documents—in the age of generative AI, we may need systems like DarXiv that track provenance of content. We have explored building various forms of agreement and consensus on top of Charlotte, exploiting entanglement, and Charlotte provides a framework in which an even greater variety of consensus mechanisms can be explored. For generality, Charlotte does not prescribe any logic for expressing attestations, but it would be useful to develop an explicit, reusable logical framework to aid developers in successfully composing services that use it. We hope for a future of interoperable ADSs created with Charlotte.

ACKNOWLEDGMENTS

We thank Mae Milano for useful feedback on the paper, Joe Boyt for suggesting the DarXiv application, Mark Moir for discussions about entanglement, and Joe Halpern and Paul Ginsparg for data on arXiv usage. We also thank Ripple for its support and the National Science Foundation for support via grants 1717554 and 1704615.

REFERENCES

[1] Artifacts: Researcher recognition. accelerated. https://artifacts.ai. Accessed September 2021.

[2] Amazon. Cloud storage with AWS. https://aws.amazon.com/products/storage.

[3] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[4] arXiv. Monthly submission rates. https://arxiv.org/stats/monthly_submissions.

[5] arXiv. Twitter: arxiv.org. https://twitter.com/arxiv/status/1192467597054361606.

[6] Vivek Bagaria, Sreeram Kannan, David Tse, Giulia Fanti, and Pramod Viswanath. Prism: Deconstructing the blockchain to approach physical limits. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 585–602, New York, NY, USA, 2019. Association for Computing Machinery.

[7] Leemon Baird and Atul Luykx. The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers. In *2020 International Conference on Omni-layer Intelligent Systems (COINS)*, pages 1–7, 2020.

[8] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *9th USENIX Symp. on Networked Systems Design and Implementation (NSDI 12)*, pages 1–14, San Jose, CA, 2012. USENIX.

[9] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 24th ACM Symp. on Operating System Principles (SOSP), 2013.

[10] Juan Benet. IPFS – content addressed, versioned, P2P file system. https://ipfs.io.

[11] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.

[12] Tamas Blummer, Sean Bohan, Mic Bowman, Christian Cachin, Nick Gaski, Nathan George, Gordon Graham, Daniel Hardman, Ram Jagadeesan, Travin Keith, Renat Khasanshyn, Murali Krishna, Tracy Kuhrt, Arnaud Le Hors, Jonathan Levi, Stanislav Liberman, Esther Mendez, Dan Middleton, Hart Montgomery, Dan O'Prey, Drummond Reed, Stefan Teis, Dave Voell, Greg Wallace, and Baohua Yang. An introduction to Hyperledger. Technical report, Hyperledger, 2018.

[13] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: Theory and implementation. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*, CCSW '09, pages 43–54, New York, NY, USA, 2009. ACM.

[14] Vitalik Buterin, Danny Ryan, Hsiao-Wei Wang, Terence Tsao, and Chih Cheng Liang. Ethereum 2.0 phase 1 – shard data chains. https://https://github.com/ethereum/eth2.0-specs/blob/master/specs/core/1_shard-data-chains.md.

[15] Christian Cachin and Marko Vukolić. Blockchain Consensus Protocols in the Wild. *ArXiv e-prints*, July 2017.

[16] Jon Callas, Lutz Donnerhacke, Hal Finney, David Shaw, and Rodney Thayer. OpenPGP message format. RFC 4880, RFC Editor, November 2007. http://www.rfc-editor.org/rfc/rfc4880.txt.

[17] Scott Chacon and Ben Straub. 7.11 Git tools—submodules. In *Pro Git*, pages 299––318. Apress, 2 edition, 2014.

[18] Bram Cohen. The BitTorrent protocol specification. http://bittorrent.org/beps/bep_0003.html, 2017.

[19] Wikipedia contributors. Magnet URI scheme — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Magnet%20URI%20scheme&oldid=888151611, 2019. [Online; accessed 2019-03-25].

[20] Kyle Croman, Christian Decker, Ittay Eyal, Adem Efe Gencer, Ari Juels, Ahmed Kosba, Andrew Miller, Prateek Saxena, Elaine Shi, Emin Gün Sirer, Dawn Song, and Roger Wattenhofer. On scaling decentralized blockchains. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[21] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symp. on Operating System Principles (SOSP)*, pages 202–215, 2001.

[22] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: A dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 34–50, New York, NY, USA, 2022. Association for Computing Machinery.

[23] George Danezis and Sarah Meiklejohn. Centrally banked cryptocurrencies. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*, 2016.

[24] Alex de Vries. Bitcoin energy consumption index. Technical report, Digiconomist, 2018.

[25] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99:476–489, 2011.

[26] Ethereum Foundation. Ethereum white paper. Technical report, Ethereum Foundation, 2018.

[27] Ethan Frey and Christopher Goes. Cosmos Inter-Blockchain Communication (IBC) Protocol. https://cosmos.network, October 2018.

[28] Juan Garay, Rosario Gennaro, Charanjit Jutla, and Tai Rabin. Secure distributed storage and retrieval. In *11th Workshop on Distributed Algorithms (WDAG)*, Saarbrücken, Germany, September 1997.

[29] Paul Ginsparg, Oya Y. Rieger, Steinn Sigurdsson, Martin Lessmeister, Erick Peirson, Jim Entwood, and Janelle Morano. ArXiv. Accessed: 2019-11-13.

[30] Git large file storage. https://git-lfs.github.com, 2018.

[31] Google. Google cloud storage. https://cloud.google.com/storage.

[32] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, and Daniel Smith. *The Java Language Specification*. Oracle America, se 11 edition, Aug 2018.

[33] gRPC: A high performance, open-source universal RPC framework. https://grpc.io, 2018.

[34] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983.

[35] Joe Halpern and Paul Ginsparg. Personal correspondence.

[36] Sascha Hanse, Luca Favetella, Hans Svensson, Emin Mahrt, Tobias Lindahl, and Erik Stenman. æternity protocol. https://aeternity.com, January 2019.

[37] Andre Heinecke. How to setup an OpenLDAP-based PGP keyserver. https://wiki.gnupg.org/LDAPKeyserver, Dec 2017.

[38] Shidokht Hejazi-Sepehr, Ross Kitsis, and Ali Sharif. Transwarp-Conduit: Interoperable blockchain application framework. https://aion.network/developers, January 2019.

[39] Marc Horowitz. A PGP public key server. Technical report, MIT, Nov 1996.

[40] Russell Housley, Tim Polk, Warwick Ford, and David Solo. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. Internet RFC-3280, April 2002.

[41] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free coordination for Internet-scale systems. In *USENIX ATC*, 2010.

[42] IBM. Blockchain for supply chain. `https://www.ibm.com/blockchain/supply-chain/`, 2018.

[43] International Blockchain Real Estate Association (IBREA). https://ibrea.info, 2021.

[44] IEEE. *The Open Group Base Specifications Issue 7*. IEEE and The Open Group, 2018 edition, 2018.

[45] IOTA. Qubic: Quorum based computations powered by IOTA. https://qubic.iota.org.

[46] IPLD. https://ipld.io.

[47] Krzysztof Janowicz, Blake Regalia, Pascal Hitzler, Gengchen Mai, Stephanie Delbecque, Maarten Fröhlich, Patrick Martinent, and Trevor Lazarus. On the prospects of blockchain and distributed ledger technologies for open science and academic publishing. *Semantic Web*, 9(5):545–555, January 2018.

[48] JPMorgan. J.P. Morgan creates digital coin for payments. https://www.jpmorgan.com/global/news/digital-coin-payments. Accessed: 2019-11-13.

[49] Ari Juels and Burton S. Kaliski Jr. PORs: Proofs of retrievability for large files. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 584–597, New York, NY, USA, 2007. ACM.

[50] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, New York, NY, USA, 2021. Association for Computing Machinery.

[51] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. *2018 IEEE Symposium on Security and Privacy (Oakland)*, pages 583–598, 2018.

[52] Bogdan Kulynych, Wouter Lueks, Marios Isaakidis, George Danezis, and Carmela Troncoso. ClaimChain: Improving the security and privacy of in-band key distribution for messaging. In *Proceedings of the 2018 Workshop on Privacy in the Electronic Society*, WPES'18, pages 86–103, New York, NY, USA, 2018. ACM.

[53] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558–565, July 1978.

[54] Leslie Lamport. Byzantizing Paxos by refinement. In *25th Int'l Conf. on Distributed Computing (DISC)*, pages 211–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[55] Colin LeMahieu. Nano: A feeless distributed cryptocurrency network. Technical report, Nano, 2018.

[56] Jed Liu, Owen Arden, Michael D. George, and Andrew C. Myers. Fabric: Building open distributed systems securely by construction. *J. Computer Security*, 25(4–5):319–321, May 2017.

[57] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. Fabric: A platform for secure distributed computation and storage. In *22nd ACM Symp. on Operating System Principles (SOSP)*, pages 321–334, October 2009.
[58] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas P. E. Barry, Eli Gafni, Jonathan Jové, Rafał Malinowsky, and J. Murphy McCaleb. Fast and secure global payments with Stellar. In *27th ACM Symp. on Operating System Principles (SOSP)*, 2019.
[59] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 17–30, New York, NY, USA, 2016. ACM.
[60] Chip Martel, Glen Nuckolls, Michael Gertz, Prem Devanbu, April Kwong, and Stuart G. Stubblebine. A general model for authentic data publication. Technical Report CSE-2001-9, UC Davis Dept. of Computer Science, December 2001.
[61] Manuel Martin, Antonio Romero, and Roberto Rabasco. Orvium: Accelerating scientific publishing. https://docs.orvium.io/Orvium-WP.pdf, 2019. Accessed September 2021.
[62] David Mazières. The Stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org, April 2015.
[63] Trent McConaghy, Rodolphe Marques, and Andreas Müller. BigchainDB: a scalable blockchain database. https://www.bigchaindb.com/whitepaper, 2016.
[64] Ralph C. Merkle. Protocols for public key cryptosystems. In *IEEE Symp. on Security and Privacy*, page 122, Los Alamitos, CA, USA, 1980. IEEE Computer Society.
[65] Microsoft. Azure storage. https://azure.microsoft.com/services/storage.
[66] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 411–423, New York, NY, USA, 2014. Association for Computing Machinery.
[67] Multiformats. https://multiformats.io.
[68] Byron Murphy. The history of 51% attacks and the implications for Bitcoin. Accessed: 2019-11-13.
[69] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.
[70] Moni Naor and Kobbi Nissim. Certificate revocation and certificate update. In *7th USENIX Security Symposium (USENIX Security 98)*, San Antonio, TX, January 1998. USENIX Association.
[71] Joseph Poon and Vitalik Buterin. Plasma: Scalable autonomous smart contracts. https://plasma.io, August 2017.
[72] Serguei Popov. The Tangle. Technical report, Iota, 2018.
[73] Protocol buffers. https://developers.google.com/protocol-buffers/, 2018.
[74] Yoav Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource managers using atomic commitment. In *18th Very Large Data Bases Conference (VLDB)*, August 1992.
[75] Team Rocket. Snowflake to Avalanche: A novel metastable consensus protocol family for cryptocurrencies. https://ipfs.io/ipfs/QmUy4jh5mGNZvLkjies1RWM4YuvJh5o2FYopNPVYwrRVGV, May 2018.
[76] Andreas Schaufelbühl, Sina Rafati Niya, Lucas Pelloni, Severin Wullschleger, Thomas Bocek, Lawrence Rajendran, and Burkhard Stiller. Eureka – a minimal operational prototype of a blockchain-based rating and publishing system. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 13–14, 2019.
[77] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.
[78] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011.
[79] David Shaw. The OpenPGP HTTP Keyserver Protocol (HKP). Internet-Draft draft-shaw-openpgp-hkp-00.txt, IETF Secretariat, March 2003.
[80] Isaac Sheff, Xinwen Wang, Kushal Babel, Haobin Ni, Robbert van Renesse, and Andrew C. Myers. Charlotte-public. https://github.com/isheff/charlotte-public, 2021. software release.
[81] Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous Paxos. In *OPODIS*, December 2020.
[82] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 325–336, New York, NY, USA, 2013. ACM.
[83] Emin Gün Sirer, Willem de Bruijn, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical attestation: An authorization architecture for trustworthy computing. In *11th ACM Symp. on Operating System Principles (SOSP)*, 2011.
[84] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016.

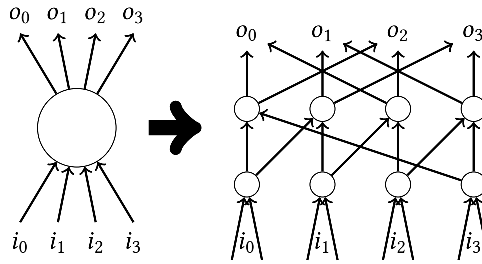

Fig. 29. Converting 4 inputs and 4 outputs to a graph of 2-account transactions.

[85] Yonatan Sompolinsky and Aviv Zohar. Phantom: A scalable blockdag protocol. Cryptology ePrint Archive, Report 2018/104, 2018.

[86] João Sousa, Alysson Bessani, and Marko Vukolic. A Byzantine fault-tolerant ordering service for the Hyperledger Fabric blockchain platform. In *48th Annual IEEE/IFIP Int'l Conf. on Dependable Systems and Networks (DSN)*, pages 51–58, June 2018.

[87] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 2705–2718, New York, NY, USA, 2022. Association for Computing Machinery.

[88] Matthew Spoke and Nuco Engineering Team. Aion: Enabling the decentralized internet. https://aion.network, July 2017.

[89] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, August 2001.

[90] CryptoKitties Team. Cryptokitties: Collectible and breedable cats empowered by blockchain technology: White pa-purr, version 2.0. Technical report, 2018.

[91] L. Torvalds and J. Hamano. Git distributed version control system. https://git-scm.com, 2010.

[92] Cheng Wang. Alephium: a scalable cryptocurrency system based on blockflow. https://alephium.org.

[93] Qin Wang, Jiangshan Yu, Shiping Chen, and Yang Xiang. SoK: DAG-based blockchain systems. *ACM Comput. Surv.*, 55(12), mar 2023.

[94] Joel Weinberger, Francois Marier, Devdatta Akhawe, and Frederik Braun. Subresource integrity. W3C recommendation, W3C, June 2016. http://www.w3.org/TR/2016/REC-SRI-20160623/.

[95] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework. https://polkadot.network.

[96] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. RapidChain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 931–948, New York, NY, USA, 2018. ACM.

[97] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Secure program partitioning. *ACM Trans. on Computer Systems*, 20(3):283–328, August 2002.

[98] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *5th USENIX Symp. on Networked Systems Design and Implementation (NSDI)*, pages 293–308, 2008.

A BITCOIN TRANSACTIONS IN TWO ACCOUNTS OR FEWER

In Bitcoin, it is advantageous to combine many small transfers of money into big ones, with many inputs and many outputs. This improves anonymity and performance. In the real financial system of the USA, however, all monetary transfers are from one account to another. They are all exactly two chain transactions.

We can simulate this limitation by refactoring each Bitcoin UTXO as 2 UTXOs, and each Bitcoin transaction as a DAG of transactions with depth:

$$\left\lceil \log_2 \left(\max \left(\text{number of inputs}, \ \text{number of outputs}\right)\right)\right\rceil$$

To do this, we create

$$n \triangleq 2^d$$

chains, each of which is

$$d \triangleq \lceil \log_2 (\max (\text{number of inputs}, \ \text{number of outputs})) \rceil$$

long. We call these chains C^0 through C^n. Original input UTXO i corresponds to both inputs to the first transaction of chain i. Original output UTXO j corresponds to one output of each of the last transactions from chains j and $\left(j + 2^{d-1}\right) \text{mod } n$. For $0 \leq k < (d-1)$, the outputs of the kth transaction in chain i, called C^i_k, go to C^i_{k+1}, and:

$$C_{k+1}^{\left(i+2^j\right) \mod n}$$

The outputs of C^i_d go to the UTXOs corresponding with output i, and output $\left(i + 2^{d-1}\right) \text{mod } n$. Each transaction divides its output values proportionately to the sums of the final output values reachable from each of the transaction's outputs. fig. 29 is an example transformation from a 4-input, 4-output transaction to a DAG of depth 2 using all 2-input, 2-output transactions.

B IMPLEMENTATION LIMITATIONS OF ATTESTATIONS

Since programmers can define their own subtypes of integrity or availability attestations, nothing prevents them from encoding availability guarantees in an integrity attestation, or violating the availability attestation monotonicity requirement (section 5.8.1). Programmers who violate the system assumptions naturally lose guarantees.

In our implementation, the only operational distinction between an availability attestation and an integrity attestation is in the `Reference` object. When one block references another, it can also reference relevant integrity and availability attestations. However, whereas an included reference to an integrity attestation is itself a `Reference` object, an included reference to an availability attestation carries only a `Hash`. This is because an integrity attestation might need an availability attestations to describe where to obtain the integrity attestation. However, the same is not true of an availability attestation: it is pointless to send availability attestation b just to describe where to fetch availability attestation a, since it is just as easy to send availability attestation a in the first place.

C PROOF FOR THEOREM 3

Proof. First, we introduce a helpful function:

$$M(D) \triangleq \left\{ \bigcup W \;\middle|\; W \in \mathcal{P}(D) \right\}$$

Note that $D \subseteq M(D)$. We can think of $M(D)$ as an ADS whose states include all possible combinations of (possibly conflicting) states of D. As a side note, we call D *monotonic*, meaning no pair of states conflict (they just form a bigger state), precisely when $D = M(D)$.

We can now re-state our definition of intersection:

$$D \Cap D' = \left\{ S \cup S' \;\middle|\; \begin{array}{ll} & S \in D \wedge S' \in M(D') \\ \vee & S \in M(D) \wedge S' \in D \end{array} \right\}$$

By the definition of *view* (section 5.6) and $\Cap$:

$$View(\alpha, D \Cap D') = \bigcup \left\{ S \cup S' \;\middle|\; \begin{array}{ll} & \forall U \in \alpha, W \in D, W' \in M(D') \,.\, W \cup S \cup S' \cup W' \in D \Cap D' \ \vee\ (W \cup W') \not\subseteq exist(U) \\ \wedge & \forall U \in \alpha, W \in M(D), W' \in D' \,.\, W \cup S \cup S' \cup W' \in D \Cap D' \ \vee\ (W \cup W') \not\subseteq exist(U) \\ \wedge & \forall U \in \alpha \,.\, S \subseteq avail(U) \\ \wedge & \forall U \in \alpha \,.\, S' \subseteq avail(U) \\ \wedge & (S \in D \wedge S' \in M(D')) \ \vee\ (S \in M(D) \wedge S' \in D') \end{array} \right\}$$

For *InBoth*, we consider only full states $S \in D$ and $S' \in D'$:

$$InBothView(\alpha, D \uplus D')DD' =$$
$$\bigcup \left\{ S \cap S' \;\middle|\; \begin{array}{ll} & \forall U \in \alpha, W \in D, W' \in M(D') \,.\, W \cup S \cup S' \cup W' \in D \uplus D' \;\vee\; (W \cup W') \nsubseteq exist(U) \\ \wedge & \forall U \in \alpha, W \in M(D), W' \in D' \,.\, W \cup S \cup S' \cup W' \in D \uplus D' \;\vee\; (W \cup W') \nsubseteq exist(U) \\ \wedge & \forall U \in \alpha \,.\, S \subseteq avail(U) \\ \wedge & \forall U \in \alpha \,.\, S' \subseteq avail(U) \\ \wedge & S \in D \\ \wedge & S' \in D' \end{array} \right\}$$

By the definition of M, it is always the case that:

$$S \in M(D) \wedge W \in M(D) \Rightarrow (S \cup W) \in M(D)$$

Recall that $S \in D \Rightarrow S \in M(D)$. If follows that:

$$S \in M(D) \wedge W \in M(D) \Rightarrow (S \cup W) \in M(D)$$

Therefore,

$$\forall S' \in D', W' \in M(D') \,.\, (S' \cup W') \in M(D')$$

By the definition of $\uplus$,

$$InBothView(\alpha, D \uplus D')DD' =$$
$$\bigcup \left\{ S \cap S' \;\middle|\; \begin{array}{ll} & \forall U \in \alpha, W \in D \,.\, W \cup S \in D \;\vee\; W \nsubseteq exist(U) \\ \wedge & \forall U \in \alpha, W' \in D' \,.\, W' \cup S' \in D' \;\vee\; W' \nsubseteq exist(U) \\ \wedge & \forall U \in \alpha \,.\, S \subseteq avail(U) \\ \wedge & \forall U \in \alpha \,.\, S' \subseteq avail(U) \\ \wedge & S \in D \\ \wedge & S' \in D' \end{array} \right\}$$

By the definition of *view*,

$$InBothView(\alpha, D \uplus D')DD' = \bigcup \left\{ S \cap S' \;\middle|\; \begin{array}{ll} & S \subseteq View(\alpha, D) \\ \wedge & S' \subseteq View(\alpha, D') \\ \wedge & S \in D \\ \wedge & S' \in D' \end{array} \right\}$$

Given that the belief α contains no universes with conflicting states of D or D', $View(\alpha, D) \in D$, and $View(\alpha, D') \in D'$, so:

$$InBoth(View(\alpha, D \uplus D'), D, D') = View(\alpha, D) \cap View(\alpha, D')$$

□