Online Payments by Merely Broadcasting Messages
(Extended Version)

Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Matej Pavlovic, Petr Kuznetsov
Matteo Monti, and Athanasios Xygkis
EPFL *IBM Research* *LTCI, Télécom Paris*
Institut Polytechnique Paris
Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh
DFINITY *Informal Systems* *National Research University*
Higher School of Economics

***Abstract*—We address the problem of online *payments*, where users can transfer funds among themselves. We introduce *Astro*, a system solving this problem efficiently in a decentralized, deterministic, and completely asynchronous manner. Astro builds on the insight that consensus is unnecessary to prevent double-spending. Instead of consensus, Astro relies on a weaker primitive—Byzantine reliable broadcast—enabling a simpler and more efficient implementation than consensus-based payment systems.**

In terms of efficiency, Astro executes a payment by merely broadcasting a message. The distinguishing feature of Astro is that it can maintain performance robustly, i.e., remain unaffected by a fraction of replicas being compromised or slowed down by an adversary. Our experiments on a public cloud network show that Astro can achieve near-linear scalability in a sharded setup, going from $10K$ payments/sec (2 shards) to $20K$ payments/sec (4 shards). In a nutshell, Astro can match VISA-level average payment throughput, and achieves a $5x$ improvement over a state-of-the-art consensus-based solution, while exhibiting sub-second 95^{th} percentile latency.

I. Introduction

Online payment systems promise secure financial transactions despite distrustful parties. Transactions need to be processed correctly despite crashes and even Byzantine (i.e, malicious) behavior of a fraction of the participants [54]. Popular examples of payment systems include centralized solutions such as PayPal or VISA, as well as decentralized ones like Bitcoin [66] and Ethereum [81]. Numerous newer alternatives are also appearing, claiming new grounds in terms of performance or security [2], [10], [37], [39].

While many payment systems [10], [81] allow for more general transactions (known as smart contracts) [27], in this paper we focus exclusively on *payments*: allowing a participant Alice to transfer funds to a beneficiary Bob if Alice's balance is high enough. Payments represent the largest application of blockchains today, they have driven blockchain systems from their very beginning (Bitcoin) and continue to do so (Facebook's Libra and many others [32], [35], [45], [46], [58], [63], [68], [78]).

We introduce *Astro*, a decentralized payment system capable of matching the performance of the largest centralized solutions (e.g., $65K$ peak, $7K$ average transactions per second, as recently reported by VISA [77]) for payments.

Astro provides honest participants with *robust* performance, namely stable throughput and latency; this holds independently of network scheduling (i.e., asynchrony) and of compromised replicas, as long as no more than 1/3 of the replicas are affected. Systems building on total order (i.e., agreement), in contrast, are often susceptible to throughput degradation due to a single slow replica, typically the leader. This is an issue that received significant attention in the literature [9], [15], [29], [34], [64], which we discuss in detail (§VII) and also quantify experimentally (§VI-D).

An important insight underlying Astro is that totally ordering all payments can be avoided. Indeed, recent theoretical results show that total order (and hence consensus) is not necessary for preventing double-spending [45], [46]. The main contribution of this paper is to apply this insight by building, for the first time, an asynchronous deterministic payment system that is decentralized and consensus-free, and reporting on the empirical evaluation of this system.

Roughly speaking, instead of requiring a total order, *we give clients direct control over (the ordering of) the payments they initiate.* Prior solutions require agreement—usually via an expensive consensus protocol [11], [36], [79]—on the order across the payments of *all* clients. Each client in Astro independently orders their payments, thus maximizing the degree of concurrency and improving efficiency. As a result, a payment operation essentially reduces to broadcasting a message. A weak broadcast primitive, called Byzantine reliable broadcast (BRB) is sufficient for this purpose [18], [43], [46]. This primitive can be implemented in an asynchronous network, unlike consensus and total order broadcast [36]. The performance of Astro, even in uncivil executions, is only limited by the speed of honest participants.

To record payment operations, Astro maintains a log separately for each client. Whenever Alice makes a new payment, she announces—through the broadcast layer—

Author names appear in alphabetical order. This is an extended version of a conference article, appearing in the proceedings of the 50th IEEE/IFIP Int. Conference on Dependable Systems and Networks (DSN 2020). This work has been supported in part by the European grant 862082, AT2 – ERC-2019-PoC and in part by a grant from Interchain Foundation.

her intent to record this payment in her (replicated) log. Payments in her log are ordered by sequence numbers she assigns herself. Astro guarantees that only Alice, the *spender*, may record new payments in her log; we call this abstraction an exclusive log, or *xlog* for short.

Essentially, preventing Alice from double-spending means preventing her from reusing sequence numbers. To do so, the broadcast layer in Astro provides Byzantine resilience. This ensures that a malicious client cannot broadcast two different payments with the same sequence number. For example, Alice cannot broadcast a payment *a* for beneficiary Bob with sequence number *s*, and for that same sequence number, announce a different payment *a'* for beneficiary Carol. At most one of these conflicting payments passes through the broadcast layer. As a result, Alice cannot double-spend.

Astro distinguishes between *clients* of the system and *replicas* that operate the payment system. Clients usually connect to the system infrequently to submit payments and check their balance. Intuitively, each client is a lightweight participant and thus relies on a certain replica—called a *representative*—to broker her payments via broadcast. Nevertheless, each client controls the ordering of her own payments. Replicas maintain the system state (i.e., client *xlogs*), remain well-connected to each other, and implement the broadcast-based replication layer. Payments are safe and live as long as the spender and 2/3 of the replicas, including the representative replica handling the request, are correct.

This distinction between *client* and *replica* allows the number of clients in Astro to scale independently of replicas; a client may, of course, be its own representative. The broadcast layer (implemented by replicas) relies on quorum systems [60] to ensure Byzantine resilience, and consequently does not scale beyond tens or hundreds of replicas. The number of clients, on the other hand, can be orders of magnitude larger.

For pedagogical reasons, we proceed in an incremental manner. We first discuss an implementation of Astro without using digital signatures, before moving to a more efficient scheme with digital signatures and fewer messages. To scale the number of replicas in Astro, we employ a sharding scheme: We partition the system state and replicate each partition among a subset of replicas. Sharding a payment system is difficult if payments need to be totally ordered (i.e., based on consensus): Approving a cross-shard payment requires all involved shards to coordinate, usually via a 2PC protocol [51], [84]. We sidestep this major difficulty because *the shard of the spender can—in our case—unilaterally approve a cross-shard payment*. Astro requires no cross-shard coordination on the critical path of payment execution. The beneficiary receives her funds via an asynchronous notification mechanism after the spender's shard approves it. Again, for simplicity of presentation, we present first the non-sharded case before explaining the sharded solution.

We evaluate Astro on a public wide-area cloud network (Amazon EC2). We show that even without sharding and even in synchronous and failure-free executions, Astro outperforms a state-of-the-art consensus-based payment system. Considering four shards with 52 replicas per shard, Astro can sustain up to $20K$ payments per second at sub-second (95^{th} percentile) latency. But more importantly, Astro provides *robust performance*: In executions where some replica crashes or suffers from high network latencies, overall throughput is unaffected (except for the failed replica). Leader-based consensus systems can experience throughput degradation in such situations, to the point where payment execution blocks altogether when the leader is affected, as we show empirically.

Contributions. We design Astro with a focus on *payments* for a *permissioned* model. Our system lacks some capabilities compared to mature blockchains (e.g., Sybil resistance, smart contracts, or full decentralization as Bitcoin or Ethereum) or global payment systems (e.g., negative balance, fraud detection as VISA). We do not intend Astro to replace such systems, but rather demonstrate the efficiency and power of broadcast for improving existing solutions.

Astro circumvents consensus-inherent complexities, being the first payment system that is completely asynchronous, deterministic, and guarantees robust performance. In summary:

1) Astro introduces the abstraction of an **exclusive log**: A record of client payments uniquely controlled by a certain client. Astro maintains the consistency of exclusive logs through a **weak broadcast primitive**, thus maximizing concurrency and efficiency.
2) Astro is fully **asynchronous**, including support for an asynchronous sharding mechanism for scalability.
3) Our Astro implementation can match the performance, with respect to payments, of centralized solutions (e.g., VISA) in a **robust** manner.

The rest of this paper is organized as follows. We first overview Astro (§II) and then detail its payment protocol (§III). We describe our two implementations of Astro (§IV), and present our asynchronous sharding (§V) scheme. Then we discuss a thorough experimental evaluation of Astro (§VI) and present related work (§VII). In the appendix of this paper, we provide additional details on asynchronous reconfiguration (§A) and the broadcast layers of Astro (§B).

II. Overview

At the heart of Astro lie two building blocks that are closely related to each other. These distinguish our payment system from prior solutions, namely: (1) exclusive logs, or *xlogs*, and (2) a broadcast-based replication layer.

Exclusive Logs. An *xlog* is an append-only log comprising all the *outgoing* payment operations initiated by a certain client. Intuitively, the *xlog* of Alice can be seen as her personal ledger of expenditures. Alice is exclusively allowed

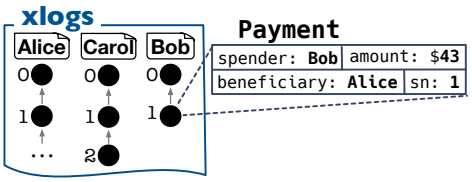

Fig. 1: System state in Astro, consisting of multiple *xlogs* (exclusive logs). Each *xlog* contains payments operations having the same spender (i.e., belonging to the same client). For example, Bob's *xlog* comprises two operations; the second of these is a payment of $43 from Bob to Alice, having sequence number 1.

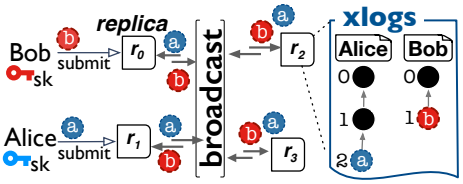

Fig. 2: Payment protocol overview in Astro. When Alice wants to make a payment a, she simply submits a to her representative replica r_1. This replica handles the broadcasting of Alice's payment. Eventually, all correct replicas deliver a, append this payment to Alice's *xlog* (on position 2), and update client balances accordingly to reflect this payment.

to append payments to her *xlog*, and we refer to Alice as the *owner* of her log.

It is Alice herself who establishes the ordering of payment operations in her *xlog*, by assigning a sequence number to each payment. Besides a sequence number, each payment also specifies the *spender* (which is always Alice in this case), the *amount*, and the *beneficiary* of the payment.

Astro's state consists of multiple *xlogs*, one per client, as we sketch in Figure 1. In the basic version of Astro, each replica holds a copy of the entire state (we revise this to consider sharding in §V).

In a static system, storing *xlogs* could be completely avoided, by only storing balances and a single sequence number for each client. Storing the *xlogs* is crucial for reconfiguration of Astro, i.e., for dynamically changing system membership (§A) and to enable auditability.

Consistent Replication of *xlogs*. The goal of the replication layer in Astro is to keep all *xlogs* consistent across replicas despite Byzantine failures. To do so efficiently, we exploit an idiosyncrasy of *xlogs*, namely that each such log restricts append access to the (authenticated) owner client. Consequently, we never have to deal with concurrent modifications on a *xlog*. Each client can modify their own *xlog* autonomously: Astro supports concurrent modification of any number of *xlogs*.

Each client is associated with a single replica acting as its *representative*. A single replica can represent many clients. The representative is in charge of broadcasting the client's payments to other replicas, and corresponds to a *broker* or a *bank*. Akin to a real bank, only the representative can broadcast outgoing payments for a client's *xlog*. All payments still have to be ordered and submitted by the client. Unlike with banks, however, multiple replicas in Astro replicate each client's data (*xlog*).

A client performs a payment by submitting it to her representative r. Replica r ensures that all copies of the client's *xlog* are updated consistently. To this end, replicas implement a broadcast primitive guaranteeing the following crucial property: no client can announce two conflicting payments (i.e., with the same spender) for the same sequence number, despite Byzantine clients and/or replicas. In other words, Astro guarantees total order within—but not across—*xlogs*, departing from prior designs that employ a total order across all payments (Figure 2). From the clients' perspective, Astro provides FIFO guarantees [49], [56].

As we pointed out, current decentralized payment systems achieve consistent replication by executing a consensus protocol [10], [51], [66], while also tackling broader problems (e.g., implementing smart contracts). In many cases, consensus poses a performance bottleneck and is the usual suspect in problems regarding correctness or complexity [1], [24], [29], given its numerous impossibilities and inherent tradeoffs [11], [36], [40], [64], [79].

In Astro, we replace the consensus building block with a broadcast layer. Formally, Astro builds on Byzantine reliable broadcast (BRB). This should not be confused with classic Byzantine Agreement (BA), which is *unsolvable* in the asynchronous model we assume [36]. The BRB primitive is not novel, appearing in the literature for over 30 years, starting with Bracha & Toueg [18], [19]. The crucial difference to BA that allows asynchronous implementations of BRB is termination: BA always guarantees termination, whereas BRB does not guarantee this property if the spender is faulty [45]. Stated differently, if the spender client proposes two conflicting payments (double-spending) under BRB, it is possible that no payment will ever execute.

III. Payments in Astro

Astro is a replicated system running on N *replicas* of an asynchronous network. The replicas implement a broadcast-based replication layer and maintain the full system state, which they update consistently to reflect client payments. Both clients and replicas hold an identifying public/secret key-pair. We assume that *(1)* replica key-pairs are distributed in advance among all replicas, which makes Astro a permissioned payment system, and *(2)* the mapping of clients to their representative replicas is publicly known. We assume less than $N/3$ of replicas to be Byzantine. This is a standard assumption, but we revisit this aspect later, when we introduce partial replication via sharding (§V). We now describe the basic payment protocol.

At a high level, payment execution comprises three steps:

1) The client **submits** a payment a to her representative.
2) The representative **broadcasts** a to all replicas.
3) Replicas locally **approve** payment a and append it to their local copy of the corresponding client's *xlog*.

If the client and representative replica are both correct, each of these three steps is guaranteed to terminate. A correct client, however, is unaffected by other Byzantine clients. Specifically, no client will ever be able to double-spend or

prevent any other client from performing payments, as long as less than $1/3$ of replicas are malicious.

We now describe the three aforementioned steps in detail. For presentation simplicity, we use pseudocode inspired by Golang which we assume to execute atomically.

Submitting a Payment. In Listing 1 we describe the algorithm a client Alice implements to submit a payment. First, she creates a payment message comprising the identity of the spender (herself), the sequence number she assigns to this payment, as well as the identity of the beneficiary, and the amount. Alice then increments her sequence number, and finally sends the payment to her representative replica through an authenticated channel (line 7).

```
1 @executes at spender Alice
2 @local state: Client Alice;
3               Sequence number mySN;
4 func Pay(Client b, Amount x):
5   a := ⟨Alice, mySN, b, x⟩
6   mySN += 1 // Increment our sequence number.
7   Send(a) // Submit the payment to her representative.
```

Listing 1: Client Alice submits a new payment.

Broadcasting a Payment. When the representative receives Alice's payment, it broadcasts this payment among the replicas using the underlying Byzantine reliable broadcast (BRB) layer. BRB ensures that all correct replicas will eventually deliver Alice's message if her representative replica is correct. This layer implements a *consistency check* ensuring that no two correct replicas deliver a different message for the same sequence number of a certain client. We discuss the broadcast layer in more detail later (§IV).

Approving a Payment. Upon delivery of a payment message from the broadcast layer, each replica locally *approves* Alice's payment, and then *settles* it (see lines 13, 14 in Listing 2).

```
8 @executes at all system replicas
9 @local state: SeqNrMap    sn[..] // last SN per client
10              BalancesMap bal[..] // balances per client
11              XLogMap     xlogs[..] // xlogs of clients
12 callback Deliver(a)
13   approve(a) // Blocks waiting for approval of this payment
14   settle(a) // Apply the payment locally
```

Listing 2: A payment *a* is ready. Each correct replica runs this callback upon delivering *a* from the underlying broadcast layer.

Approval. The approval procedure is described in Listing 3. Each replica in Astro executes this procedure with the goal of ensuring two important properties:

1) All Alice's preceding payments are approved (line 17).
2) Alice has sufficient funds for her payment, as reflected by her balance (line 18).

If both Alice and her representative are correct, these conditions may be unfulfilled at replica q only if q has not yet approved either:

1) Alice's preceding payment, or
2) Some other payment crediting Alice.

In such a case, q simply waits until both conditions are satisfied. Under normal conditions, correct clients would initiate payments which they can fulfill straight away. Nevertheless, it can be useful to allow Alice to initiate payments despite not having enough funds to settle them right away. Such payments (and all subsequent ones) will not be approved until Alice has sufficient balance.

```
15 func approve(a)
16   let a be ⟨Alice, n, _, x⟩
17   wait until sn[Alice] = n - 1 // Approval criterion (1)
18   wait until bal[Alice] ≥ x // Approval criterion (2)
```

Listing 3: Payment approval. Every replica executes this to approve a payment *a*, assuming spender Alice.

Settling. As the final step in payment execution, each replica *settles* this payment (Listing 4), i.e., updates the balances of the spender and beneficiary, updates the sequence number of the spender client, and records the payment in the spender's *xlog*. Note that maintaining the whole history of payments in the *xlog* is not strictly necessary for the safety of the basic payment protocol. In a *static* system, storing the balances and sequence numbers for each client suffices. Yet, having this log enables auditability and supports a system where the set of replicas may change for growth, repair or reconfiguration (§A).

```
19 func settle(a)
20   let a be ⟨Alice, n, b, x⟩
21   bal[Alice] -= x // Withdraw from Alice's balance
22   bal[b] += x // Deposit to beneficiary
23   sn[Alice] += 1
24   xlogs[Alice].append(a)
```

Listing 4: Payment settling procedure. Each replica executes this protocol to transition a payment *a* to the final, settled state.

Client notification. By default, we assume clients to be lightweight and intermittently connected, so we omit a specific step of notifying clients that their transaction settled (or is cleared in the system). It is simple, however, to achieve end-to-end notification, by having the client query her representative for the status of the payment. The latter can reply after it has finished with the *settle* step.

Checking the Balance. A client can check her balance by querying her representative *r*. To obtain the balance, replica *r* simply returns the value from the *bal* state (defined on line 10, Listing 2).

IV. A Tale of Two Versions

We now turn our attention to the broadcast layer in Astro. Replicas use this layer to replicate client payments consistently, and it is implemented using a BRB protocol. The BRB interface has two methods. First, a replica *r* can use *Broadcast(a)* to reliably send payment *a* to all replicas in the system. Second, the *Deliver(a)* callback triggers at any correct replica to notify about the delivery of payment *a*. The broadcast layer is aware of the payload *a*, which specifies: the spender *s*; sequence number *n*; beneficiary

b; and amount *x*. We denote the pair *(s,n)* to be the *identifier* of payment *a*. We now define the properties of the broadcast layer, inspired by [59], where payment identifiers are particularly important:

- *Agreement*. If a correct replica delivers a payment *a* with identifier *(s, n)*, then no correct replica delivers a payment *a' ≠ a* with the same identifier.
- *Integrity*. A correct replica delivers a payment *a* at most once, and under the condition that *a* is broadcast by a replica *r*.
- *Reliability*. If the broadcaster replica of payment *a* is correct, then all correct replicas eventually deliver *a*.
- *Totality* (optional). If a correct replica delivers payment *a*, then every correct replica eventually delivers *a*.

There is a rich history of protocols implementing BRB [18], [21], [22], [61]. We mark totality property as optional because there exist BRB protocols which in fact do not offer this property by default. Such protocols are appealing because they are more efficient. If totality is missing, however, an adversary can mount a *partial payments* attack against our payment protocol, as follows. Suppose Alice issues a payment to Bob, who initially has $0. Let Alice's representative r_A be malicious, whereas the representative r_B of Bob is correct. In the absence of totality, since r_A is malicious, only r_B would deliver and settle Alice's payment, while Bob's *xlog* in any other replica still has a balance of $0. Bob cannot spend the $10 he received, because there are no $2f + 1$ replicas with the updated version of Bob's *xlog*.

We implement and evaluate two versions of BRB, and thus obtain two versions of our system: Astro I and Astro II. Astro I uses a BRB protocol [18] that has a similar communication pattern to our consensus-based baseline and allows for a fair performance robustness comparison (§VI-D). Astro II, on the other hand, uses stronger cryptographic primitives to reduce communication complexity, achieve higher performance, and enable sharding. Additionally, Astro II lacks the totality property, so we compensate for that with an additional mechanism to prevent the attack we mentioned above.

Both BRB protocols underlying Astro I and Astro II assume less than a third of replicas to be Byzantine and offer the API we specified earlier. We now describe the broadcast protocols in our systems; for the pseudocode, we refer the interested reader to the appendix (§B).

A. Broadcast Protocols & Astro Versions

Astro I implements BRB based on Bracha's algorithm [19]. Let *a* be a payment with identifier *(s, n)* that the representative replica *r* is broadcasting on behalf of spender client *s*. This protocol relies on authenticated links, e.g., via message authentication codes (MACs), and comprises three phases.
(1) PREPARE. To broadcast payment *a*, correct replica *r* simply sends *a* to all replicas in the system.
(2) ECHO. The first time a replica *q* receives a payment with identifier *(s, n)*, it sends an ECHO message for this payment to all replicas in the system.
(3) READY. In this last phase of the protocol, every replica *q* waits to collect a Byzantine quorum [60] of ECHO messages for tuple *(s,n)* and then *q* sends a READY message. Alternatively, replica *q* may send a READY after observing $f + 1$ READY messages. A correct replica delivers payment *a* after gathering $2f + 1$ matching READY messages for *a* and after having delivered the previous payment of client *s*, i.e., the payment with identifier *(s, n-1)*.

Observe that Bracha's protocol entails two phases (ECHO and READY) of all-to-all communication, i.e., has message complexity of $O(N^2)$. On the plus side, this protocol uses MACs, thus it is not computationally intensive.

Astro II implements the broadcast layer using a BRB protocol with linear ($O(N)$) message complexity. At a high-level, this protocol employs digital signatures, and also comprises three phases. The first phase, called PREPARE, is identical to the first phase of the broadcast protocol of Astro I. The other two phases of this protocol are as follows:
(2) ACK. Upon receiving payment *a* from replica *r*, every replica *q* verifies whether there exists *a' ≠ a* previously received for identifier *(s, n)*. If this is not the case, then *q* sends a signed ACK message (i.e., a signed hash) of *a* directly to replica *r*. Otherwise, replica *q* does nothing.
(3) COMMIT. Upon gathering a Byzantine quorum [60] of matching acknowledgments for payment *a*, replica *r* sends to all other replicas a COMMIT message, comprising the gathered acknowledgments. Each correct replica delivers *a* after receiving a correct commit message for *a*.

To prevent the partial payments attack, we introduce *dependencies* in Astro II. A correct replica that approved Alice's payment, unicasts the signed approval called CREDIT message to Bob's representative, and allows Bob to prove the existence of a payment crediting his account unequivocably with $f + 1$ such CREDIT messages. To this end, Bob's representative replica collects and aggregates CREDIT messages for the same incoming payment into a dependency certificate for Bob's *xlog*. If Bob's representative fails in any way, this certificate is not lost; the certificate is permanently stored as CREDIT messages, distributed across the replicas that approved the payment, so it can be reconstructed directly from these replicas.

Note that replicas must keep track of used certificates, ensuring that each payment takes effect not more than once. This way, it is impossible for replicas to mistakenly apply a dependency twice (e.g., double-deposit, as in a replay attack). Listings 3 and 4 have to be adjusted to take dependencies into account, see pseudocode in §B.

Certificates also play an important role in a sharded environment, as they are transferable across shards: They enable Bob to spend the money mentioned in the dependency not only within his representative's shard, but also across shards

(§V). Whenever Bob submits an outgoing payment, his representative replica attaches the accumulated dependencies alongside the outgoing payment.

Comparison. Astro II is well-suited for environments where bandwidth is scarce (e.g., WAN), whereas Astro I has lower computation requirements and is therefore suited for systems where computing resources are more scarce. Given a batching scheme, however, we can amortize the cost of digital signatures in Astro II, as we describe later (§VI-A). Moreover, we expect the typical deployment of our system to be a wide-area network where bandwidth is the scarce resource. Because of these reasons, Astro II has an edge over Astro I in terms of performance—a hypothesis we quantify in our experimental evaluation (§VI-C).

The two systems handle transitive transactions differently. Astro I does not reject insufficiently funded transactions (line 18, Listing 3), instead it queues them until enough funds arrive. Queuing is necessary even with totality, since different replicas may receive crediting transactions at different times. Instead, the dependencies mechanism in Astro II allow the spender's representative to prove that the spender has sufficient funds to issue a payment.

There is an additional important distinction between Astro I and Astro II: the latter is amenable to sharding. To understand why this is the case, we observe that sharding requires the approval of payments across different shards. In other words, some shard *s1* has to convince some other shard *s2* that *s1* approved a certain payment and *s2* can settle it. Digital signatures simplify this transfer of trust between shards, because the payment of a spender from *s1* appears as a dependency in the *xlog* of the beneficiary in *s2*. Replicas in *s2* accept this dependency when they verify it is signed by $f + 1$ replicas of *s1*. In the next section, we provide the full details of the sharding mechanism which we implement in Astro II.

V. Asynchronous Sharding

So far we described our payment protocol (§III) assuming full replication. In this model, all replicas maintain a full copy of the system state (i.e., *xlogs*) and approve and settle every payment. The full replication architecture is simple to understand and implement, and excels at small scale. This design poses two scalability problems. First, throughput degrades with increasing replica count (as we observe experimentally in §VI-C). Second, each replica has to keep more state as the number of *xlogs* (i.e., clients) increases.

We now refine the architecture of our payment system with sharding, which Astro II implements. Sharding is a well-known technique [3], [5], [16], [51], [57], [80], [84], allowing our system to scale-out in terms of both number of replicas and number of clients. We define a *shard* as a subset of system replicas, and to be associated with a subset of all *xlogs*. We use the notation *s(·)* to denote the shard to which some replica or client "·" belongs. Importantly, sharding requires strengthening our assumption from §III, so that the threshold $N/3$ on Byzantine replicas applies to every shard.

Intuitively, each shard in Astro II executes an instance of the basic payment protocol (§III) for its associated clients. It also incorporates an additional mechanism that not only prevents partial payment attacks, but also supports sharding seamlessly. The broadcast step of Astro II is executed in the shard of the spender, while the Credit messages may be sent to a representative in another shard.

Let us consider a payment of amount *x* from spender *A* to beneficiary *B* and illustrate how Astro II processes it. Let *r* be the representative replica of *A*.

After broadcasting and approving the payment of client *A*, all honest replicas in shard *s(A)* unicast a Credit message to the beneficiary's representative in *s(B)*, indicating the crediting of amount *x* to the balance of client *B*. This message comprises all details of this payment (including the sequence number *n* assigned by client *A*), as well as a signature *sig* indicating the approval of the payment from the perspective of that replica. The representative of *B* interprets f+1 distinct Credit messages as a dependency certificate, i.e., a proof that the payment has been accepted by shard *s(A)*. This dependency certificate is stored at the representative of *B* and gets added to*B*'s balance when the next outgoing transaction issued by *B* is settled by the replicas in shard *s(B)*.

Traditional sharded designs employ a 2PC protocol for coordinating transactions that span multiple shards [12], [38]. The 2PC protocol relies on synchrony and has a delay of 3 communication steps; each such step usually has complexity $O(m)$ and in the Byzantine case it can reach up to $O(m^2)$, where m is the size of a shard [38], [41]. In contrast, our protocol based on the Credit message entails exactly 1 communication step and has overall complexity $O(m)$. In our experiments with Astro II implementing the Smallbank application [33] we observe that this sharding mechanism has negligible overhead (§VI-C2).

The insight enabling such a simple sharding mechanism in Astro is that we *decouple payment processing at the spender from the beneficiary*. In fact, this mechanism is orthogonal to how a payment is executed inside a shard (e.g., using a consensus or a broadcast based protocol).

VI. Experimental Evaluation

We now report on the experimental evaluation of our consensus-free approach to payment systems. We first describe the systems we evaluate, namely Astro I and II and a baseline based on consensus (§VI-A). We also detail our evaluation methodology (§VI-B) and present our comprehensive evaluation, covering both the common-case and performance robustness (§§ VI-C and VI-D).

A. Systems under Evaluation

We build our baseline on top BFT-SMaRt, a mature state-of-the-art BFT SMR (i.e., consensus) implementation [15], used, for example, as the ordering service of Hyperledger Fabric [74][1]. For both Astro systems and BFT-SMaRt under evaluation we assume the optimal threshold of $N = 3f + 1$ replicas, where f bounds the number of faulty replicas.

Batching in Astro I and II. We employ a 1- or 2-level batching scheme, depending on the variant of our system. First, we perform batching at the level of the broadcast protocol. Note that the first step of the broadcast protocol (PREPARE in §IV-A) is identical across our two systems. Briefly, some replica i sending a PREPARE is the one assembling a batch of payments—potentially from different clients—with the goal of amortizing both the cost of message authentication and network processing overheads.

Second, to reduce the overhead of digital signatures necessary for the BRB and the CREDIT messages, Astro II groups together payments for which the beneficiary clients have the same representative replica. Thus, when a replica i builds a batch of payments to be broadcast, it includes sub-batches of payments segregated according to the beneficiary replica. As a result, there are as many signatures for CREDIT messages as there are sub-batches. All payments in the batch are processed together during broadcast, while the payments in the sub-batches are processed together when settling and unicasting.

Even though batching alleviates the computational burden of cryptographic signatures, it relies on the fact that clients have to trust their replicas for not issuing transactions without the former's consent. However, our approach can protect clients from malicious representative behavior if the same protocol adopts end-to-end client signatures.

Cryptography in Astro II. We used ECDSA on the NIST P-256 curve provided from the Golang standard library, which offers adequate performance. To avoid cryptographic operations acting as a CPU bottleneck, we use one signature per batch of 256 payments in the broadcast layer. With this batch size, Astro II's performance is only limited by available bandwidth.

B. Evaluation Methodology

We use Amazon EC2 as our experimental platform. Throughout all experiments, we use commodity-level virtual machines (VMs) of type *t2.medium* [8], equipped with 4GiB of RAM and 2 vCores. Unless we explicitly state otherwise, we deploy each system so that every replica executes on a separate VM. This avoids creating noise in our results, which could arise due to performance interference.

Our deployment setup comprises four Amazon EC2 regions in Europe, namely Frankfurt, Ireland, London, and Paris. On average, the bandwidth and round-trip latency across machines of these four regions is around 30 MiB/sec and $20ms$, respectively. We deploy the replicas of each system randomly across the corresponding regions. This deployment reflects a scenario where participants are localized in one geographic region of the globe (Europe). Later in our experiments, we also introduce network delays at each replica. As a result, we lessen the effect of sub-millisecond latency between replicas in the same region and obtain more realistic conditions with larger latencies (§VI-C2).

We use up to 15 VMs to deploy clients. Each request from a client represents one payment. A request contains three fields (the *spender* and *beneficiary* identities, along with the *amount*) and the client authentication data. The beneficiary and amount fields are random, and each payment operation covers roughly 100 bytes.

For simplicity, we place all client VMs in Ireland. Spreading clients around Europe does not influence our results. Each such VM hosts a varying number of client processes. The number of processes varies greatly, depending on each system and the system size. For instance, to saturate BFT-SMaRt at system size $N = 4$, we use around 800 total client threads; for $N = 100$, 30 threads are sufficient for saturation. For Astro, we require more client threads to reach saturation, since they are capable of higher performance. We report the maximum achievable performance: our experiments assume that all transactions can be settled immediately, i.e. clients have enough balance, so transactions can not be blocked due to insufficient funds.

For throughput, we report on how many payments each system settles per second, labeled *pps*. All experiments have a runtime of 60 seconds, and we present the average result across 3 runs. We also plot the standard deviation, but often this is negligible and not clearly visible in the plots.

In BFT-SMaRt, each client keeps connections to all replicas (a design decision of this protocol) [15]. For this reason, all BFT-SMaRt clients experience similar latencies. In our results we report on the latency as observed by a random client. In our Astro systems, each client connects to a single, random replica. To make all replicas execute payments (which is the most realistic scenario), clients pick and submit their workload to a random replica.

C. Performance Evaluation Results

We seek to answer the broad question of how our asynchronous approach compares in performance, at varying system sizes, with the consensus baseline. We discuss microbenchmarks for latency and throughput in a single shard (§VI-C1), as well as results with the Smallbank [7] benchmark in a sharded scenario (§VI-C2).

[1]In general, there is a notable difference in complexity between consensus—in particular, the Byzantine-fault tolerant versions—and broadcast algorithms. Both Astro implementations require less than $3.5K$ LOC in Golang. Contrast this with `libpaxos` [71], a simple consensus implementation for the crash-only model, stretching over more than $6K$ LOC in C. At the time of its original publication, the BFT-SMaRt implementation counted around $13.5K$ LOC in Java [15, §III].

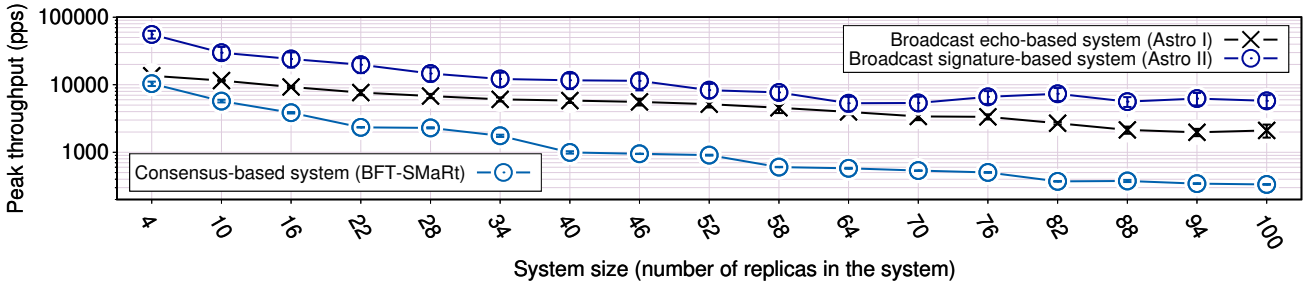

Fig. 3: Throughput vs. system size. We measure peak throughput as we increase the number of replicas in different payment system implementations, one based on consensus (BFT-SMaRt), and two based on broadcast (Astro I and II). We do not employ sharding.

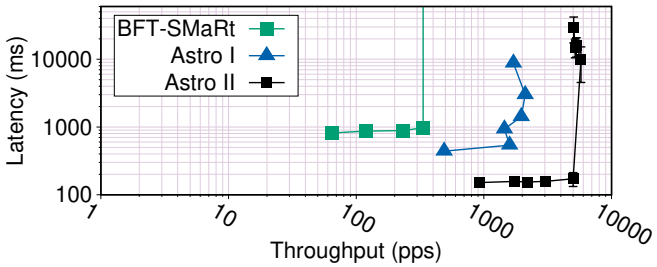

Fig. 4: Latency/throughput. Performance evaluation of three payment systems each running at $N = 100$.

1) Microbenchmarks:

Throughput. In Figure 3 we depict how throughput evolves as a function of system size. For each system size, we plot the peak throughput, i.e., before latency saturates. Note the logscale axis, to better capture performance differences. We increase the system size in increments of 6, starting from the smallest size of 4, until we reach 100.

As an overall observation, our two Astro prototypes outperform the consensus-based solution at every system size we investigate. At small size, all systems exhibit their respective highest throughput. The consensus-based implementation using BFT-SMaRt sustains over $10K$ *pps*, while Astro reaches almost $13.5K$ *pps* and Astro II sustains $55K$ *pps*. The 4x improvement of Astro II over Astro I is owed to the the linear communication complexity of the former system (§IV-A). As can be seen, however, this benefit slowly tapers off with increasing system size. At maximum system size ($N = 100$), the consensus-based system saturates at 334 *pps*; Astro I sustains 6x higher throughput, being able to apply $2K$ *pps*, and Astro II can sustain $5K$ *pps* (a 16x improvement over consensus and 2.5x over Astro I).

Latency-Throughput. We now explore the difference in performance between the consensus-based baseline and Astro I/II at the maximum system size we consider, $N = 100$. As before, all systems are running in a single-shard setup. The results depicted in Figure 4 show how latency evolves with respect to throughput. For clarity sake, the y-axis (latency) starts at $100ms$, and we convey order of magnitude differences using logscale axes.

The consensus-based implementation typically exhibits sub-second latencies. We do not show the 95^{th} percentile latencies because they obstruct visibility, but these are between 1.3 and 1.5 seconds. Latencies in Astro I are more variable, between 400 and $500ms$ prior to saturation, while the $95th$ percentile latencies are on the order of one second. Recall that clients connect to random replicas, which are geographically spread. Astro II exhibits more stable performance and lower latencies: prior to saturation, clients observe a confirmation latency of $200ms$ on average. The 95^{th} percentile latency (at low load) is under $240ms$. The 99^{th} percentile for all these systems are within the same order of magnitude as the 95^{th}.

We remark that the latencies for these three systems are not necessarily at their worst when $N = 100$. We also investigate the same execution at $N = 10$, for instance, and observe only slightly better performance (e.g., latency for Astro II is $150ms$ on average). The latencies do not change considerably because there is a lot of parallelism inherent in the underlying quorum-based protocols, both for consensus and broadcast. This is intuitive: obtaining one response from a particular distant replica takes roughly as much time as obtaining several responses (in parallel) from multiple distant replicas. Primarily, it is throughput that suffers in quorum-based systems, and latency secondarily [29], [76], [79].

An important observation here is that our evaluation concerns the critical part of a payment system, the ordering layer. For the deterministic system model, we are only aware of prior experiments of this layer which considered a maximum system size of $N = 10$, concretely for Hyperledger Fabric [74], which builds on BFT-SMaRt. To conclude this part of our evaluation, for systems of moderate size—up to 100 replicas—broadcast-based systems are simpler and significantly outperform consensus-based solutions for decentralized payments. Even if Astro relies on broadcast, it still employs quorum-gathering to achieve consistent replication; hence the throughput of Astro is inversely proportional to the system size (akin to consensus-based solutions). To avoid this throughput decay and scale to larger systems, we now discuss experiments with sharding.

2) Sharding in Smallbank Application: For a real-world application workload, we use the Smallbank transaction family from the BLOCKBENCH framework [33]; this is a version of the H-Store Smallbank benchmark [25] adapted to the cryptocurrency setting. The application models bank accounts, where the owners of these accounts are clients that can issue several types of transactions. In particular,

# of shards	#	tc delay (ms)	Throughput (Kilo-*pps*) per-shard**total**		Latency (ms) Average**95th** %ile	
			Astro II	BFT-S†	Astro II	BFT-S†
	2	0	7.9**15.7**	1.0**2.0**	204**279**	600**808**
	2	20	5.1**10.2**	0.3**0.5**	479**705**	2245**2673**
	3	0	5.1**15.4**	1.0**3.1**	213**375**	600**808**
	3	20	4.5**13.6**	0.3**0.8**	368**656**	2245**2673**
	4	0	5.0**20.1**	1.0**4.1**	213**259**	600**808**
	4	20	4.5**18.1**	0.3**1.1**	354**620**	2245**2673**

TABLE I: Smallbank sharded benchmark. Performance results for up to 4 shards (each $N = 52$ replicas). †BFT-SMaRt results are upper-bound values based on a single-shard experiment.

accounts can be of either *savings* or *checking* type. Some transactions model payments across two accounts of the same owner, while other transactions deal with the transfer of funds between different owners. For the sake of consistency, hereinafter we refer to bank *accounts* and their *owners* as *xlogs* and *clients*, respectively.

Experimental Setup. We associate each client with two *xlogs* (for checking and savings). Thus same-client transactions at the application level appear as full-fledged payments between two distinct *xlogs* in the underlying layer. We use a multi-shard setup for Astro II, ensuring that both *xlogs* of any client belong to the same shard. Whenever a transaction involves different shards, the cross-shard coordination consists of the Credit message described earlier (§V). For BFT-SMaRt, we use an equivalent setup.

Each shard consists of $N = 52$ replicas uniformly spread among the four EC2 regions of Europe. We execute using $2, 3$ and 4 shards (total of 208 replicas); we limit ourselves to 4 mainly due to financial constraints, but also because it is straightforward to estimate performance at larger scales. Clients attach to a certain replica and simultaneously issue transactions as prescribed by the Smallbank benchmark, meaning that 12.5% of the overall number of transactions are cross-shard. To produce more realistic network conditions, we introduce artificial network delays: We use the traffic control (`tc`) subsystem of the Linux Kernel, and increase inter-replica latencies by $20ms$. Network latency between replicas in Europe is around $20ms$, so having this delay essentially doubles latencies; additionally, this also eliminates any advantage that may arise due to co-location of some replicas in the same EC2 region.

Experimental Results. We provide the results in Table I. We show both per-shard and overall (i.e., total) throughput for a given latency envelope. Astro II sustains the highest per-shard throughput when there are 2 shards. As the number of shards increases (the # column), per-shard throughput slowly decreases: This is because intra-shard payments are more lightweight (lacking the cross-shard notification mechanism) and the number of intra-shard operations decreases with growing number of shards [84]. We observe that the $20ms$ network delay affects performance. The reason is TCP's congestion control: Astro II saturates the links and network delays become the bottleneck.

As Table I shows, performance in Astro II scales well with the number of shards. In absolute numbers, Astro II sustains up to $20K$ *pps* using four shards, with average latencies of around $200ms$. The BFT-SMaRt baseline running on four shards sustains a total throughput of just above $4K$ *pps*; importantly, these result are only for comparison, and represent optimistic upper-bounds. In BFT-SMaRt we omit the cross-shard coordination step, which typically consists of a 2PC protocol posing significant overhead, thus a fully working sharded solution would necessarily sustain less than $4K$ *pps* [51], [84].

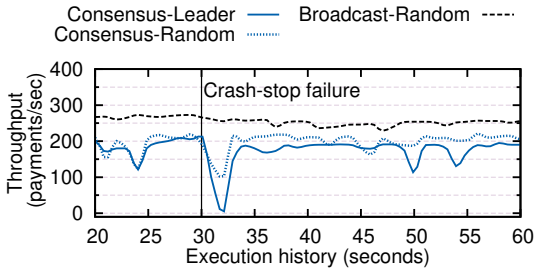

Fig. 5: Throughput robustness during crash-stop failures. We plot throughput when a replica crashes in the consensus-based system (either the leader or a random replica) and Astro I.

D. Performance Robustness

We now investigate how our Astro and the baseline react to two problems that can arise in practice, namely *failure* (e.g., crash) and *asynchrony* (network delays) at a replica. We consider the impact of these issues when they affect a random replica in each system, as well as the case when the leader is affected in the consensus-based system.

Astro I and Astro II have similar robustness characteristics: they are completely decentralized (there is no leader) and making a payment only requires broadcasting a message. To maximize fairness of comparison, we experiment with Astro I, as its message pattern and cryptographic primitives (MAC-based channel authentication) are the most similar to BFT SMaRt.

We study the evolution of throughput within a window of execution of $40s$, ignoring a warm-up period of $20s$. For all these experiments, we introduce asynchrony or failure after $30s$ elapse. To induce asynchrony, we again use the traffic control utility `tc` with the network emulator queuing discipline. We always use a delay of $100ms$. For instance, to introduce such a delay on all packets outgoing from interface `eth0` at a replica, we use the following command:

```
tc qdisc change dev eth0 root netem delay 100ms.
```

We use 10 clients, each running a single thread. The goal is to evaluate these systems below saturation point. If we introduce failures at saturation, this can lead BFT-SMaRt to halt or enter a livelock where the system is unable to do view-change (i.e. leader election). Moreover, at saturation point Astro can sustain the same throughput independently of how many replicas accept client operations; this is because no single replica in our broadcast-based system is a bottleneck. In other words, stopping a replica at saturation point in Astro would not impact throughput, giving an

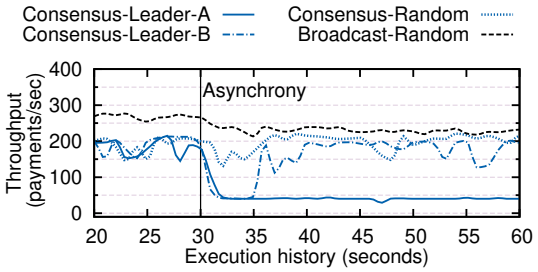

Fig. 6: Throughput robustness during asynchrony. We show, for N=49, how throughput evolves in the consensus- and broadcast-based systems during asynchrony ($100ms$ delay for each outgoing packet) at one replica, either the leader or random.

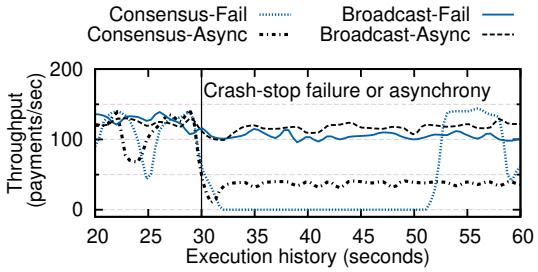

Fig. 7: Throughput robustness. We show how throughput evolves for $N = 100$ when a crash-stop failure or asynchrony affects the consensus-based system or the broadcast-based system.

advantage to our system over the consensus-based solution. We first report results for a system size $N = 49$. We run these experiments with larger and smaller systems, but similar observations emerge as the ones we describe below. For completeness, we also discuss a set of interesting results with a larger system size of $N = 100$.

In Figure 5 we show how the throughput evolves when we introduce a crash-stop failure at a replica ($N = 49$). For consensus, this failure has a severe impact on throughput if the leader is affected (the *Consensus-Leader* curve), because the view-change protocol has to execute. The throughput drops to 0 while this protocol runs, typically a few seconds. For larger system sizes, this protocol can take longer to execute, as we will show later. When a random replica fails in the consensus-based system (*Consensus-Random*), there is a brief decrease in throughput when all clients and replicas get disconnected from the affected replica, but thereafter performance recovers. In Astro I we stop a random replica (*Broadcast-Random*), and thereafter throughput drops from 270 pps to 250 pps, which accounts for the failed replica which was handling roughly 20 pps from one of the clients. This decrease is barely visible in the plots.

Figure 6 shows how asynchrony impacts the performance in the two systems ($N = 49$). We depict two separate executions for the case of consensus when the leader is affected, because there are two possible outcomes. First, it may happen that throughput decreases and remains that way; this is the *Consensus-Leader-A* timeline. Second, the system can go through a view-change (*Consensus-Leader-B*) because the leader is too slow or its buffers can overflow and packets get dropped (inflating the replica-to-replica delay). Clearly, initiating a view-change is preferable in this case, because the throughput penalty is smaller. There is a well-known tradeoff, however, in choosing the view-change timeout [29], [64]: initiating view-change too aggressively can lead to frequent leader changes even in good conditions, which can erode performance on the long-run.

When a random replica is affected with asynchrony in the consensus-based system (*Consensus-Random* execution in Figure 6), performance drops briefly because there is a quorum switch, i.e., the affected replica is replaced by a different one in the active quorum [11]. For the broadcast-based system (the *Broadcast-Random* timeline), asynchrony affects performance in the same manner in which a failure does. Concretely, the affected replica no longer sustains the same amount of client operations, so the overall throughput reduces correspondingly.

We also show results for the case of a larger system size ($N = 100$) in Figure 7. There are four timelines in this execution, as follows. For the consensus-based solution, we show what happens when there is either a crash-stop failure or asynchrony at the leader. In the former case (*Consensus-Fail*), the view-change protocol kicks off and lasts for roughly 20 seconds, while throughput stays at zero; this is similar to the *Consensus-Leader* execution in Figure 5. In the latter case (*Consensus-Async*), performance degrades and stays that way for as long as the affected replica remains the leader; this is similar to the *Consensus-Leader-A* in Figure 6. For the broadcast-based solution we consider the same two issues affecting a random replica. When either of these issues arises (*Broadcast-Fail* or *Broadcast-Async*) throughput is affected correspondingly with the number of operations that the failed replica is handling (and which is unable to continue). Note that Astro relies on fate-sharing [28] between a client and its representative: when a replica stops, all the associated *xlogs* naturally stop as well.

We conclude with two general observations. First, Astro does not suffer from overall (i.e., global) throughput degradation that can happen in leader-based protocols such as most consensus algorithms. Second, our system does not rely on timeouts for liveness. Simply put, Astro progresses at the speed of the network. These two advantages are closely linked, and they both follow from the asynchronous nature of the broadcast protocol we rely on.

VII. Related Work

Since Nakamoto's original Bitcoin paper [66], follow-up payment systems seek to prevent double-spending by establishing a total order of transactions, i.e., solving consensus. Consequently, a lot of effort has been devoted to improving the consensus layer.

Dealing with the Consensus Bottleneck. Research on consensus algorithms has shown significant breakthroughs and modern protocols quote impressive performance numbers [83], [84]. To push performance even further, several interesting systems address the consensus bottleneck with sharding [5], [51], [57], [84], [80]. Approving a cross-shard

payment, however, requires special coordination [51], [80], [84]. Off-chain payment networks such as Lightning [68] and Raiden [67] strive to minimize the impact of consensus protocols. They allow parties to move funds from a blockchain into high-performance payment *channels*, for which the final balance is settled back on the blockchain after use. Recent advances in this field rely on trusted hardware to provide an asynchronous protocol for all interactions [55]. These results bring noticeable improvements over Bitcoin, enabling good scalability and very fast payments. Nevertheless, the underlying problem of consensus is only reduced, not overcome. In Astro we take a different approach: We provide robust performance by avoiding consensus protocols altogether.

Performance Instability of Consensus. Recent work emphasizes the problem that performance of consensus algorithms hangs on a fragile thread, namely their view-change sub-protocol [20], [40], [82]. HotStuff, for instance, proposes to absorb view-change in the common-case consensus algorithm; this sidesteps performance instability but comes with the cost of a higher common-case latency [82].

Another line of research circumvents the view-change issue with randomized consensus protocols, such as HoneyBadgerBFT [64] or BEAT [34]. Both are based on work by Ben-Or et al. [13] combining reliable broadcast (BRB) with binary Byzantine agreement (ABA). In a nutshell, these protocols comprise a broadcast phase (where replicas form encrypted batches of payments which they disseminate using BRB), an agreement phase (involving N instances of the ABA protocol to agree on a common set of batches), and a decryption phase (requiring each replica to obtain $f + 1$ decryption shares). These protocols push the performance of consensus by carefully choosing modern cryptographic tools and system parameters.

Various *leaderless* consensus protocols have been proposed, for both crash and Byzantine models [17], [30], [31], [52], [65]. These protocols, however, either make use of some form of coordinator in corner-cases, or rely on additional synchrony assumptions, or provide probabilistic guarantees. For example, a thorough study of the appendix of [65] reveals that EPaxos only ensures probabilistic liveness and, as shown recently [75], has correctness issues.

Astro is deterministic and fully asynchronous. It does not solve the general consensus problem, but instead focuses on payments. Since our system relies exclusively on BRB and no BA primitive is necessary, Astro is simpler and more efficient than modern leaderless randomized consensus protocols.

Avoiding Consensus Protocols. Recent theoretical results [45], [46] show that consensus is unnecessary for implementing a payment system, contrary to popular belief. For instance, [45] showed that the basic double-spending problem, as defined by Nakamoto [66], can be cast as a sequential object type and that it has consensus number 1 in Herlihy's hierarchy [48]. Whilst the observation that consensus is unnecessary to prevent double-spending in a theoretical context has been made, we apply this insight for the first time to obtain Astro: a full system solution (design, implementation, evaluation), that is also efficient.

The exclusive logs in Astro resemble conflict-free replicated data types (CRDTs) [72]. Similar to a CRDT, different *xlogs* support concurrent updates while preserving consistency. Since each log has a unique owner, we rule out the possibility of conflicting operations on each log. Note, however, that appending a payment to the history of a client's *xlog* A is not commutative, i.e., any two payments within the same history need to be ordered with respect to one another. This is a departure from classic CRDTs, but it ensures in our case that the state at correct nodes always converges to a consistent version.

Our *xlog* abstraction in Astro resembles the acyclic graph (DAG) in various novel payment systems [26], [47], [50], [73]. The distinctive feature of Astro, however, is that consensus is entirely sidestepped—whereas all prior solutions we are aware of, even those building on a DAG, employ a consensus algorithm to order payments.

Broadcast Protocols. BRB protocols have a long tradition starting with the algorithms of Bracha and Toueg [18], [19]. Later work refined and improved performance and properties of these algorithms [22], [59], [61], [69]. Asynchronous verifiable information dispersal algorithms [23] are closely related to BRB protocols, and both of these classes of protocol represent an essential building block in modern asynchronous consensus protocols [34], [64].

There are several ways to improve the scalability of broadcast protocols. Sharding—the technique we recalled out above and we adopt in Astro II—is a clean approach to scalability, as it allows each shard to maintain the same (deterministic) properties as a non-sharded system. Other approaches, such as clustering [41], [70], probabilistic quorum-based [62], or sample-based [44], typically yield a design providing probabilistic guarantees.

VIII. Conclusions

Astro is a decentralized payment system that can sustain $20K$ payments/sec in a deployment of 200 replicas, while exhibiting sub-second latency. It can do so by not relying on any consensus layer and thus remaining mostly unaffected by network asynchrony and compromised replicas. We do not claim Astro to be a silver bullet: we only focused on payments and did not consider the general abstraction of state machine replication, e.g., as might be required by smart contracts. Yet, determining the exact set of problems (besides payments) that can be addressed by Astro's broadcast layer is an open problem. We also identified several avenues for improving Astro, namely: *(1)* a more flexible representation scheme, instead of the fixed dependency between a client and its representative replica, *(2)* use more advanced cryptographic primitives

(e.g., threshold signatures, key revocation schemes), *(3)* a fine-grained state transfer protocol for reconfiguration, and *(4)* a hybrid system that incorporates asynchronous payments and consensus-based smart contracts.

References

[1] Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R., and Martin, J.-P. Revisiting fast practical byzantine fault tolerance. *arXiv:1712.01367* (2017).

[2] Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Spiegelman, A. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *CoRR abs/1612.02916* (2016).

[3] Adya, A., Myers, D., Howell, J., Elson, J., Meek, C., Khemani, V., Fulger, S., Gu, P., Bhuvanagiri, L., Hunter, J., et al. Slicer: Auto-Sharding for Datacenter Applications. In *OSDI* (2016).

[4] Aguilera, M. K., Keidar, I., Malkhi, D., Martin, J.-P., Shraer, A., et al. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS 102* (2010).

[5] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., and Danezis, G. Chainspace: A sharded smart contracts platform. *arXiv preprint arXiv:1708.03778* (2017).

[6] Alchieri, E., Bessani, A., Greve, F., and Fraga, J. Efficient and modular consensus-free reconfiguration for fault-tolerant storage, 2016.

[7] Alomari, M., Cahill, M., Fekete, A., and Rohm, U. The cost of serializability on platforms that use snapshot isolation. In *2008 IEEE 24th International Conference on Data Engineering* (2008), IEEE, pp. 576–585.

[8] Amazon. Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/. [Online; accessed 19-Sept-2018].

[9] Amir, Y., Coan, B., Kirsch, J., and Lane, J. Prime: Byzantine replication under attack. *IEEE Transactions on Dependable and Secure Computing 8*, 4 (2010), 564–577.

[10] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., Caro, A. D., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolic, M., Cocco, S. W., and Yellick, J. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018* (2018), pp. 30:1–30:15.

[11] Antoniadis, K., Guerraoui, R., Malkhi, D., and Seredinschi, D.-A. State Machine Replication is More Expensive Than Consensus. In *DISC* (2018).

[12] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011).

[13] Ben-Or, M., Kelmer, B., and Rabin, T. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing* (1994), ACM, pp. 183–192.

[14] Bessani, A., Santos, M., Felix, J., Neves, N., and Correia, M. On the efficiency of durable state machine replication. In *USENIX Annual Technical Conference* (2013), pp. 169–180.

[15] Bessani, A., Sousa, J., and Alchieri, E. E. State Machine Replication for the Masses with BFT-SMaRt. In *DSN* (2014).

[16] Bezerra, C. E., Pedone, F., and Van Renesse, R. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2014), IEEE, pp. 331–342.

[17] Borran, F., and Schiper, A. A leader-free byzantine consensus algorithm. In *Distributed Computing and Networking* (Berlin, Heidelberg, 2010), K. Kant, S. V. Pemmaraju, K. M. Sivalingam, and J. Wu, Eds., Springer Berlin Heidelberg, pp. 67–78.

[18] Bracha, G. Asynchronous Byzantine agreement protocols. *Information and Computation 75*, 2 (1987), 130–143.

[19] Bracha, G., and Toueg, S. Asynchronous Consensus and Broadcast Protocols. *JACM 32*, 4 (1985).

[20] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938* (2018).

[21] Cachin, C., Guerraoui, R., and Rodrigues, L. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[22] Cachin, C., and Poritz, J. A. Secure intrusion-tolerant replication on the internet. In *DSN* (2002).

[23] Cachin, C., and Tessaro, S. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)* (2005), IEEE, pp. 191–201.

[24] Cachin, C., and Vukolić, M. Blockchains consensus protocols in the wild. *arXiv preprint arXiv:1707.01873* (2017).

[25] Cahill, M. J., Röhm, U., and Fekete, A. D. Serializable isolation for snapshot databases. *ACM TODS 34*, 4 (2009), 20:1–20:42.

[26] Churyumov, A. Byteball: A decentralized system for storage and transfer of value. *https://byteball.org/Byteball.pdf* (2016).

[27] Clack, C. D., Bakshi, V. A., and Braine, L. Smart Contract Templates: foundations, design landscape and research directions. *CoRR abs/1608.00771* (2016).

[28] Clark, D. D. The Design Philosophy of the DARPA Internet Protocols. *ACM SIGCOMM Computer Communication Review 18*, 4 (1988), 106–114.

[29] Clement, A., Wong, E. L., Alvisi, L., Dahlin, M., and Marchetti, M. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI* (2009).

[30] Correia, M., Neves, N. F., Lung, L. C., and Veríssimo, P. Low complexity byzantine-resilient consensus. *Distributed Computing 17*, 3 (2005), 237–249.

[31] Crain, T., Gramoli, V., Larrea, M., and Raynal, M. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)* (2018), IEEE, pp. 1–8.

[32] Danezis, G., and Meiklejohn, S. Centrally banked cryptocurrencies. In *NDSS* (2016).

[33] Dinh, T. T. A., Wang, J., Chen, G., Liu, R., Ooi, B. C., and Tan, K.-L. BLOCKBENCH: A Framework for Analyzing Private Blockchains. In *SIGMOD* (New York, NY, USA, 2017), ACM, pp. 1085–1100.

[34] Duan, S., Reiter, M. K., and Zhang, H. BEAT: Asynchronous BFT Made Practical. In *CCS* (2018).

[35] Expert Panel (Forbes Technology Council). 10 Tech Industry Experts Predict The Next 'Blockchain Wave', Feb 13 2019.

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

[37] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling byzantine agreements for cryptocurrencies. In *SOSP* (2017).

[38] Glendenning, L., Beschastnikh, I., Krishnamurthy, A., and Anderson, T. Scalable Consistency in Scatter. In *SOSP* (2011).

[39] Golan-Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M. K., Seredinschi, D., Tamir, O., and Tomescu, A. SBFT: a scalable decentralized trust infrastructure for blockchains. In *DSN* (2019).

[40] Guerraoui, R., Hamza, J., Seredinschi, D.-A., and Vukolic, M. Can 100 Machines Agree? *arXiv:1911.07966* (2019). https://arxiv.org/abs/1911.07966.

[41] Guerraoui, R., Kermarrec, A.-M., Pavlovic, M., and Seredinschi, D.-A. Atum: Scalable group communication using volatile groups. In *Proceedings of the 17th International Middleware Conference* (New York, NY, USA, 2016), Middleware '16, ACM, pp. 19:1–19:14.

[42] Guerraoui, R., Komatovic, J., and Seredinschi, D.-A. Dynamic Byzantine Reliable Broadcast [Technical Report]. *arxiv:2001.06271* (2020). https://arxiv.org/abs/2001.06271.

[43] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., and Seredinschi, D.-A. AT2: Asynchronous Trustworthy Transfers. *arXiv/cs.DC 1812.10844* (2018). http://arxiv.org/abs/1812.10844.

[44] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., and Seredinschi, D.-A. Scalable Byzantine Reliable Broadcast. In *DISC* (2019).

[45] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovic, M., and Seredinschi, D.-A. The Consensus Number of a Cryptocurrency. In *PODC* (2019).

[46] Gupta, S. A Non-Consensus Based Decentralized Financial Transaction Processing Model with Support for Efficient Auditing. Master's thesis, Arizona State University, USA, 2016.

[47] Hearn, M. Corda: A distributed ledger. *Corda Technical White Paper* (2016).

[48] Herlihy, M. Wait-free synchronization. *ACM Trans. Program. Lang. Syst. 13*, 1 (1991), 123–149.

[49] Hunt, P., Konar, M., Junqueira, F. P., and Reed, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC* (2010).

[50] Karlsson, K., Jiang, W., Wicker, S., Adams, D., Ma, E., van Renesse, R., and Weatherspoon, H. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *ICDCS* (2018).

[51] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *IEEE S&P* (2018).

[52] Lamport, L. Brief announcement: Leaderless byzantine paxos. In *Distributed Computing* (Berlin, Heidelberg, 2011), D. Peleg, Ed., Springer Berlin Heidelberg, pp. 141–142.

[53] Lamport, L., Malkhi, D., and Zhou, L. Vertical paxos and primary-backup replication. In *PODC* (2009).

[54] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *TOPLAS 4*, 3 (1982).

[55] Lind, J., Naor, O., Eyal, I., Kelbert, F., Pietzuch, P., and Sirer, E. G. Teechain: A secure payment network with asynchronous blockchain access, 2017.

[56] Lloyd, W., Freedman, M. J., Kaminsky, M., and Andersen, D. G. Don't settle for eventual. In *SOSP* (2011).

[57] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 17–30.

[58] Malanov, A. Cryptocurrency Threat Predictions for 2019. KasperskyLab, November, 2018.

[59] Malkhi, D., Merritt, M., and Rodeh, O. Secure Reliable Multicast Protocols in a WAN. In *ICDCS* (1997).

[60] Malkhi, D., and Reiter, M. K. Byzantine quorum systems. In *STOC* (1997), vol. 97, pp. 569–578.

[61] Malkhi, D., and Reiter, M. K. A high-throughput secure reliable multicast protocol. *Journal of Computer Security 5*, 2 (1997), 113–128.

[62] Malkhi, D., Reiter, M. K., Wool, A., and Wright, R. N. Probabilistic quorum systems. *Inf. Comput. 170*, 2 (Nov. 2001), 184–206.

[63] McCorry, P., Möser, M., Shahandasti, S. F., and Hao, F. Towards bitcoin payment networks. In *Australasian Conference on Information Security and Privacy* (2016), Springer, pp. 57–76.

[64] Miller, A., Xia, Y., Croman, K., Shi, E., and Song, D. The Honey Badger of BFT Protocols. In *CCS* (2016).

[65] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 358–372.

[66] Nakamoto, S. Bitcoin: A peer-to-peer electronic cash system. *Whitepaper* (2008).

[67] Network, T. R. The raiden network. https://raiden.network/, 2017. [Online; accessed 9-September-2019].

[68] Poon, J., and Dryja, T. The bitcoin lightning network: Scalable off-chain instant payments, 2016.

[69] Reiter, M., and Birman, K. How to securely replicate services. *ACM Transactions on Programming Languages and Systems (TOPLAS) 16*, 3 (1994).

[70] Scheideler, C. How to Spread Adversarial Nodes? Rotate! In *STOC* (2005), ACM, pp. 704–713.

[71] Sciascia, D. LibPaxos3. https://bitbucket.org/sciascid/libpaxos. [Online; accessed 19-Sept-2018].

[72] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems*. Springer, 2011.

[73] Sompolinsky, Y., and Zohar, A. Accelerating Bitcoin's transaction processing: fast money grows on trees, not chains. *IACR Cryptology ePrint Archive, 2013:881* (2013).

[74] Sousa, J., Bessani, A., and Vukolic, M. A byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform. In *DSN* (2018).

[75] Sutra, P. On the correctness of egalitarian paxos. *CoRR abs/1906.10917* (2019).

[76] Van Renesse, R., and Schneider, F. B. Chain replication for supporting high throughput and availability. In *OSDI* (2004).

[77] VISA Inc. Visa fact sheet and quarter numbers. https://usa.visa.com/dam/VCOM/global/about-visa/documents/visa-fact-sheet-april-2019.pdf and https://s1.q4cdn.com/050606653/files/doc_financials/2019/Q3/Visa-Inc-Q3-2019-Operational-Performance-Data.pdf, 2019. [Online; accessed 9-September-2019].

[78] Vogelsteller, F., and Buterin, V. EIP 20: ERC-20 Token Standard, 2015. https://eips.ethereum.org/EIPS/eip-20.

[79] Vukolić, M. The Quest for Scalable Blockchain Fabric: Proof-of-work vs. BFT Replication. In *International Workshop on Open Problems in Network Security* (2015), Springer, pp. 112–125.

[80] Wang, J., and Wang, H. Monoxide: Scale out blockchains with asynchronous consensus zones. In *NSDI* (2019).

[81] Wood, G. Ethereum: A secure decentralized generalized transaction ledger. White paper, 2015.

[82] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G., and Abraham, I. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 2019), PODC '19, ACM, pp. 347–356.

[83] Yu, J., Kozhaya, D., Decouchant, J., and Esteves-Verissimo, P. RepuCoin: Your Reputation Is Your Power. *IEEE Transactions on Computers 68* (2018), 1225–1237.

[84] Zamani, M., Movahedi, M., and Raykova, M. Rapidchain: Scaling blockchain via full sharding. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2018), CCS '18, ACM, pp. 931–948.

Appendix A
Asynchronous Reconfiguration

The description of Astro is focused on a static system with a fixed set of replicas and clients, in order to clearly present its design. For long-lived systems, which we expect a payment system to be, adding and removing replicas is desirable, e.g., if participants decide to start/stop using the payment system, or when replacing old replica machines by new ones.

Reconfiguration of clients is straightforward: each client has a representative replica r, so adding a client simply means that r executes a BRB instance announcing the new client; subsequently all replicas start maintaining the *xlog* of this new client. Reconfiguration of system replicas is a more challenging problem, which we discuss in the rest of this section.

In consensus-based systems, reconfiguration can be handled by the consensus module. For instance, BFT-SMaRt [14] and similar systems [53] treat a reconfiguration request as a special request which is totally-ordered just like ordinary client requests.

Consensus, however, is not always necessary for reconfiguration. For example, DynaStore [4] and FreeStore [6] provide solutions for consensusless reconfiguration of read-/write storage in the asynchronous crash-stop model.

The purpose of this appendix is to briefly present a line of research that – we believe – answers in the affirmative the question of whether reconfiguration is possible for a payment system in the Byzantine model. The consequence is that our payment system does not require consensus throughout the entirety of its lifetime, which eradicates any possible argument supporting the necessity of consensus. In this line of research, we adopt ideas from the FreeStore [6] protocol, and to account for the Byzantine failure model we build on Byzantine quorum systems [60]. Admittedly, the details of reconfiguration are non-trivial and a thorough explanation of it is an independent publication. Our goal is to present a high-level overview of our ongoing result.

A. Overview

Throughout the lifetime of a system, each correct replica passes through a sequence of numbered *views*. A view is a set of replicas that a replica considers to constitute the system. At any point in time, each replica has exactly one *current view*.

The interface of the reconfiguration protocol exposes operations *Join*/*Leave*. Those operations consist of broadcasting a Join/Leave message to some view v, which represents the current state of the system as seen from the perspective of the joining/leaving replica.

Our reconfiguration protocol guarantees that, for a finite number of reconfiguration requests in any execution, all replicas converge to a single final view which incorporates every reconfiguration request issued in the execution. We

say that a view v is *installed* if some correct replica considers v its current view and processes payment operations in v. Moreover, our reconfiguration protocol ensures that the installed views form a sequence. Our state transfer protocol simply consists of sending all *xlogs* to the joining replica.

We adapt the payment protocol so that all messages include the current view of the sending replica. Correct replicas behave consistently across views with respect to each payment.

When a replica r observes a view that is more recent than r's current view, r pauses payment execution. Roughly speaking, r resumes execution after coordinating with a quorum of replicas belonging to the new view, and then r executes payments assuming the membership of the new view. Since reconfiguration is not a very frequent operation, we expect the overall downtime caused by reconfiguration to be insignificant. We evaluate the reconfiguration overhead of joining replicas in the next section.

B. Evaluation

The experiment of asynchronous reconfiguration starts with a system of $N = 4$ replicas; subsequently, new replicas join the system until $N = 80$, one by one. Note that our reconfiguration protocol allows batched joins (which we avoid so that we can measure the latency of the protocol itself), and that all replicas are randomly distributed across Europe (§VI-B). During this experiment, the system is quiescent, i.e. no client submits any payment.

We measure the latency (i.e. time to join) both for Astro II and BFT-SMaRt in Figure 8. In Astro II, latency represents the elapsed time between the moment when the joining replica sends the reconfiguration request until this replica becomes able to participate in the payment protocol. Latency in BFT-SMaRt represents the elapsed time between sending the special type of operation by the View Manager [15] and sending message to the joining replica that it can start participating in the protocol and should get up-to-date with the rest of system. The first data point for Astro II shows slightly higher latency than for subsequent points, which is due to the fixed overhead of establishing connections between replicas already in the system. As Figure 8 shows, latency in BFT-SMaRt is an order of magnitude higher than in Astro II. We are not aware of any published numbers on consensus-based reconfiguration latency; we believe that the primary reason for this difference in performance is owed to a simpler, more efficient protocol.

C. Dynamic Byzantine Reliable Broadcast

Dynamic Byzantine Reliable Broadcast (DBRB) represents the continuation of the work briefly introduced earlier (§A-A). An in-depth theoretical analysis of DBRB, along with a thorough proof of its correctness, is provided in [42].

Instead of using BRB based on Bracha's algorithm, Astro I can adopt DBRB as an underlying broadcast layer. Since DBRB provides the exact same properties (adapted to the dynamic environment) as those of the broadcast layer (including totality), no further modifications of Astro I are needed. Given that Astro II does not require the broadcast layer to provide totality, we can use a modified version of DBRB that does not provide totality (denoted with QDBRB).

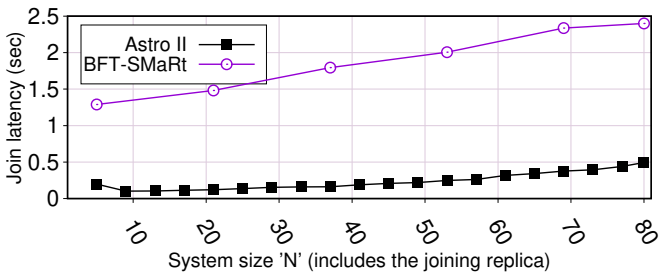

Fig. 8: Reconfiguration latency. Latency (seconds) of a join operation at different system sizes, in Astro II and in BFT-SMaRt.

The algorithm for QDBRB is obtained from the DBRB by excluding the last "all-to-all" [42, Section 4.3] communication step. With this modification, QDBRB becomes a direct replacement of the BRB within Astro II.

Appendix B
Broadcast Layer Algorithms

In Listing 5 we sketch the algorithm implementing BRB based on the work of Bracha and Toueg [19]. We use this algorithm to build the broadcast layer in Astro I.

Astro II uses a BRB implementation based on digital signatures [61], which we detail in Listing 6.

For completeness, we also provide pseudocode describing the use of dependencies, i.e., the optimization that allows Astro II to resolve the partial payments attack and enable sharding. To address these issues, the representative replica broadcasts a message a consisting of a payment together with the dependencies accumulated by the issuer of the payment since the last broadcast. More precisely, when a replica receives a payment from a client, it executes the steps outlined in Listing 7.

In the life-cycle of a payment, the dependencies of the spender (Alice) are materialized into balance in her replicated *xlog*, while the payment itself becomes a new dependency for the beneficiary (Bob). In particular, we obtain a full picture of the system by just re-defining the original approval and settling procedures of Listing 3 and 4 with Listing 8 and 9, respectively. Finally, Listing 10 shows how to handle the delivery of a proof at the representative replica of the beneficiary, which happens after a payment is settled.

```
// Called at replica 'r' to broadcast a message 'a'.
func Broadcast(a):
  prep := ⟨PREPARE, a⟩
  sendToAll(prep) // Send to all replicas.

// Process a protocol message m received from replica 'q' at
// replica 'r'.
callback receive(q, m = ⟨PREPARE, a⟩):
// Handler for PREPARE messages.
  let a be {s, ts, _, _}
  if echoSent[q, (s, ts)] == false:
    echoSent[q, (s, ts)] := true
    sendToAll(⟨ECHO, q, (s, ts), a⟩)

callback receive(q, m = ⟨ECHO, r, (s, ts), a⟩):
// Handler for ECHO messages.
  echoes[r, (s, ts), a] += q
  if (|echoes[r, (s, ts), a]| ≥ 2F+1) &&
      (readySent[r, (s, ts), a] == false):
    readySent[r, (s, ts)] := true
    sendToAll(⟨READY, r, (s, ts), a⟩)

callback receive(q, m = ⟨READY, r, (s, ts), a⟩):
// Handler for READY messages.
  readys[r, (s, ts), a] += q
  if (|readys[r, (s, ts), a]| ≥ F+1) &&
       (readySent[r, (s, ts), a] == false):
    readySent[r, (s, ts), a] := true
    sendToAll(⟨READY, r, (s, ts), a⟩)
  if (|readys[r, (s, ts), a]| ≥ 2F+1) &&
       (delivered[r, (s, ts), a] == false) &&
       (ts == allTS[s] + 1):
    delivered[r, (s, ts), a] := true
    trigger Deliver(a)
    allTS[s] += 1
```

Listing 5: BRB protocol which we use in Astro I, based on [19].

```
// Called at replica 'r' to broadcast a message 'a'.
func Broadcast(a)
  prep := ⟨PREPARE, a⟩
  // Send the prepare message to all replicas.
  sendToAll(prep)

// Process a protocol message m received from replica 'q' at
// replica 'r'.
callback receive(q, m)
  if (m = ⟨PREPARE, a⟩)
    let a be {s, ts, _, _}
    pending[(s, ts)] := a
    sig := Sign(m)
    ackMsg := ⟨ACK, (s, ts), sig⟩
    unicast(q, ackMsg) // Reply to replica q with ACK.
  else if (m = ⟨ACK, (s, ts), sig⟩)
    return if invalidSignature(sig)
    acks[(s, ts)] := acks[(s, ts)] ∪ {(q, sig)}
    if (|acks[(s, ts)]| == 2f+1)
      commitMsg = ⟨COMMIT, (s, ts), acks[(s, ts)]⟩
      sendToAll(commitMsg) // Broadcast the commit
     message.
  else if (m = ⟨COMMIT, (s, ts), proof⟩)
    return if (|proof| < 2f+1) || (invalidSignatures(
     proof))
    a := pending[(s, ts)]
    // Release payment 'a' to the payment layer
    trigger Deliver(a)
```

Listing 6: BRB protocol based on digital signatures, inspired by early work of Malkhi and Reiter [61], which we use in Astro II.

```
@executes at the representative replica
@local state: DepMap deps[..] //dependencies per client

callback receive(a):
  let a be ⟨Alice, n, b, x⟩
  Broadcast(⟨Alice, n, b, x, deps[Alice]⟩)
  deps[Alice] := {}
```

Listing 7: Using dependencies in Astro II. Representative replica broadcasts payment with dependencies.

```
func approve(a)
  let a be ⟨Alice, n, _, x, dependencies⟩
  wait until sn[Alice] = n - 1
```

Listing 8: Payment approval for BRB of Listing 6. Every replica executes this to approve a payment *a*, assuming spender Alice.

```
@executes at all system replicas
// Used dependencies per client
@local state: DepMap usedDeps[..]

func settle(a)
  let a be ⟨Alice, n, b, x, dependencies⟩

  // Keep only the never seen before dependencies
  newDeps = set(dependencies) \ usedDeps[Alice]
  usedDeps[Alice] = usedDeps[Alice] ∪ newDeps

  bal[Alice] += balanceOf(newDeps) // Credit balance
  if bal[Alice] < x: return

  bal[Alice] -= x // Withdraw from Alice's balance
  sn[Alice] += 1
  xlogs[Alice].append(a)

  d = (Alice, n, b, x)
  // Send proof to Bob's representative (Credit message)
  trigger unicast(b, (d, Sign(d)))
```

Listing 9: Payment settling procedure for BRB of Listing 6. Each replica executes this protocol to transition a payment *a* to the final, settled state.

```
@executes at the representative replica
@local state: DepMap deps[..] // dependencies per client
              DepMap partialDeps[..]

callback DeliverUnicast(proof)
  let proof be ⟨payment, sig⟩
  let payment be ⟨Alice, n, b, x⟩

  if !check(proof, payment):
    return
  partialDeps[payment].add(proof)

  // An incoming payment that collects f+1 proofs becomes
  // a dependency.
  if len(partialDeps[payment]) = f + 1:
    deps[Alice].add(partialDeps[payment])
    delete(partialDeps[payment])
```

Listing 10: Handling of dependencies for BRB of Listing 6. The representative replica executes this protocol every time a proof of an incoming payment is received.