Bigger, not Badder: Safely Scaling BFT Protocols

David C. Y. Chu, Chris Liu, Natacha Crooks, Joseph M. Hellerstein, Heidi Howard[†]
UC Berkeley, Microsoft[†]

Abstract

Byzantine Fault Tolerant (BFT) protocols provide powerful guarantees in the presence of arbitrary machine failures, yet they do not scale. The process of creating new, scalable BFT protocols requires expert analysis and is often error-prone. Recent work suggests that localized, rule-driven rewrites can be mechanically applied to scale existing (non-BFT) protocols, including Paxos. We modify these rewrites—decoupling and partitioning—so they can be safely applied to BFT protocols, and apply these rewrites to the critical path of PBFT, improving its throughput by 5×. We prove the correctness of the modified rewrites on *any* BFT protocol by formally modeling the arbitrary logic of a Byzantine node. We define the Borgesian simulator, a theoretical node that simulates a Byzantine node through randomness, and show that in any BFT protocol, the messages that a Borgesian simulator can generate before and after optimization is the same. Our initial results point the way towards an automatic optimizer for BFT protocols.

1 Introduction

Dealing with arbitrary failures is inherently complex; dealing *efficiently* with arbitrary failures even more so. Designing correct, scalable BFT [9] protocols is thus extremely challenging, and even experts often make mistakes [2, 10]. While one cannot do much about the inherent complexity of BFT, recent work suggests an appealing middle ground. Instead of creating new, scalable protocols from scratch, one can break down existing (and usually simpler) protocols into components [7] that can be scaled individually [15].

Gupta et al. [8] manually pipelined and partitioned existing BFT protocols, achieving up to 6× throughput improvement. Chu et al. [5, 6] obtained similar results with simple, local, *program rewrites* for Paxos [14]: **decoupling** (splitting logic across nodes) and **partitioning** (splitting data across nodes), as seen in Figure 1. Their rewrites are promising because they are protocol-agnostic and rule-driven: given a protocol written in a distributed DSL [3], the rewrites can be used

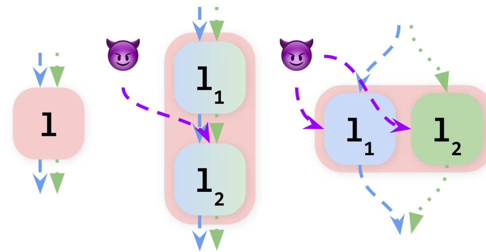

Figure 1. Decoupling (middle) and partitioning (right), splitting a node l into two nodes l_1 and l_2. Byzantine nodes (evil emoji) can insert messages on decoupled message channels and duplicate messages across partitions.

to correctly modify any protocol. However, these rewrites were proven correct assuming non-Byzantine failures only.

This paper modifies the rewrites introduced by Chu et al. and proves the correctness of the resulting rewrites when applied to BFT protocols.

To illustrate the challenge, consider the critical path of PBFT [4], a fundamental BFT protocol that reaches consensus across $3f+1$ replicas. Replicas in PBFT receive PrePrepare messages from a primary replica: these messages contain, among other things, a command to execute, its hash digest, and a signature from the primary over the hash digest. The replica accepts the message if the digest is indeed the hash of the command and the signature is valid. In a later phase, once this replica receives $2f+1$ Commit messages (which also contain a digest) from other replicas, it will compare the digest within the Commits to the digest of the PrePrepare it received earlier; if the digests match, then the replica will execute the command.

Because the replica monotonically accumulates sets of PrePrepare and Commit messages through time, the rewrites from Chu et al. suggest that replicas can be scaled up via monotonic decoupling. For each node, the logic that collects PrePrepare messages and the logic that collects Commit messages can be decoupled into two nodes: a *pre-preparer* and *committer* . Crucially, each pre-preparer must forward PrePrepares to its own committer so it can later compare digests and execute the command. The remaining phases of PBFT can be similarly decoupled, as we will discuss in detail in Section 6.

Intuitively, decoupling logic into two nodes reduces load on any one machine and can improve throughput. However, in the presence of Byzantine failures, this decoupling would be *unsafe*. A single Byzantine node could send two committers doctored PrePrepares with the same digest but different commands. Committers, without checking whether

This work is licensed under a Creative Commons Attribution International 4.0 License.
PaPoC '24, April 22, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0544-1/24/04
https://doi.org/10.1145/3642976.3653033

the message was forwarded from their own pre-preparers, would then execute the wrong commands, breaking the consensus invariant. Because Chu et al.'s decoupling rewrite was not designed to handle Byzantine attacks, the messages between decoupled nodes are neither signed nor verified, allowing Byzantine nodes to insert their own messages as seen in Figure 1.

To prevent the attack above, we will modify decoupling with *sender verification* (Section 4.1), which signs and verifies each message sent between decoupled nodes. We will also modify partitioning with *message verification* (Section 4.2), such that each partition individually verifies that it is receiving the correct subset of hash-partitioned messages.

Given an input protocol that is BFT, our proof strategy is two-fold: we show that (1) to nodes untouched by the rewrites, Byzantine nodes in the rewritten protocol cannot generate any more messages than they could have generated before, and (2) for modified nodes, all invariants required by the rewrites are reinforced against Byzantine attacks. With these guarantees, any Byzantine attacks on the untouched nodes in the rewritten protocol would be identical to a Byzantine attack in original protocol (which is already BFT), and any Byzantine attacks on the modified nodes are ineffective.

In order to discuss what messages a Byzantine node can produce, we must formally model *all* possible Byzantine behavior as part of our proof framework. This is tricky: Byzantine behavior is, by definition, arbitrary. To this effect, we make the following observations: (1) a Byzantine node's behavior is a function of its outputs, and (2) the set of all possible Byzantine behaviors is exactly the set of all *random* behaviors. We introduce the *Borgesian simulator*[1], a theoretical gadget that simulates all possible Byzantine behavior in a protocol-agnostic way through randomness. At any point in time, the Borgesian simulator can generate arbitrarily many random messages to all input channels. This theoretical gadget allows us to compare Byzantine behaviors and prove the correctness of the modified rewrites.

Our initial results are promising. The modified rewrites are simple to check, yet applying them scales the throughput of PBFT by 5× when applied to its critical path.

2 Background

BFT. BFT protocols can tolerate up to f node failures, where failed nodes can exhibit Byzantine behavior. Byzantine nodes can send arbitrary messages and share private keys. They cannot, however, break standard cryptographic primitives [4]. We assume a shared-nothing architecture in which nodes can only communicate through messages. Messages between correct nodes will eventually be delivered.

[1]Jorge Luis Borges' story *The Library of Babel* posits a library of books composed of every possible ordering of characters. By definition, this library contains all books, though it may take arbitrary time to find good ones.

Rule-driven rewrites. Chu et al. presents a set of rule-driven rewrites that can be applied to arbitrary distributed protocols implemented in Dedalus [3], a declarative dataflow language for distributed systems based on Datalog. Due to limited space, we will instead discuss these rewrites in the context of event-driven pseudocode instead.

Rewrites improve protocol throughput by *decoupling* and *partitioning* [15] (Figure 1). Decoupling splits *logic* on a single node l into two nodes l_1 and l_2. Partitioning splits *data* on a single node l into multiple new nodes $\bar{l} = \{l_1, l_2, \ldots\}$. Both rewrites alleviate single-node bottlenecks by introducing pipeline- and data- parallelism, respectively [5]. We say that these new nodes $l_1, l_2, \ldots$ **correspond** to l.

Used naively, decoupling and partitioning can make a protocol that was previously fault tolerant vulnerable to Byzantine attacks. The rewrites can (1) create new nodes and introduce message channels between them, in order to enable pipelining or coordination, and (2) modify existing message channels so messages to a single node must be forwarded to its correct corresponding nodes. New message channels present exploitable opportunities to Byzantine nodes, which can send arbitrary messages to those channels.

2.1 Running example and syntax

We will use Algorithm 1 as a case study in our rewrites.

This snippet of PBFT [4] collects Prepare messages, waits for a quorum, then broadcasts a Commit message, representing a common pattern that can be found in many other BFT protocols. Prepare messages arrive on the `prepareIn` channel and Commit messages are sent on the `commitOut` channel. We will use the syntax `commitOut@dest` ⇜ m to indicate sending a message m to node `dest` on channel `commitOut`. The address of each node is stored in the variable `self`, and the addresses of all nodes is stored in `allNodes`.

Each prepare message has the following fields: view v, slot number n, digest d of a command, sender index i, and signature σ. Each Prepare message is first filtered based on its signature and view (Line 4), then added to the log (Line 5). Once $2f + 1$ Prepare messages have been received for a particular slot number and digest (Line 7), a Commit message is signed and broadcasted to all other nodes (Lines 9 and 10).

Nodes have access to the cryptographic functions `sign(sk, m)` and `verify(pk, σ, m)`. `sign(sk, m)` returns the signature σ for a secret key sk and message m. `verify(pk, σ, m)` returns true if σ is the signature of the secret key sk corresponding to the public key pk over m; it returns false otherwise. Keys are available to each node through the maps `skeys` and `pkeys`; for each node at location l, `skeys[`l'`]` and `pkeys[`l'`]` returns the secret and public keys used to sign and verify messages from l to l', respectively.

We treat the signature schemes commonly used by BFT protocols—MACs (Message Authentication Codes) [11] and

public-key signatures [13]—as black boxes and do not distinguish between them in our formalism. Both produce a signature that can be verified with the correct key; this key is symmetric for MACs and asymmetric for public-key signatures.

In our formalism, we will store all keys, symmetric or asymmetric, in `skeys` and `pkeys`, and sign and verify them with the same syntax. Any symmetric key k used between nodes l and l' is *only* available to those two nodes: k = `skeys[`l'`]` = `pkeys[`l'`]` on l, and k = `skeys[`l`]` = `pkeys[`l`]` on l'. The availability of asymmetric keys is asymmetric. A node l signs all its outgoing messages with sk, which is known to no other node, but all nodes know its public key pk: for any l', sk = `skeys[`l'`]` on l, and on any l', pk = `pkeys[`l`]`.

Algorithm 1: The running example from PBFT.

```
1  prepareLog = {}
2  numPrepares[][] = {}
3  for msg ∈ prepareIn(v, n, d, i, σ) do
4      if verify(pkeys[i], σ, <v, n, d>) and v = view then
5          prepareLog.add(msg)
6          numPrepares[n][d] += 1
7          if numPrepares[n][d] ≥ 2f+1 then
8              for dest ∈ allNodes do
9                  σ′ ← sign(skeys[dest], <v, n, d, self>)
10                 commitOut@dest ⇜ (v, n, d, self, σ′)
```

2.2 Correctness

This paper is focused on proving the correctness of *rewrites* across BFT protocols, not the correctness of any specific BFT protocol. Assuming that the original distributed protocol is correct in the presence of f Byzantine failures, the modified rewrites will decouple and partition the protocol while preserving correctness up to f Byzantine failures.

We discuss failures in the rewritten protocol in terms of *fault domains*. Each node l in the original protocol belongs to its own fault domain; when l is scaled up into its corresponding nodes $\bar{l}$, all nodes in $\bar{l}$ remain in the same fault domain. Therefore, the failure of *any* node in $\bar{l}$ is equivalent to the failures of *all* nodes in $\bar{l}$, which is equivalent to the failure l, and the rewrites do not change the number of fault domains.

Intuitively, to any observer, the output of all nodes $\bar{l}$ together represents the output of the corresponding original node l. A "partially" Byzantine node is still Byzantine: if any single node $l_i \in \bar{l}$ becomes Byzantine while the remaining corresponding nodes are still correct, it is as if the original node l becomes Byzantine but continues to send *some* correct messages.

Correctness is defined by observable program behavior. A rewrite is correct if given any program P, a rewritten program P', and any set of inputs (and their respective send times), P' always generates the same outputs with the same timestamps as some possible run of P with up to f Byzantine failures [5].

3 Borgesian simulators

To prove the correctness of rewrites in a Byzantine setting, we must first formally model a Byzantine node in a protocol-agnostic way.

This is tricky; a Byzantine node can execute arbitrary logic. Fortunately, other nodes only observe the output of the node, not the logic by which it generated a particular message. Two Byzantine nodes that generate the same outputs are indistinguishable, even if one executes complex logic and the other creates messages at random. It follows that a Byzantine node can be modeled solely by its outputs.

Byzantine behavior can thus be fully captured by a *random message generator*: if, at each point in time, a node sends a random number of random messages to each channel, then that node must contain, in its set of possible runs, *exactly all* possible runs of any Byzantine node. We name this random message generator the **Borgesian simulator**. This simulator can generate both nonsensical and seemingly intelligent outputs, all of which must be correctly handled by a BFT protocol. We will formalize the Borgesian simulator by attaching a **Borgesian harness** to all nodes in a BFT protocol. Note that this harness defines what a protocol must defend against and is *not* meant to be applied in practice or used as a fuzz tester.

3.1 Plaintext message channels

In the simplest case, given a plaintext message channel on any node, a node simulating Byzantine behavior should be able to send an arbitrary number of messages to that channel.

Algorithm 2 is our running example with this harness applied: it executes the original logic if it is not Byzantine, and sends a random number of random messages to `commitOut` at each moment in time if it is. In our pseudocode, `isByzantine` is a mapping from nodes to booleans that returns `true` for up to f nodes, and `randInt()` and `rand()` provide random values.

Algorithm 2: The running example with the plaintext Borgesian harness.

```
1  if !isByzantine[self] then
       // Original logic
2  else
3      for t ∈ (0,∞), dest ∈ allNodes do
4          for i ∈ (0, randInt()) do
5              commitOut@dest ⇜ (rand(), rand(), rand(),
                 rand(), rand())
```

This harness can be systematically applied to any node in any protocol by iterating over all message channels on Line 3, replacing `commitOut` with each channel on Line 5, and populating message fields with random values.

Algorithm 3: The running example with the signed message channel Borgesian harness.

```
1 if !isByzantine[self] then
    // Original logic
2 else
3   for t ∈ (0,∞), dest ∈ allNodes, byz ∈ allNodes where
      isByzantine[byz] do
4     for i ∈ (0, randInt()) do
5       <v, n, d, s> ← (rand(), rand(), rand(), rand())
6       σ ← sign(allSkeys[byz][dest], <v, n, d, s>)
7       commitOut@dest ↜ (v, n, d, s, σ)
```

Observe that a Byzantine node with the Borgesian harness can also behave like a non-Byzantine node; the set of all possible runs includes runs in which Byzantine nodes do not deviate from the protocol.

3.2 Signed message channels

Our Borgesian harness up to this point is flawed: by using `rand()` to populate the "signature" field in Algorithm 2 Line 5, the Borgesian simulator is allowed to break cryptography, mimicking the signatures of keys it does not have access to. To model practical assumptions, we would like to forbid the Borgesian simulator from randomly emitting messages that break cryptography. To this end, we separate message fields into three types: (1) unprotected fields, (2) protected fields, and (3) signatures created by signing over the protected fields. A Borgesian simulator should only generate signatures by signing the protected fields with either its own keys or the keys of other Byzantine nodes, simulating collusion.

Algorithm 3 is our running example, with the new Borgesian harness that generates the signature of messages on `commitOut` using the keys available to it. Note that all fields of `commitOut` are protected. In the psuedocode, `allSkeys` is a mapping from nodes to their `skeys` array, which allows a Byzantine node to access the keys of all other Byzantine nodes. In general, the Borgesian harness must differentiate between plaintext and signed message channels: on signed channels, it must generate a valid signature by using the keys it has to sign the messages' protected fields.

The Borgesian simulator does not need to generate messages with incorrect signatures in our formalism, because we assume that all correct nodes verify the messages they receive on signed channels, so any incorrectly signed messages will be discarded.

3.3 Forwarding

Although Byzantine nodes can only sign new messages using their keys, they can still *forward* along messages signed by other nodes as long as they do not alter the protected fields. We will modify the Borgesian harness such that it takes previously received messages into consideration.

Algorithm 4 is our running example, with the new Borgesian harness which in addition to signing messages with its own key, can store received messages in `commitStore` and output them randomly.

Algorithm 4: The running example with the forwarding Borgesian harness.

```
1  if !isByzantine[self] then
     // Original logic
2  else
3    for msg ∈ commitIn(v, n, d, s, σ) do
4      commitStore.add(msg)
5    for t ∈ (0,∞), dest ∈ allNodes do
6      for byz ∈ allNodes where isByzantine[byz] do
7        for i ∈ (0, randInt()) do
8          <v, n, d, s> ← (rand(), rand(), rand(), rand())
9          σ ← sign(allSkeys[byz][dest], <v, n, d, s>)
10         commitOut@dest ↜ (v, n, d, s, σ)
11     for msg ∈ commitStore do
12       for i ∈ (0, randInt()) do
13         commitOut@dest ↜ msg
```

3.4 Nested types

For more complex data types (such as View-Change from PBFT, which contains arrays of sets of messages) the Borgesian harness should populate fields with a combination of the approaches described above. It should populate each data structure with a random number of elements, generating random data for each unprotected field and either signing or reusing signatures for protected fields.

Now the Borgesian harness is complete; any node for which `isByzantine` is true becomes the Borgesian simulator, whose possible behaviors are exactly the set of possible behaviors of a Byzantine node.

4 Modifications to rewrites

We now present two modifications to the rewrites introduced by Chu et al.: sender verification (Section 4.1) and message verification (Section 4.2).

4.1 Sender verification

The rewrites introduce new channels, with implicit assumptions about which nodes will send on those channels.[2] Without additional verification, these channels would be vulnerable to Byzantine attacks from other nodes.

Modification: Sender verification. Replace each new channel `chan(m)` from node l_1 to l_2 with `signed_chan(m, σ)`, where σ = `sign(skeys[`l_2`], m)`. The receiving node l_2

[2] Rewrites that fall into this category include monotonic decoupling, functional decoupling, asymmetric decoupling, and partial partitioning [5, 6].

only processes a message m if `verify(pkeys[`l_1`], σ, m)` returns true.

As an optimization, if functional dependencies [1, 5] are known, then the sender does not have to sign all fields of the message. If there is a functional dependency from message field x to y—that is, there is some deterministic function f where $f(x) = y$—then signing over x is sufficient. If there is a one-to-one dependency between the fields—there exists some $f^{-1}(y) = x$—then signing over *either* field is sufficient. This optimization applies to any BFT protocol that uses collision-resistant hash functions (such as PBFT), where there is a one-to-one dependency between the hash digest and the original message. In that case, only the digest has to be signed. PBFT itself makes the same observation and only signs the digest on its preexisting channels. Note that the inverse of the hash function does not need to be known; this optimization is valid as long as such a function *exists*.

4.2 Message verification

The partitioning rewrites also introduce channels with implicit assumptions on message content. Nodes created through partitioning all share the same types of input channels, but each partition assumes that it will receive a specific disjoint subset of the messages.[3] Byzantine nodes can exploit this assumption and send messages to the wrong nodes.

Modification: Message verification. For each input channel `chan(m)` on partition l_i, replace all instances of `chan(m)` with `correct_chan(m)`, which drops any messages from `chan(m)` that are not meant for this partition. This criteria can be checked with the distribution policy function $D(m)$ as defined in [5], which maps each message to its intended partition.

5 Proof of correctness

We now prove the correctness of the modified rewrites in the presence of Byzantine failures. The rewrites are correct if the number of Byzantine fault domains remains unchanged (as discussed in Section 2.2), and all input channels on correct nodes can correctly handle messages from Byzantine nodes.

Without loss of generality, we will prove this holds over a specific correct node $dest_c$ that receives messages from a specific Borgesian simulator sender, sim_b in the original protocol and sim'_b in the rewritten protocol.

We will break down the input channels of $dest_c$ into four types—unmodified, redirected, duplicated, and new channels—and prove that each type can correctly handle messages from sim'_b after the rewrites. Unmodified channels are unchanged between the original and rewritten protocols. Redirected channels existed on some node l in the original protocol and exist on one of its corresponding nodes in the rewritten protocol. Duplicated channels existed on l in the original protocol and on multiple corresponding $\bar{l}$ in the rewritten protocol. New channels are introduced by the rewrites between between l_1 and l_2 nodes that correspond to the same original node. All input channels in a rewritten protocol can be categorized into one of the four types above.

[3]All subcategories of partitioning in [5]—partitioning with co-hashing, partitioning with dependencies, and partial partitioning—share this assumption.

Unmodified channels. These channels existed on $dest_c$ in the original protocol and remain on $dest_c$ after the rewrites. Because we assumed that the original protocol is already BFT, any messages sent by sim_b must already be correctly handled. We will show that sim'_b in the rewritten protocol cannot generate additional messages by reasoning with the Borgesian simulator.

Because the channel is unmodified, by construction, the output logic of the sim'_b to that channel must also be unchanged from sim_b. However, we might still be concerned that the *state* at sim'_b could change after the rewrites, by receiving more signed messages from some other correct node n_c. Then sim'_b would then be able to forward along more signed messages than sim_b (Section 3.3). We will show that this concern is unjustified: sim'_b cannot receive more messages than sim_b, so it cannot send more either.

To assess this, we need to examine the input channels to sim'_b through which these messages arrived. If these messages were sent by another Byzantine node, then they could have been generated by sim'_b as well; therefore, we will only analyze situations where the sender n_c is correct. *Unmodified input channels*: by definition of "unmodified", any n_c that sends a message to an unmodified channel in sim'_b would have sent the same message to sim_b, so the state is unchanged. *Redirected or duplicated input channels*: the rewrites guarantee that a n_c will only send messages to a node sim'_b if, in the original protocol, it sent the same messages to the corresponding node sim_b. Again, the state is unchanged. *New input channels*: the rewrites only introduce new channels between nodes corresponding to the same original node. Therefore, nodes connected by new channels must be in the same fault domain. Any node that can send a message on a new channel to a Byzantine node sim'_b must therefore also be Byzantine, breaking our assumption that the sender n_c is correct.

Because the Borgesian simulator in the rewritten protocol cannot *receive* new messages from correct nodes on any of its input channels, it cannot *produce* any new messages either.

Redirected channels. A redirected channel is essentially an unmodified channel where senders redirect all messages from an original node l to a specific corresponding node l_i. The proof is similar; any message sent to l_i after rewrites could have been sent to l before rewrites.

Duplicated channels. Partitioning duplicates a channel on l across its corresponding partitions $\bar{l} = \{l_1, l_2, ...\}$. Each channel expects to get a disjoint subset of messages, depending on the distribution policy [5]. This new invariant

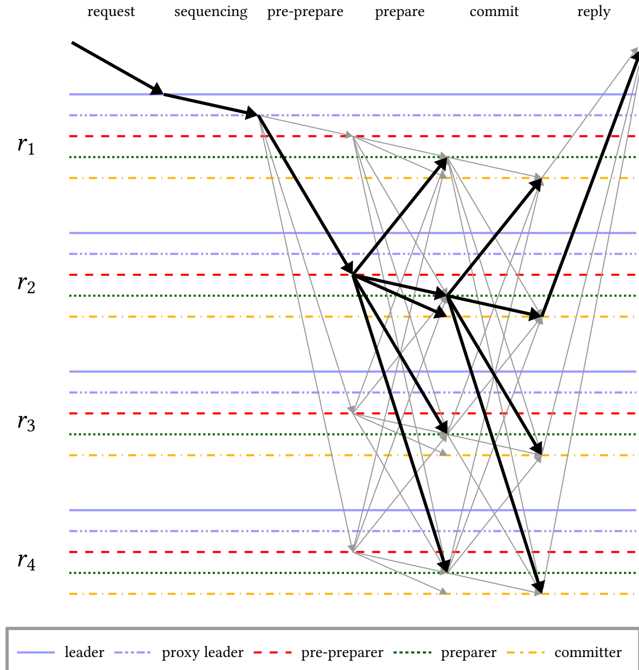

Figure 2. The critical path of ScalablePBFT, with the message path through r_2 bolded. r_1 is the primary.

is enforced by message verification (Section 4.2); the channel is otherwise unmodified and the proof is similar.

New channels. These channels have no counterpart in the original protocol, so we must show that they can correctly handle all Byzantine messages. As stated above, new channels are only introduced between nodes in the same fault domain. Since $dest_c$ is correct, all nodes in its fault domain must be correct. Sender verification (Section 4.1) filters messages from nodes outside the fault domain; therefore, $dest_c$ will not process any Byzantine messages on a new channel.

We have demonstrated that all input channels on a correct node $dest_c$ after rewriting are safe against Byzantine attacks, proving that the rewrites correctly preserve fault tolerance.

6 Initial results

We evaluate the efficacy of our rule-driven rewrites by manually applying them to scale the critical path of PBFT [4]. Our experimental scripts and setup can be found at https://github.com/rithvikp/autocomp.

Our evaluation mirrors the evaluation from Chu et al. PBFT is implemented in Dedalus [3] and compiled to Hydroflow [12], a Rust dataflow runtime for distributed systems. We deploy on GCP using n2-standard-4 machines with 4 vCPUs, 16 GB RAM, and 10 Gbps network bandwidth. Throughput and latency are measured over one minute runs after 30 seconds of warmup. Clients send 16 byte unbatched commands in a closed loop. The state machines, clients, and protocol nodes are all run on separate machines. Performance is measured with an increasing set of clients

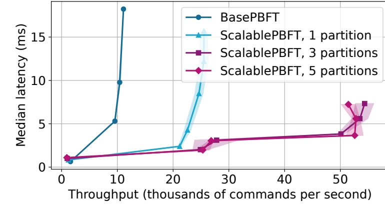

Figure 3. Throughput/latency comparison of PBFT before and after rewrites.

until throughput saturates, averaging across 3 runs with standard deviations in shaded regions.

We will refer to the unoptimized implementation of PBFT as BasePBFT and the rewritten implementation as ScalablePBFT. Both implementations only include the critical path and assume that there is no view-change and no checkpoints. Our deployments tolerate $f = 1$ failures. Figure 3 compares the resulting throughput-latency graph from both implementations.

BasePBFT. The base deployment contains $3f + 1 = 4$ nodes. Clients send messages to a pre-elected primary, which sequences the command and broadcasts PrePrepares (including both the command and its signed digest). Replicas that receive PrePrepares broadcast Prepare (now only including the command's digest). Replicas that receive $2f+1$ Prepares broadcast a Commit. Replicas that receive $2f+1$ Commits find their matching PrePrepare from earlier and send the command and its sequence number to its state machine, which executes the command and notifies the client. The client waits for $f+1$ state machine results before sending the next message. BasePBFT achieves a maximum throughput of 11,000 commands/s.

ScalablePBFT. We created ScalablePBFT through *mutually independent decoupling, functional decoupling, monotonic decoupling, and partitioning with co-hashing*. Each replica is decoupled into five components: the leader, proxy leader, pre-preparer, preparer, and committer. Each component, aside from the leader, is hash-partitioned on the commands' sequence number.

Figure 2 illustrates the roles of the individual components; partitioning is implicit. The leader listens to the client and sends a PrePrepare to proxy leaders. Proxy leaders broadcast PrePrepares to pre-preparers. Pre-preparers broadcasts a Prepare to preparers upon receiving PrePrepare and also sends a PrePrepare to its corresponding committer. Preparers broadcast a Commit to committers upon receiving $2f + 1$ Prepares. Committers send the command and sequence number to its state machine after receiving $2f+1$ Commits and the corresponding PrePrepare.

We evaluate ScalablePBFT on 1, 3, and 5 partition configurations, where each partitionable component is partitioned n ways. The 3-partition configuration of ScalablePBFT—with 4 leaders, 12 proxy leaders (only 1 leader and its 3 proxy leaders are active), 12 pre-preparers, 12 preparers, and 12

committers—achieves a maximum throughput of 55,000 commands/s, a 5× improvement. The additional latency overhead from scaling is negligible. In our experiments, scaling beyond 5 partitions did not significantly improve throughput.

In order to validate our results, we aim to evaluate these rewrites on the entirety of PBFT and on more BFT protocols. The attentive reader might notice that decoupling PBFT becomes complex when logic *outside* the critical path is considered; a view-change, for example, must update all decoupled components. To this end, we will introduce *partial decoupling*, which adds a round of coordination on the off chance that values "shared" between components are updated, much like partial partitioning [5]. This rewrite is not yet in the literature and will be formalized in future work.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/pdfs/all.pdf

[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Ramakrishna Kotla, and Jean-Philippe Martin. 2017. Revisiting Fast Practical Byzantine Fault Tolerance. *CoRR* abs/1712.01367 (2017). arXiv:1712.01367 http://arxiv.org/abs/1712.01367

[3] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. 2011. Dedalus: Datalog in Time and Space. In *Datalog Reloaded*, Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 262–281.

[4] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OSDI*, Vol. 99. 173–186.

[5] David C. Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites. *Proc. ACM Manag. Data 2, N1 (SIGMOD)*, Article 2 (Feb. 2024), 25 pages. https://doi.org/10.1145/3639257

[6] David C. Y. Chu, Rithvik Panchapakesan, Shadaj Laddad, Lucky E. Katahanas, Chris Liu, Kaushik Shivakumar, Natacha Crooks, Joseph M. Hellerstein, and Heidi Howard. 2024. Optimizing Distributed Protocols with Query Rewrites [Technical Report]. https://github.com/rithvikp/autocomp.

[7] Suyash Gupta, Mohammad Javad Amiri, and Mohammad Sadoghi. 2023. Chemistry behind Agreement. In *Conference on Innovative Data Systems Research (CIDR).(2023)*.

[8] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. 2020. Resilientdb: Global scale resilient blockchain fabric. *arXiv preprint arXiv:2002.00160* (2020).

[9] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.* 4, 3 (jul 1982), 382–401. https://doi.org/10.1145/357172.357176

[10] George Pirlea. 2023. Errors found in distributed protocols. https://github.com/dranov/protocol-bugs-list.

[11] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM* 21, 2 (1978), 120–126.

[12] Mingwei Samuel, Joseph M Hellerstein, and Alvin Cheung. 2021. Hydroflow: A Model and Runtime for Distributed Systems Programming. (2021).

[13] Gene Tsudik. 1992. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review* 22, 5 (1992), 29–38.

[14] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (Feb. 2015), 36 pages. https://doi.org/10.1145/2673577

[15] Michael Whittaker, Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, Neil Giridharan, Joseph M. Hellerstein, Heidi Howard, Ion Stoica, and Adriana Szekeres. 2021. Scaling Replicated State Machines with Compartmentalization. *Proc. VLDB Endow.* 14, 11 (July 2021), 2203–2215. https://doi.org/10.14778/3476249.3476273