EVIL SERVERLESS CONSENSUS

Chris Liu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-138
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-138.html

May 17, 2024

Copyright © 2024, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

To Natacha Crooks, for advising me throughout my research. To David Chu, for his constant support, encouragement, and advice—and for introducing me to the joys of distributed systems. To Joseph M. Hellerstein, Shadaj Laddad, and other members of the wonderful Hydro team for their additional support. This work was supported by gifts from AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

EVIL SERVERLESS CONSENSUS

by Chris Liu

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

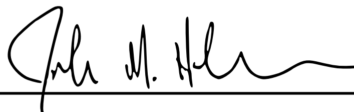
Committee:

Professor Natacha Crooks
Research Advisor

May 17 2024

(Date)

* * * * * * *



Professor Joseph M. Hellerstein
Second Reader

May 16 2024

(Date)

EVIL SERVERLESS CONSENSUS

ABSTRACT

Byzantine fault tolerant protocols provide powerful tools to cope with the presence of arbitrary failures, but have historically struggled to do so at scale. The creation of new, scalable BFT protocols regularly hinges on the unearthing of novel insights, the process of which is often error-prone. Alternatively, localized, rule-driven rewrites have shown promise in recent work through their ability to mechanically scale existing CFT protocols like Paxos using decoupling and partitioning techniques. These techniques are powerful, but are unable to be safely applied to a BFT environment as-is. We introduce modifications to these rewrites such that they can be safely applied to BFT protocols, and additionally prove the correctness of our modified rewrites on arbitrary BFT protocols. To this end, we must formally model the capabilities of Byzantine nodes: we propose a Borgesian simulator, a proof concept meant to reframe Byzantine actors as random actors, capable of simulating all possible Byzantine behavior through random outputs. We reason on the Borgesian simulator to prove that the possible outputs a Byzantine actor can create before and after our modified rewrites are the same. We additionally introduce a new rewrite, partial independent decoupling, to adapt our suite of rewrites to design patterns common in BFT protocols such as PBFT. Our proposed rewritten version of PBFT is thus capable of view changes and checkpointing while optimizing the critical path.

CCS CONCEPTS

• **Computing methodologies → Distributed computing methodologies**; • **Information systems → Query optimization**.

KEYWORDS

Distributed Systems, Byzantine Fault Tolerance, Query Optimization, PBFT, Datalog, Partitioning, Dataflow

1 INTRODUCTION

Designing systems capable of handling arbitrary failures is challenging [23]—designing such systems that do so at scale doubly so [5, 6, 12, 15, 17]. As such, the path to creating safe, scalable BFT [19] protocols currently remains uncertain and error-prone. The prevailing approach to creating new protocols relies on the ad-hoc discovery of insights and often rebuilds protocols from scratch, usually with the introduction of additional complexity that becomes increasingly difficult to reason about. We argue that taking an alternative approach—adapting existing, simpler protocols to become scalable by breaking them down into components to scale individually—yields just as good results with far less trouble.

Gupta et al. [16] demonstrated the potential behind this approach by pipelining and partitioning existing BFT protocols to increase their throughput by up to 6×. Chu et al. [11] similarly optimized Paxos using local, rule-driven program rewrites, leveraging **decoupling** (dividing program logic across sequential nodes) and **partitioning** (dividing message flow across multiple nodes in parallel) techniques to achieve a 3× throughput improvement. This approach is especially attractive due to its small footprint and protocol-agnostic application domain—rewrites can be applied to arbitrary distributed programs in small increments and are thus easily verified.

We would like to borrow these ideas used to scale protocols like Paxos, but these optimizations are not safe in the face of Byzantine actors. To support this claim, we consider PBFT [8], a seminal BFT protocol reaching consensus on $n = 3f + 1$ machines tolerating f Byzantine failures.

On the critical path, replicas receive PRE-PREPARE messages from the primary replica containing a command to execute, its hash digest, and a signature from the primary over the hash digest. Replicas accept such PRE-PREPARE messages if the signature over the digest is valid and the hash digest of the command matches the provided digest. Later, when replicas receive $2f + 1$ corresponding COMMIT messages (each containing a digest) from other replicas, they verify that the digest of the COMMIT messages matches that of the PRE-PREPARE; if so, replicas execute the command and send a reply to the client containing the result.

Since the replica accumulates PRE-PREPARE and COMMIT messages monotonically over time, the original rewrites from Chu et al. suggest that *monotonic decoupling* [11] can be used to scale each replica by dividing the logic responsible for processing PRE-PREPARE messages from the logic responsible for processing COMMIT messages. We will henceforth refer to these components as the *prepreparer* and *committer*, respectively. The remaining phases of PBFT can be decoupled in a similar fashion, which is further discussed in Section 7. When a prepreparer accepts a valid PRE-PREPARE from the primary replica, it validates the signature, verifies the hash digest of the included command matches that of the signed digest, and crucially forwards the PRE-PREPARE message to its corresponding committer to allow it to execute the client request.

This rewrite, while having the potential to reduce the load on both the prepreparer and committer and improving throughput, is actually *unsafe* in the presence of Byzantine failures. Since the existing rewrites do not secure the new message channel created between each prepreparer and its corresponding committer, a Byzantine prepreparer could send tampered PRE-PREPARE messages to two other committers containing different commands and the original signed digest. Other committers, not checking that the incoming PRE-PREPARE message originates from its corresponding preprepaer, may compare the digests within the $2f + 1$ COMMIT messages with the digest of this tampered PRE-PREPARE, find that they match, and execute different commands—violating the consensus invariant.

The solution is simple: we introduce *sender verification* (Section 5.1) by signing and verifying messages on new message channels, and additionally introduce *message verification* (Section 5.2) by requiring partitions to individually enforce that they are sent the correct subset of hash-partitioned messages. These forms of verification

are proposed as modifications to the existing rewrites to prevent Byzantine nodes from introducing unwanted behavior. Specifically, we want to prove that after our modified rewrites, (1) nodes unaffected by rewrites cannot be sent Byzantine messages not possible before the rewrites, and (2) all modified nodes uphold the invariants required by the rewrites in the face of Byzantine faults. These guarantees would demonstrate that Byzantine attacks on nodes unaffected by rewrites are identical to attacks carried out pre-rewrites and that attacks on nodes modified by the rewrites are ineffective.

To reason on what Byzantine actors can and cannot send others in the system, we devise a model of Byzantine nodes that aims to capture the full scope of their capabilities. The creation of this model is driven by the observations that (1) the behavior of a Byzantine node can be described in terms of its outputs, and (2) the set of possible Byzantine outputs is exactly the same as the set of all random outputs. As such, we introduce the *Borgesian simulator*, used as a proof construct to allow us to formally reason about the capabilities of a Byzantine node before and after our modified rewrites.

Protocols designed to tolerate Byzantine faults additionally require different design paradigms compared to those of CFT protocols. Consider the view-change protocol of PBFT, during which the acceptance of a valid NEW-VIEW message prompts the replica to enter a new view. If the replica is rewritten to decouple the prepreparer and the committer, asynchrony may cause the two components to become out of sync on the current view, potentially threatening liveness, as discussed in Section 6. We introduce a new rewrite, *partial independent decoupling*, as a means for decoupled protocols to asynchronously "share" values with each other while satisfying linearizability.

Existing results demonstrate that these rewrites, while simple and small in scale, are capable of scaling the throughput of PBFT's critical path by 5× [9].

Portions of the material in this report previously appeared in [9]; here, our work formalizes those same ideas in Dedalus. Additionally, the *partial independent decoupling* rewrite is introduced as a follow-up to challenges posed in [9].

2 BACKGROUND

2.1 Byzantine Fault Tolerance

Byzantine fault tolerance (BFT) involves collaboration between nodes in the face of arbitrary Byzantine [19] failures, often applied to the problem of consensus and state machine replication (SMR). BFT is useful for decentralized systems—seeing widespread adoption in blockchains like Ethereum, which runs the Gasper BFT consensus protocol [7]—or for general-purpose computing systems—which may wish to tolerate unexpected behavior beyond crash faults due to memory or disk corruption, application bugs, etc. [12, 13, 24]

PBFT [8] is a fundamental BFT protocol that takes a leader-based approach to solving BFT SMR in an asynchronous network environment, proposing a solution requiring $O(n^2)$ communication and two round trips under a stable leader. Optimizations focused on increasing the throughput of PBFT have led to protocols such as HotStuff [27], which pipelines the different stages of PBFT to achieve linear communication for consensus and view changes under partial synchrony [14], and Zyzzyva [17], which introduces an optimistic linear fast path into PBFT using speculative execution. Safety is a key concern when solving BFT consensus, and remains tricky to this day; in 2017, Abraham et al. exposed a safety violation in Zyzzyva's view change protocol [2]—ten years after its publication. New developments in the space of BFT consensus continue to be built on novel insights requiring rigorous, time-consuming proofs of correctness.

2.2 PBFT

PBFT (Practical Byzantine Fault Tolerance) [8] is a seminal BFT consensus protocol that we will continually use to demonstrate and evaluate our contributions. PBFT requires $3f + 1$ replicas to tolerate f Byzantine faults in an asynchronous environment, using a leader-based protocol broken up into *pre-prepare*, *prepare*, and *commit* phases. All replicas, including the leader, begin in the pre-prepare phase. Upon receiving a client REQUEST message, the leader assigns the request a sequence number and broadcasts a PRE-PREPARE messages to all replicas, including itself. Upon receiving a valid PRE-PREPARE message from the leader, replicas enter the prepare phase for the corresponding request and subsequently broadcast PREPARE messages to all other replicas, including itself. Upon receiving a quorum of $2f + 1$ valid PREPARE messages from other replicas, replicas enter the commit phase and subsequently broadcast COMMIT messages to all other replicas, including itself. Upon similarly receiving a quorum of $2f + 1$ valid COMMIT messages from other replicas, the request is committed; replicas execute the request and send a REPLY message back to the client. Clients wait for a weak quorum of $f + 1$ matching REPLY messages before accepting a response and sending another request.

Leaders are deterministically chosen based on the current **view** of the system. Unresponsive leaders are rotated out and replaced via a view-change protocol. If too much time has passed since receiving a client request without committing it in the current view, replicas attempt a view change into the next view by broadcasting VIEW-CHANGE messages to all other replicas. The leader of the next view, upon receiving $2f + 1$ valid VIEW-CHANGE messages from other replicas, consolidates them into a NEW-VIEW message, broadcasting it to all replicas. Replicas, upon receiving a valid NEW-VIEW message from the new leader, transition into the new view.

Checkpoints in the form of snapshots of the state machine state are also periodically produced to aid with garbage collection. A replica produces a checkpoint after executing a fixed interval of client requests. Upon producing a checkpoint, replicas broadcast CHECKPOINT messages containing the hash digest of the snapshot, alongside its associated sequence number. Upon receiving $2f + 1$ matching valid CHECKPOINT messages, the replica's associated checkpoint becomes a **stable checkpoint**. Messages associated with sequence numbers below the latest stable checkpoint are able to be discarded for garbage collection.

2.3 Rule-Driven Rewrites

Chu et al. introduced a series of simple, protocol agnostic, local rewrites that can be systematically applied to remove bottlenecks and scale crash fault tolerant (CFT) protocols [10, 11]. These rewrites are driven by static analysis of the protocol's code, and are able to be automatically performed if certain preconditions are met. The rewrites are proven to maintain the existing semantics of the protocol implementation, guaranteeing a property known as *observable equivalence*—which will be discussed more in detail in Section 2.5. The rewrites aim to improve the throughput of protocols by leveraging techniques known as *decoupling* and *partitioning*. Decoupling introduces pipelining by dividing the protocol logic of a single node l onto two nodes l_1 and l_2, while partitioning introduces parallelism by dividing the flow of data through one node l over multiple nodes $\bar{l} = \{l_1, l_2, \ldots\}$. Chu et al. demonstrated how, using their presented rewrite rules, they were able to scale the throughput of 2PC [22] by 5× and Paxos [18] by 3×.

These rewrites, if applied directly to a Byzantine fault tolerant environment, will introduce new avenues of attack for Byzantine actors to exploit. Decoupling rewrites, for instance, turn a single node into a pipeline of multiple nodes—introducing new message channels for Byzantine nodes to unexpectedly interject messages into. Partitioning modifies existing message channels, requiring a set of messages originally meant for a single node to be selectively rerouted instead towards one of its many partitions—opening the door for Byzantine nodes to reroute messages incorrectly or duplicate messages across partitions. As such, protocols designed to be Byzantine fault tolerant may find themselves vulnerable to unexpected behavior if these rewrites are not modified accordingly.

2.4 Dedalus

The rewrites presented by Chu et al. are expressed and performed in the Dedalus [4] programming language. Dedalus is a spatiotemporal programming language used to represent distributed systems as the derivation of data between tables over time. It is developed as a restricted subset of Datalog¬ [1], a SQL-like declarative logic language [21] that provides syntax for rule expression that is equivalent to the relational algebra constructs for selection, join and projection—and additionally includes support for recursion, aggregation, and negation. Dedalus modifies Datalog¬ to include the notion of *space* and *time*, allowing it to represent how data may be distributed across and passed asynchronously between many different nodes in a system, as well as allowing it to model how the state of the system evolves over time.

We first introduce the relevant Datalog¬ terminology required to formalize the ideas presented in this paper. Datalog¬ programs are composed of **rules** which dictate how data—henceforth called **facts**, **tuples**, or **rows**—are produced for tables called **relations**. Relations define tables whose columns are called **attributes**. A fact of a relation r is thus a tuple of values, one for each attribute of r, representing a row in the table. A Datalog¬ rule references a single relation in its **head** (the relation for which facts will be derived) and a series of literals in its **body** (representing conditions under which facts can be derived). A **literal** can be a relation or a boolean expression. The attributes of body relations can be either constants or **variables**—the attributes of head relations can additionally be **aggregation functions** (such as SQL's `MAX` or `COUNT`). Positive literals within the body of a rule are joined and negative literals are anti-joined. Repeating the same variable in different attribute positions enforces the equality of values in those positions; hence when a variable is repeated across multiple body literals of the same rule, it enforces an equi-join on the matching attributes (rather than a cartesian product). A fact is created for the head relation if (1) attributes that have the same variable bound to the same variable share the same value, and (2) the result of joining all body literals (and anti-joining negated literals) is non-empty.

Functions can be modeled in Datalog¬ as **infinite relations** with facts in the form of `(in_1,in_2,...,out_1,out_2,...)`. An example of an infinite relation is `digest(x,d)`, which contains a tuple `(x,d)` if and only if taking the hash digest of `x` yields `d`. Rather than explicitly containing tuples for all possible combinations of input values and their corresponding outputs, infinite relations can be lazily evaluated for the finite set of output values corresponding to a given input. Therefore, to ensure finite derivations, infinite relations must have their input variables repeated in the attributes of other (finite) relationsso that the set of variable bindings for the function evaluation is finite.

Relations containing facts that are specified before the execution of the program are known as extensional relations, or **EBD relations**. Relations with derived facts, defined by the heads of rules, are called intensional relations, or **IDB relations**. We say that function symbols, including boolean operators, are also part of the EDB—despite being infinite, they are defined independently of the program.

Dedalus programs are legal Datalog¬ programs, additionally constrained to represent the notion of time and space. Specifically, all IDB relations in Dedalus must contain the space and time suffix attributes l and t, representing where and when a fact exists in the system, respectively. Additionally, the location and time of all body literals of a rule must be bound to the same l and t, respectively—this is to represent the physical constraint that facts can only be joined if they exist at the same place at the same time. Constants, infinite relations, and other forms of EDB relations are assumed to be unchanging and replicated across all nodes in a system, and as such are not required to have l and t suffix attributes for syntax sugar purposes.

The time and location attributes in head relations are also constrained to model the requirements of physical reality, and differ across three kinds of rules.

Synchronous rules, also known as deductive rules, bind the time attribute in the head to the same variable as all body literals. These rules represent simultaneous computation within the same logical timestep, which is only possible within one machine—as such, the location attribute of the head relation must be similarly bound to the same variable as all body literals.

Sequential rules, also known as inductive rules, are characterized by the head relation's time attribute being bound to the successor t' of the time t in the body literals (such that $t' = t + 1$). In an asynchronous system, this is only possible to guarantee within one machine; as such, the location attribute of the head relation is

bound to the same variable as all body literals. Sequential rules are used to **persist** the existence of facts across time on a machine, and are therefore used to represent the evolving state of machines in the system.

Asynchronous rules represent the asynchronous passing of facts between distinct locations. As such, the location attribute in the head (the destination of the sent fact) may be bound to a different variable than the rest of the body, and the time attribute of the head (the arrival time of the sent fact) is bound to a variable which is arbitrarily delayed behind the time variable of the body literals. Specifically, this behavior is represented by the built-in `delay` relation, which independently and non-deterministically chooses a time for each sent fact to arrive at its destination. A more detailed discussion on the formalism of `delay` is discussed in [3]. We refer to **input channels** as the relations through which facts are received asynchronously, and **output channels** as the relations through which facts are sent.

We additionally refer to a **component** of a Dedalus program to be a non-empty set of rules. This terminology will be useful when determining how protocols can be rewritten—decoupling, for instance, divides Dedalus program logic into two components to be run on separate machines. Rewrites are performed in Dedalus based on preconditions that examine which relations are **referenced** by others—specifically, a head relation references all relations in its body.

2.5 Correctness

The rewrites introduced by Chu et al. (with the modifications presented here) must not alter the behavior of the original input protocol if they are to be correct. It is important to note that we make no assumptions about what goals the original protocol aims to achieve, nor do we make any assumptions about the "correctness" of any given protocol design. We reason here about the correctness of rewrites, not the protocols being rewritten. As such, we define our rewrites as correct if they *maintain* the existing semantics of the given protocol—we call this property *observable equivalence*. Colloquially, this means that to an outside observer, the behavior of the rewritten protocol is indistinguishable from that of the original implementation. More specifically, a rewrite is correct if—given a program P, a set of inputs to the system (and their send times), and a rewritten program P'—the set of outputs received from P' was possible under some run of P. In other words, for any given set of inputs, the set of possible outputs under P' is a subset of the possible outputs under P.

The safety of rewrites is of key concern as well—in the face of Byzantine failures, it is crucial that our rewrites do not increase the capabilities of Byzantine nodes. Our discussion on the modifications made to the original rewrites proposed by Chu et al. will aim to address the concern of increased Byzantine influence on the rewritten protocol. We additionally introduce the notion of a *fault domain* to discuss failures under rewritten protocols. Each node l in the original protocol belongs in its own fault domain; when the components of l are decoupled and partitioned, the collection of nodes $\bar{l}$ originating from l remain in the same fault domain. We claim that the rewrites with our presented modifications do not cause an increase in the number of fault domains present in the system. Consider how, to an outside observer, the set of outputs from all $l_i \in \bar{l}$ in a rewritten protocol can correspond to the set of outputs from l in the original protocol. Any Byzantine failure in a machine l_i from the rewritten protocol can therefore be considered as a failure in l from the original protocol—if l is "partially" faulty due to a failure in some of its components, l's behavior as a whole can be classified as Byzantine. The influence of a failure in any machine will never expand beyond its fault domain; as such, if the original protocol is able to tolerate up to f Byzantine failures, the rewritten protocol can tolerate up to f Byzantine failures as well.

3 SYSTEM MODEL

3.1 Failure Model & Network Assumptions

Dedalus assumes an asynchronous network environment, meaning messages can be arbitrarily delayed or reordered in transit, but must eventually be delivered after an unbounded amount of time. We assume that f Byzantine failures are present in the system, with nodes suffering Byzantine failures being able to arbitrarily deviate from the defined protocol—this includes but is not limited to crashing, delaying messages, equivocation, message duplication, or sending otherwise incorrect messages. We additionally assume Byzantine nodes are capable of collusion and can share secret keys. Lastly, we assume the system operates under a shared-nothing architecture; nodes can only communicate with each other through messages.

3.2 Cryptography Primitives

BFT protocols commonly assume that adversaries cannot subvert the security guarantees of available cryptographic primitives [8, 17, 20, 26]; for example, Byzantine actors should not be able to produce a valid signature over a message on behalf of a correct node without knowledge of its secret key. To properly reflect these assumptions made by many BFT protocols and model their behavior appropriately, we introduce new Dedalus formalisms to represent common cryptography primitives. These primitives will be used to allow us to formalize the capabilities and limitations of Byzantine nodes, as well as reason about correctness.

We model the behavior of signatures in Dedalus with the infinite EDB relations `signMessage(msg,sk,sig)` and `verifySignature(msg,pk,sig)`. `signMessage` contains the tuple `(msg,sk,sig)` if and only if signing message `msg` with secret key `sk` yields the signature `sig`. Likewise, `verifySignature` contains the tuple `(msg,pk,sig)` if and only if the signature `sig` over message `msg` was created using a secret key corresponding to the public key `pk`. We additionally define the EDB relations `publicKeys(l,l',pk)` and `secretKeys(l,l',sk)` to denote the public and secret keys used to sign and verify messages sent from l to l', respectively.

It is additionally common for BFT protocols to involve signing over sets of values—for instance, NEW-VIEW messages in PBFT require the new primary to send (and thus sign over) the set of VIEW-CHANGE messages it has received from other replicas. As such, we introduce additional formalisms in Dedalus to allow for such signing capabilities. To this end, we introduce the `signMessage<(...),sk>` aggregation function, alongside its

counterpart verifySignature<(...),pk,sig> aggregation function. signMessage<(...),sk> outputs the signature sig resulting from signing over all aggregated values with the secret key *sk*. verifySignature<(...),pk,sig> outputs a boolean representing if signing over the set of values (...) with the secret key corresponding to the public key *pk* produces the signature sig.

Our introduced formalism is agnostic to the signature scheme actually used to sign messages and can be made capable of supporting their different behaviors and properties—we specifically highlight MACs (Message Authentication Codes) and public-key signatures as the most common schemes in BFT protocols. Both MACs and public-key signatures produce signatures that can be verified with the correct key; these keys are symmetric for MACs and asymmetric for public-key signatures. The difference between the two schemes lies in the likeness of the secret and public keys, as well as their availability to nodes. MACs rely on the symmetric key k to both sign and verify messages sent between locations l and l'. This key is known only to l and l'. As such, the key k used to create and verify MACs for messages sent from l to l' will correspond to the secretKeys tuple (l,l',k) at location l and the publicKeys tuple (l,l',k) at location l'. Likewise, public-key encryption relies on a secret key and public key to create and verify messages sent from l, respectively. The secret key differs from the public key and is known only to l, whereas the shared public key is known to all l'. As a result, using public-key signatures involves distinct secret and public keys *sk* and *pk* to sign and verify messages sent from l. These keys correspond to the secretKeys tuples (l,l',sk) at location l for all l', alongside the publicKeys tuple (l,l',pk) at each location l'.

4 BORGESIAN SIMULATORS

While much attention in BFT protocol design is paid to the intelligent and malicious collusion driving Byzantine behavior, it's important to note that Byzantine behavior is fundamentally defined by its *arbitrary* nature. Just as Byzantine nodes can act maliciously to take down the system, they also have the capability to correctly adhere to agreed-upon protocols or even act completely nonsensically. We argue that in order to completely account for the full capabilities of Byzantine actors, it's necessary to model a scope of behavior that stretches far beyond what could be considered "intelligent" or even "interesting."

We propose an approach to reasoning about Byzantine behavior by framing Byzantine nodes as *random actors*. We argue that a node that exhibits completely random behavior not only encapsulates intelligent behavior that is interesting to reason about, but is sufficient to capture the full scope of Byzantine actors' capabilities.

This line of reasoning begins with the observation that the capabilities of a Byzantine node are defined with how it interfaces with the rest of the system; in other words, it is only necessary to focus on the outputs of Byzantine nodes when reasoning about their behavior—their internal state is mostly irrelevant to an analysis of their capabilities. A Byzantine node intelligently acting with malicious conviction has just as much impact on the system as a purely random Byzantine node that just so happened to act the same way. Furthermore, if we take the set of outputs (and their corresponding timestamps) resulting from any given execution of a Byzantine node, it's possible to generate an identical output trace—by chance—using a purely random actor. In fact, *every* possible output trace from a Byzantine node can be found in the set of possible outputs from a random actor. It follows that the set of all possible Byzantine executions is *exactly the same* as the set of executions under a random actor.

We call the random model of Byzantine behavior a **Borgesian simulator**[1], which is able to emulate Byzantine executions producing "intelligent" outputs alongside admittedly overwhelmingly large volumes of executions producing random garbage. As a result, it's important to note that we intend to construct a Borgesian simulator as a proof technique to reason about Byzantine behavior, rather than use it in practice as a viable implementation of a Byzantine actor. Modeling Byzantine behavior in this way is particularly useful to us because it is protocol agnostic, and can therefore fit our needs to reason about the correctness of rewrites on arbitrary protocols. Additionally, its output-oriented design aligns nicely to our approach to reasoning about the observable equivalence of systems.

To implement our Borgesian simulator in a way that is protocol agnostic, we devise a **Borgesian harness** that can be added on top of existing protocol implementations to represent Byzantine behavior within the system. Chu et al. proposed an implementation of a Borgesian harness in event-driven pseudocode [9]; we will formalize the same ideas in Dedalus.

4.1 Running Example

In demonstrating the implementation of our Borgesian harness, we invoke the following commitOut example, responsible for collecting PREPARE messages in PBFT. Upon obtaining a quorum of $2f + 1$ matching messages, it broadcasts COMMIT messages to all replicas:

```
1 # Broadcast signed COMMIT messages
2 commitOut(v,n,d,l,sig,l',t') :-
    prepareCertSize(v,n,d,size,l,t),
    currentView(v,l,t), FAILURES(f), size>=2*f+1,
    self(l), secretKeys(l,l',sk),
    signMessage((v,n,d,l),sk,sig), nodes(l'),
    delay((v,n,d,l,sig,t,l'),t')
3 # Receive and log valid PREPARE messages
4 prepareLog(v,n,d,l',sig,l,t) :-
    prepareIn(v,n,d,l',sig,l,t), publicKeys(l',l,pk),
    verifySignature((v,n,d,l'),pk,sig),
    currentView(v,l,t)
5 # Persist logged PREPARE messages
6 prepareLog(v,n,d,l',sig,l,t') :-
    prepareLog(v,n,d,l',sig,l,t), t'=t+1
7 # Count the number of logged PREPARE messages with a
    matching (v,n,d)
8 prepareCertSize(v,n,d,count<l'>,l,t) :-
    prepareLog(v,n,d,l',sig,l,t)
```

Specifically, the component executing this logic accepts PREPARE messages as input via prepareIn. Upon validating the message

[1]The inspiration for this name comes from the author Jorge Luis Borges' short story *The Library of Babel*, which envisioned a library containing a book for every possible fixed-width permutation of characters. This library would theoretically hold books containing the opening acts of Shakespeare's plays, endless alternate endings to *Moby Dick*, and even accurate and vivid depictions of the future—alongside heaps upon heaps of nonsensical gibberish.

signature, checking that the view `v` is up-to-date, and that the `id` attribute matches that of the sender attribute `l'`, the component adds the message to its log—where it is persisted. The relation `prepareCertSize` tracks the number of matching PREPARE messages. Upon receiving $2f + 1$ matching PREPARE messages in the current view, the component signs and broadcasts COMMIT messages with the same view, sequence number, and digest as the PREPARE quorum. The address of the component is found in the `self` relation, and the addresses of all nodes in the system are present in the `nodes` relation. The constant f is stored in the `FAILURES` EDB relation.

This quorum-driven collect-then-broadcast approach is common in many distributed protocols, and serves as a simple foundation for us to apply our Borgesian harness upon.

4.2 Arbitrary Message Output

We formalize the behavior of our Borgesian simulator in Dedalus by first defining its ability to act "completely randomly." Specifically, at every timestep, for each available input channel on every node, the Borgesian simulator should send random numbers of messages populated with random values to each machine.

We can accomplish this by introducing a new kind of Dedalus relation: for each output relation r, define the function `rRandomArgs(l,l',t,u_1,u_2, ...)`.

`rRandomArgs(l,l',t,u_1,u_2,...)` generates a random number of facts with random values `u_1`, `u_2`, etc. each timestep. Each fact corresponds to the values assigned to attributes of r in a message being sent to l' at timestep t. As such, the data types of `u_1`, `u_2`, etc. correspond to the data types of the attributes of r.

We introduce a new `commitOut` rule to model Byzantine outputs through this channel:

```
1  # Output random garbage through this output channel
     if Byzantine
2  commitOut(v,n,d,id,sig,l',t') :- byzantine(l),
     nodes(l'), commitOutLimit(l,l',t,lim),
     commitOutRandomArgs(l,l',t,v,n,d,id,sig), i<n,
     delay((v,n,d,id,sig,i,l,t,l'),t')
```

Note the addition of the `byzantine(l)` relation, which contains a row for each node l in the system that is Byzantine. Its inclusion in the body of `commitOut` ensures that the harness is only used by Byzantine nodes and does not affect the behavior correct nodes. Likewise, we need to add `!byzantine(l)` to the body of all existing relations to ensure all Byzantine logic goes through our harness—Byznatine nodes should not have to execute regular logic.

```
1  # Correct nodes continue using existing logic
2  commitOut(v,n,d,l,sig,l',t') :- !byzantine(l),
     prepareCertSize(v,n,d,size,l,t),
     currentView(v,l,t), FAILURES(f), size>=2*f+1,
     self(l), secretKeys(l,l',sk),
     signMessage((v,n,d,l),sk,sig), nodes(l'),
     delay((v,n,d,l,sig,t,l'),t')
```

4.3 Protected Fields

Observant readers may notice that our current approach to modeling Byzantine nodes is actually too powerful—namely, by allowing signature fields like `sig` in `commitOut` to be directly assigned completely random values, we technically leave open the possibility for our Borgesian simulator to generate valid signatures on behalf of other correct nodes without knowing their secret keys. This would break the security guarantees of the cryptographic primitives in our system, and cannot be allowed. As such, it becomes necessary to distinguish between different kinds of message fields and place restrictions on how certain values are generated.

We divide all message fields into three types: unprotected fields, protected fields, and signatures over protected fields. Our Borgesian simulator will continue to randomly generate values for unprotected fields and protected fields. For signatures over protected fields, our simulator will instead randomly generate signatures using the secret keys it has available. To simulate the sharing of secret keys between colluding Byzantine nodes, we can join `byzantine(l)` with `secretKeys(l,l',sk)` in our harness to allow a Byzantine node to access the keys of any Byzantine l.

We create the `rProtectedArgs(l,l',l'',t,p_1,p_2,...)` relation to generate random values for protected fields in the same way as is currently done for unprotected fields—but with the inclusion of the extra `l''` parameter, defining the Byzantine location from which the Borgesian simulator will take secret keys. Specifically, `rProtectedArgs` now generates a random number of facts for every (l, l', l'', t) quartet, denoting the attribute values of messages that a Byzantine node l will send to l' through output channel r using the secret keys of l'' at timestep t.

We do not create a means for our Borgesian simulator to create invalid signatures—we assume that correct nodes will verify the signatures on the protected fields of incoming messages, discarding any messages with invalid signatures.

```
1  # Randomly generate, sign, and send messages with
     each key
2  commitOut(v,n,d,id,sig,l',t') :- byzantine(l),
     nodes(l'),
     commitOutProtectedRandomArgs(l,l',l'',t,v,n,d,id),
     i<lim, byzantine(l''), secretKeys(l'',l',sk),
     signMessage((v,n,d,id),sk,sig),
     delay((v,n,d,id,sig,l,t,l'),t')
```

Note that in our running example, all attributes of `commitOut` are protected by the signature `sig`.

4.4 Message Forwarding

Byzantine actors cannot spoof the signatures of other correct nodes, but they are still capable of forwarding messages signed by correct nodes. As a result, our Borgesian simulator should log all messages with valid signatures received from others in order to potentially forward them later.

For each input relation r, define the relation `rProtectedLog` in the form of `rProtectedLog(p_1,p_2,...,sig,l,t)`. `rProtectedLog` extracts protected values `p_1`, `p_2`, etc. from input messages of r with a valid signature `sig`, persisting them alongside their signature.

For our example, commitIn is the input relation receiving sent messages from commitOut:

```
1 # Log messages with valid signatures
2 commitInProtectedLog(v,n,d,l',sig,l,t) :-
    commitIn(v,n,d,l',sig,l,t), publicKeys(l',l,pk),
    verifySignature((v,n,d,l'),pk,sig)
3 # Persist logged messages
4 commitInProtectedLog(v,n,d,l',sig,l,t') :-
    commitInProtectedLog(v,n,d,l',sig,l,t), t'=t+1
```

We introduce the function rForward((...),l,l',t), where the existence of the tuple ((...),l,l',t) is randomly determined and dictates that the fact (...) will be forwarded from location l to location l' at timestep t.

We introduce a new commitOut rule to model Byzantine message forwarding through this channel, being sure to restrict its usage to Byzantine nodes with byzantine(l):

```
1 # Randomly forward a random number of each logged
    message
2 commitOut(v,n,d,id,sig,l',t') :- byzantine(l),
    nodes(l'), commitOutForward((v,n,d,id,sig),l,l',t),
    commitInProtectedLog(v,n,d,id,sig,l,t),
    delay((v,n,d,id,sig,l,t,l'),t')
```

4.5 Complex Data Types

It may be necessary to generate random values for more complex message types. Consider PBFT NEW-VIEW messages, which are signed messages containing a set of signed VIEW-CHANGE messages, which themselves contain an array of sets containing signed PREPREPARE and PREPARE messages. The Borgesian harness must be adapted accordingly to accurately generate random messages for these complex message types, selecting a random number of elements to include in each list or set, populating unprotected fields with random values while randomly generating and signing—or forwarding—values for protected fields.

5 MODIFICATIONS TO REWRITES

The rewrites proposed by Chu et al. are meant to be applied to crash fault tolerant environments, and cannot be directly applied to protocols meant to tolerate Byzantine faults. We present two modifications to adapt these rewrites to the BFT environment and formalize them in Dedalus.

5.1 Sender Verification

Decoupling rewrites, as well as the partial partitioning rewrite, introduce new message channels between nodes: in the case of decoupling rewrites, between the newly-decoupled nodes l_1 and l_2, and in the case of partial partitioning, between the coordinator and the partitions. These new message channels are created with the inherent assumption that only certain nodes will send messages through them. Without explicit enforcement of these assumptions, Byzantine nodes can exploit the rewrite by unexpectedly interjecting their own messages through these input channels.

We patch this exploit by introducing **sender verification** to the affected rewrites: each newly introduced message channel from l_1 to l_2 now involves l_1 signing outgoing messages with their secret key, to be verified by l_2 upon receipt. Messages with signatures unable to be verified from l_1 are discarded. Formally, for each relation r on l_1 that outputs to l_2, sign the outgoing message:

```
1 # Before
2 r(...,l',t') :- ... # existing logic
3 # After
4 r(...,sig,l',t') :- secretKeys(l,l',sk),
    signMessage((...),sk,sig), ... # existing logic
```

Additionally, for each input relation r on l_2 receiving inputs from l_1, include a signature attribute sig and create the relation rSenderVerified:

```
1 # Enforce that only messages from desired senders are
    kept
2 rSenderVerified(...,l,t) :- r(...,sig,l,t),
    forward(l',l), publicKeys(l',l,pk),
    verifySignature((...),pk,sig)
```

We take the existing forward relation to derive the location of the upstream node l', taking its public key to verify the signature sig of the received message. We complete the modification by replacing all other references to r in the program with rSenderVerified.

5.2 Message Verification

The rewrite for partitioning replicates the logic on one node l across multiple nodes $\bar{l}$, rerouting messages originally meant for l instead to a single partition $l_i \in \bar{l}$. Messages are rerouted based on a known distribution policy $D(f)$, which deterministically maps facts f to a corresponding destination partition. While these rewrites do not introduce any new message channels, they inherently assume that facts arriving at each partition correctly follow this distribution policy—an assumption that can be exploited by Byzantine nodes by incorrectly rerouting facts to other partitions.

We prevent this exploit by introducing message verification to the affected rewrites: each partition, upon receiving a fact f through one of its input channels, verifies the distribution policy $D(f)$ indeed routes the f to this partition—discarding the message if not. Formally, for each input relation r on a partition l, create the relation rPartitionVerified:

```
1 # Enforce that messages are only processed at the
    correct partition
2 rPartitionVerified(...,l,t) :- r(...,l,t), D(...,l)
```

We model the distribution policy $D(f)$ in Dedalus as an infinite relation mapping the attributes of a fact of r (excluding space and time) to a location l. We complete the modification by replacing all other references to r in the program with rPartitionVerified.

5.3 Correctness

To prove the correctness of these modified rewrites, we will reason with our Borgesian simulator to demonstrate that (1) the number of fault domains doesn't increase after performing our rewrites (as described in Section 2.5), and that (2) post-rewrite, all input channels on correct nodes can correctly handle messages from Byzantine nodes.

Without loss of generality, we prove these properties hold for a given correct node $dest_c$ receiving messages from a specific Borgesian simulator sim_b in the original protocol and sim'_b in the rewritten protocol [9].

We divide input channels into four categories based on how they are affected by our rewrites:

(1) *Unmodified channels*, which correspond to input channels that remain unchanged after rewrites.
(2) *Redirected channels*, which correspond to input channels existing on some l before rewrites and some l' corresponding to l after rewrites.
(3) *Duplicated channels*, which correspond to input channels existing on some l before rewrites and multiple $l_i \in \bar{l}$ corresponding to l after rewrites.
(4) *New channels*, which have no pre-rewrite counterpart and exist to receive messages from l_1 on a corresponding l_2 (where l_2 and l_1 both correspond to an original l before rewrites). These channels are only created as a result of decoupling or partial partitioning.

We prove that our desired properties hold for each of these input channel types.

5.3.1 Unmodified channels. As the name suggests, unmodified channels are the same before and after performing our rewrites. As such, the input logic on these channels remains unchanged, and $dest_c$ continues to handle Byzantine messages through these channels the same as before.

We will now show that sim'_b is unable to send any additional messages through this channel compared to sim_b. As this channel is unmodified, the output logic for sim'_b is identical to that of sim_b. However, we need to demonstrate that sim'_b does not potentially receive additional signatures from correct nodes as a result of the rewrites—otherwise, sim'_b could forward $dest_c$ signatures of correct nodes that sim_b could not, expanding the capabilities of Byzantine nodes and violating correctness.

Assume for the sake of contradiction that sim'_b received a "new" signature that sim_b was unable to receive before the rewrites. This signature must have be received through one of the four kinds of input channels:

(1) *Unmodified channels.* The "new" signature received by sim'_b must have been sent by a correct node. Since the input channel is unmodified, correct nodes send the same messages to this channel post-rewrites as they did pre-rewrites. Therefore, correct nodes that send a signature to an unmodified input channel post-rewrites must have also sent the same signature to the same channel pre-rewrites. It follows that such a "new" signature could not have arrived through this kind of channel.
(2) *Redirected channels.* The messages originally sent by correct nodes to redirected channels at l are simply redirected to arrive at l' after our rewrites. As such, signatures sent by correct nodes to a redirected channel at l' correspond 1:1 to signatures sent by correct nodes to the original location l. Therefore, such a "new" signature could not have arrived through this kind of channel.
(3) *Duplicated channels.* The messages originally sent by correct nodes to duplicated channels at l are simply redirected to arrive at one of $l_i \in \bar{l}$ after our rewrites. As with redirected channels, signatures sent by correct nodes have a 1:1 correspondence between the channels in $\bar{l}$ and l; such a "new" signature could not have arrived through this kind of channel.
(4) *New channels.* These input channels are meant to facilitate message passing between nodes in the same fault domain, as they are only created as a result of decoupling or partial partitioning. As such, a correct node that sends a signature to sim'_b through this kind of channel must be in the same fault domain as the Byzantine sim'_b. Thus, the entire original node can be considered Byzantine (Section 2.5); no correct nodes outside the fault domain of sim'_b will send signatures to sim'_b through these channels.

A "new" signature is therefore unable to arrive at sim'_b through any of its input channels—as such, sim'_b cannot receive (and therefore forward) any signatures from correct nodes previously unable to be received sim_b.

5.3.2 Redirected channels. Redirected channels existing at l before rewrites are simply moved to exist at l' after the rewrites. As such, messages sent by sim'_b to these channels at l' correspond 1:1 to messages sent by sim_b to these channels at l. Therefore, the proof follows similarly to that of unmodified channels.

5.3.3 Duplicated channels. Channels existing at l before the rewrites may be duplicated to exist at each of $l_i \in \bar{l}$ after the rewrites, becoming duplicated channels. These channels assume that each l_i receives a disjoint set of the inputs originally received at l, with the message set received at $\bar{l}$ being equivalent to that received at l in the original protocol. This assumption is enforced by our message verification rewrite (Section 5.2), and as such messages sent by sim'_b to these channels at $\bar{l}$ correspond 1:1 to messages sent by sim_b to these channels at l. Therefore, the proof follows similarly to that of unmodified channels.

5.3.4 New channels. New channels exist only after our rewrites, and as such we must reason that a $dest_c$ input channel of this type is able to correctly handle all Byzantine messages. Say a node l in the original protocol was decoupled into l_1 and l_2 after our rewrites, introducing a new input channel at $dest_c = l_2$. If the fault domain containing $dest_c$ is correct, then we know that the l_1 must also be correct (Section 2.5). As such, $dest_c$ will only receive Byzantine messages through this input channel from nodes outside its fault domain. With sender verification (Section 5.1), however, messages arriving from such locations are discarded—therefore, $dest_c$ will appropriately handle all Byzantine messages (by not processing them at all). Note that if the fault domain containing $dest_c$ is Byzantine, the behavior of $dest_c$ in response to potentially Byzantine messages from l_1 is irrelevant due to us being able to consider the original l as Byzantine (Section 2.5).

6 PARTIAL INDEPENDENT DECOUPLING

Chu et al. decouples the PBFT consensus protocol by the stages of its critical path: leader, proxy leader, pre-preparer, preparer, and committer [9]. However, in PBFT, there exists multiple instances

of "global" variables that are shared between these different components of the protocol. Recall that once a PBFT replica accepts a NEW-VIEW message, it enters a new view—updating its current view to match that of the NEW-VIEW message. The current view of the replica is used to determine who the primary is believed to be, which messages to accept, and so on. Now imagine that each PBFT replica is decoupled, such that the logic to process PREPARE messages and the logic to process COMMIT messages are located on separate machines, respectively dubbed the *preparer* and the *committer*; in order to maintain correctness, the components must act as if they are in sync with each other about the current view. If this weren't the case, consider a preparer component that is in an later view v_1 than its committer counterpart in view v_2. This committer would refuse to commit requests in view v_1 by discarding COMMIT messages sent to it in that view (due to mismatching views), including from its own corresponding preparer. This would cause a correct committer to appear down or Byzantine to other nodes, and could further prove a threat to liveness if this were to occur across multiple replicas.

More generally, asynchrony between decoupled components means that one component cannot immediately send outputs caused by changes to its "shared" relations if it is unsure that a decoupled component referencing those relations has observed the same changes. PBFT involves a couple of instances of these "shared" relations besides the current view, such as the value of the low watermark or a boolean representing whether a view change currently is in progress. The existing rewrites are insufficient to address this concern, and the solution is not immediately obvious. For our preparer/committer example, simply broadcasting NEW-VIEW messages to all components is insufficient: network delays and message dropping may cause different components of the original replica to become out of sync with each other, leaving our problem unresolved.

We propose a new rewrite to identify and handle these kinds of cases without violating observable equivalence: *partial independent decoupling*.

6.1 Overview

Say we want to decouple a component C into C_1 and C_2, such that changes to some relations in C_1 must now be shared asynchronously with C_2. If a component C behaves like a state machine [11], its state at timestep t is only dependent on the inputs received and their orderings prior (regardless of their exact times of arrival). However, to preserve linearizability, there must additionally appear to be a single point in time during which both C_1 and C_2 processed changes to shared relations. Without special attention, this property can be violated—consider a flawed decoupling rewrite that has C_1 immediately process a change to relations referenced by C_2 as it forwards the same change to C_2. To demonstrate how this rewrite is incorrect, say we have a C that implements an append-only log able to be queried for its size or full contents. C can be divided into C_1, containing logic to append entries and output the size of the log, and C_2, containing logic to output the contents of the log. Since C_2 may not have received and processed new entries appended by C_1 by the time C_1 responds to other requests, a client could (1) send a message to C_1 appending an entry to the log, (2) query the size of the log from C_1, but then (3) query the full contents of the log from C_2 before the entry appended by C_1 is received, seeing one less entry than it expects and giving the appearance that its change has been reverted. To avoid scenarios like this and otherwise preserve linearizability, we will thus prevent C_1 from sending outputs until it receives an ACK from C_2 confirming it has received and processed changes to relations it references in C_1. Additionally, we will *freeze* the processing of new inputs at C_1 while waiting for an ACK from C_2; this is likewise to prevent the sending of other outputs from C_1 before it knows that C_2 has received and processed the necessary changes to relations it references in C_1.

Therefore, we propose the following outline for *partial independent decoupling*:

(1) C_1 receives and processes inputs as they arrive, tracking updates to relations referenced by C_2 each timestep. When updates are detected, C_1 forwards these updates to C_2 as a batch.
(2) In the same timestep, C_1 additionally prevents itself from sending outputs until it receives an ACK from C_2.
(3) Starting the next timestep, C_1 additionally *freezes*. Additional inputs arriving while C_1 is frozen are blocked from being processed and persisted until C_1 unfreezes.
(4) C_2, upon receiving a batch from C_1, processes it. C_2 then sends an ACK to C_1 confirming it has processed the received batch.
(5) C_1, upon receiving an ACK for a batch from C_2, is able to send the outputs it initially blocked in step 2.
(6) The next timestep, C_1 additionally *unfreezes*. Inputs blocked from being processed in step 3 are all simultaneously processed. This may cause C_1 to block outputs again and additionally freeze in the next timestep.

Note that as this rewrite introduces new message channels between C_1 and C_2, it will have to be secured as described in Section 5.1 if it's to be safely applied to a Byzantine fault tolerant environment.

6.2 Rewrite

6.2.1 Precondition. We first adjust the definition of independent given by Chu et al. in [11]: component C_1 is now **independent** of component C_2 if C_1 does not reference any relation found in the head of a rule in C_2. We additionally introduce the notion of input sharing: components C_1 and C_2 are **input sharing** if they each contain a rule referencing the same relation in their bodies.

Partial independent decoupling thus requires the precondition that component C is a state machine that can be divided into C_1 and C_2, where C_1 is independent of C_2.

Note the precondition for partial independent decoupling does not involve the notion of input sharing. However, this definition applies to the precondition of mutually independent decoupling [11], which we modify to now require both that (1) C_1 is independent of C_2 and (2) C_1 and C_2 are not input sharing. Without the second condition, facts for relations referenced by both C_1 and C_2 would to be replicated to both components. In the absence of coordination,

there is no guarantee that C_1 and C_2 will process these shared facts at the same time, threatening linearizability.

Conservative checks to confirm if C is a state machine are discussed by Chu et al. in their description of state machine decoupling [10], and will not be discussed here in detail. However, we borrow the terms **no-change dependency** and **existence dependency** [10] coined by Chu et al. in their description for our own use.

6.2.2 C_1 Batch Creation & Forwarding. As mentioned above, the partial independent decoupling rewrite involves forwarding facts from C_1 to C_2 for each relation r referenced in C_2 whose head is in C_1. In their definition of state machine decoupling [10], Chu et al. shows how each of these relations falls into one of two categories—either having a no-change dependency or an existence dependency on the inputs of C_2 [10].

We will first discuss the rewrites necessary for relations with a no-change dependency on their parents. Let $\overline{r}_{nc} = \{r_1, r_2, ...\}$ be the set of relations referenced in C_2 whose head is in C_1 such that $r_{nc} \in \overline{r}_{nc}$ has a no-change dependency on its parents (in C_1). C_1 tracks the delta of what is in each r_{nc} between timesteps:

```
1 # Track the state of r_nc during the last timestep
2 r_ncPrev(...,l,t) :- r_nc(...,l,t), t'=t+1
3 # Facts present in r_nc during the current timestep
    but not the last
4 r_ncDeltaPos(...,l,t) :- r_nc(...,l,t),
    !r_ncPrev(...,l,t)
5 # Facts present in r_nc during the last timestep but
    not the current
6 r_ncDeltaNeg(...,l,t) :- r_ncPrev(...,l,t),
    !r_nc(...,l,t)
```

To determine when changes to the contents of relations in $\overline{r}_{nc}$ have occurred, we count the number of facts in all deltas of $\overline{r}_{nc}$ during each timestep. For each $r_{nc} \in \overline{r}_{nc}$, add the following to C_1:

```
1 r_ncDeltaPosCount(count<(...)>,l,t) :-
    r_ncDeltaPos(...,l,t)
2 r_ncDeltaNegCount(count<(...)>,l,t) :-
    r_ncDeltaNeg(...,l,t)
```

Additionally, create the `currentBatchSize` relation, where r_{nc1}, r_{nc2}, etc. are all in $\overline{r}_{nc}$:

```
1 # Sum the size of all deltas across all relations r_nc
2 currentBatchSize(size,l,t) :-
    r_nc1DeltaPosCount(n1Pos,l,t),
    r_nc1DeltaNegCount(n1Neg,l,t),
    r_nc2DeltaPosCount(n2Pos,l,t),
    r_nc2DeltaNegCount(n2Neg,l,t), ...,
    size=n1Pos+n1Neg+n2Pos+n2Neg+...
```

The total batch size will be used by C_2, allowing it to know when it has received all facts within a batch.

We create a batch if deltas are observed during the current timestep. C_1 notifies C_2 of an incoming batch by sending it its size via `batchOut`:

```
1 # Send the batch size to C_2
2 batchOut(size,l',t') :- currentBatchSize(size,l,t),
    size>0, forward(l,l'), delay((size,l,t,l'),t')
```

We now prepare to forward the deltas in $\overline{r}_{nc}$ to C_2 as part of a batch. For each $r_{nc} \in \overline{r}_{nc}$, add the following in C_1:

```
1 r_ncDeltaPosBatchedOut(...,l',t') :-
    r_ncDeltaPos(...,l,t), forward(l,l'),
    delay((...,t,tPrev,l,t,l'),t')
2 r_ncDeltaNegBatchedOut(...,l',t') :-
    r_ncDeltaNeg(...,l,t), forward(l,l'),
    delay((...,l,t,l'),t')
```

C_2 may also reference relations in C_1 with an existence dependency on their parents; facts generated in these relations must be similarly forwarded to and processed by C_2. To this end, let $\overline{r}_e = \{r_1, r_2, ...\}$ be the set of relations referenced in C_2 whose head is in C_1 such that $r_e \in \overline{r}_e$ has an existence dependency on its parents (in C_1). C_1 watches for the generation of facts in $\overline{r}_e$ by counting the number of facts present in each r_e during each timestep:

```
1 r_eCount(count<(...)>,l,t) :- r_e(...,l,t)
```

These counts will contribute to the same `currentBatchSize` relation used by no-change dependency relations $r_{nc} \in \overline{r}_{nc}$. Modify the `currentBatchSize` relation, where r_{e1}, r_{e2}, etc. are all in $\overline{r}_e$:

```
1 # Sum the size of all deltas across all relations
    r_nc and r_e
2 currentBatchSize(size,l,t) :-
    r_nc1DeltaPosCount(n1Pos,l,t),
    r_nc1DeltaNegCount(n1Neg,l,t),
    r_nc2DeltaPosCount(n2Pos,l,t),
    r_nc2DeltaNegCount(n2Neg,l,t), ...,
    r_e1Count(m1,l,t), r_e2Count(m2,l,t), ...,
    size=n1Pos+n1Neg+n2Pos+n2Neg+...+m1+m2+...
```

No-change dependency relations and existence dependency relations share the same batch—a batch will be created in the current timestep if any deltas are detected in $\overline{r}_{nc}$ *or* any facts are created in $\overline{r}_e$.

We additionally need C_1 to forward facts of relations in $\overline{r}_e$ to C_2. For each $r_e \in \overline{r}_e$, add the following in C_1:

```
1 r_eBatchedOut(...,l',t') :- r_e(...,l,t),
    forward(l,l'), delay((...,l,t,l'),t')
```

In the same timestep C_1 forwards a batch to C_2, it additionally prevents the sending of outputs until if receives an ACK from C_2 for this batch. In future timesteps, C_1 will additionally *freeze*, delaying the processing of inputs until after it has received the ACK from C_2 for the batch. The logic behind both these mechanisms is discussed in Section 6.2.4.

6.2.3 C_2 Batch Processing & Acknowledgement. C_2 collects batches from C_1, using the provided size to recognize when a batch has been fully received. First, C_2 must collect and persist deltas and facts forwarded from C_1.

For each $r_{nc} \in \overline{r}_{nc}$, add the following rules to C_2:

```
1  # Receive forwarded deltas from the
     r_ncDeltaPosBatchedIn and r_ncDeltaNegBatchedIn
     input channels
2  r_ncDeltaPosBatched(...,l,t) :-
     r_ncDeltaPosBatchedIn(...,l,t)
3  r_ncDeltaNegBatched(...,l,t) :-
     r_ncDeltaNegBatchedIn(...,l,t)
4  # Conditional persist as long as the batch hasn't
     been sealed
5  r_ncDeltaPosBatched(...,l,t') :-
     r_ncDeltaPosBatched(...,l,t), !batchSealed(l,t),
     t'=t+1
6  r_ncDeltaNegBatched(...,l,t') :-
     r_ncDeltaNegBatched(...,l,t), !batchSealed(l,t),
     t'=t+1
```

For each $r_e \in \overline{r}_e$, add the following rules to C_2:

```
1  # Receive forwarded facts from the r_eBatchedIn input
     channel
2  r_eBatched(...,l,t) :- r_eBatchedIn(...,l,t)
3  # Conditional persist as long as the batch hasn't
     been sealed
4  r_eBatched(...,l,t') :- r_eBatched(...,l,t),
     !batchSealed(l,t), t'=t+1
```

C_2 additionally learns the size of a batch upon receiving the batch's size from C_1 via the `batchIn` input channel:

```
1  pendingBatch(size,l,t) :- batchIn(size,l,t)
2  # Conditional persist as long as the batch hasn't
     been sealed
3  pendingBatch(size,l,t') :- pendingBatch(size,l,t),
     !batchSealed(l,t), t'=t+1
```

We say a batch is *sealed* once it has been fully received. C_2 now counts the deltas and facts received to know when it has fully received the batch.

For each $r_{nc} \in \overline{r}_{nc}$, add the following rules to C_2:

```
1  r_ncDeltaPosBatchedCount(count<(...)>,l,t) :-
     r_ncDeltaPosBatched(...,l,t)
2  r_ncDeltaNegBatchedCount(count<(...)>,l,t) :-
     r_ncDeltaNegBatched(...,l,t)
```

For each $r_e \in \overline{r}_e$, add the following rule to C_2:

```
1  r_eBatchedCount(count<(...)>,l,t) :-
     r_eBatched(...,l,t)
```

Additionally, add:

```
1  # Sum the size of all deltas across all relations
     r_nc and r_e
2  receivedBatchSize(size,l,t) :-
     r_nc1DeltaPosBatchedCount(n1Pos,l,t),
     r_nc1DeltaNegBatchedCount(n1Neg,l,t),
     r_nc2DeltaPosBatchedCount(n2Pos,l,t),
     r_nc2DeltaNegBatchedCount(n2Neg,l,t), ...,
     r_e1BatchedCount(m1,l,t), r_e2BatchedCount(m2,l,t),
     ..., size=n1Pos+n1Neg+n2Pos+n2Neg+...+m1+m2+...
```

Upon receiving all deltas and facts from C_1, the batch is sealed:

```
1  batchSealed(l,t) :- pendingBatch(size,l,t),
     receivedBatchSize(size,l,t)
```

Once a batch is sealed, C_2 can safely apply the deltas and add the facts forwarded from C_1. To control access to these deltas and facts by the other relations of C_2, replace all references to relations r in $\overline{r}_{nc}$ or $\overline{r}_e$ with `rSealed`.

Additionally, for relations $r_{nc} \in \overline{r}_{nc}$, add the following:

```
1  # Facts in the positive delta are added to r_ncSealed
     after this timestep
2  r_ncSealed(...,l,t') :- r_ncDeltaPosBatched(...,l,t),
     batchSealed(l,t), t=t'+1
3  # Facts in the negative delta are removed from
     r_ncSealed after this timestep
4  r_ncRemoved(...,l,t) :- r_ncDeltaNegBatched(...,l,t),
     batchSealed(l,t)
5  # When no batch is sealed, this persists r_ncSealed
6  r_ncSealed(...,l,t') :- r_ncSealed(...,l,t),
     !r_ncRemoved(...,l,t), t=t'+1
7  # The delta is applied to rSealed during this timestep
8  rSealed(...,l,t) :- r_ncDeltaPosBatched(...,l,t),
     batchSealed(l,t)
9  rSealed(...,l,t) :- r_ncSealed(...,l,t),
     !r_ncRemoved(...,l,t)
```

`r_ncSealed` is created to enable `rSealed` to apply the necessary deltas in the same timestep a batch is sealed at C_2. Note that `r_ncSealed` contains the same facts as `rSealed`, only with all time attributes delayed by one timestep.

For relations $r_e \in \overline{r}_e$, add the following:

```
1  # Add the fact to rSealed
2  rSealed(...,l,t) :- r_eBatched(...,l,t),
     batchSealed(l,t)
```

C_2 additionally sends C_1 an ACK once it has sealed a batch; it does so via the output channel `batchAckOut`:

```
1  # Send an ACK once a batch is sealed
2  batchAckOut(l',t') :- batchProcessed(l,t),
     forward(l',l), delay((l,t,l'),t')
```

6.2.4 Blocking Outputs & Freezing C_1. We will now rewrite each of the output channels of C_1 to delay sending outputs while waiting for an ACK from C_2. For each output rule of relation r (specifically, each rule of r containing the `delay` relation in the body), we will turn it from an asynchronous rule of r to a synchronous rule of a new relation *rPending*. *rPending* has the same attributes as r, with the addition of an additional attribute that will act as the new location variable for facts in *rPending*.

We create a rule φ' of *rPending* from an output rule φ of r by (1) removing the `delay` rule in the body of φ, (2) binding the time attribute in the head of φ to the same variable as all body literals' time attributes, (3) adding an additional location attribute to the head of φ, bound to the same variable as all body literals' location attributes, and (4) replacing the reference to r in the head of φ with `rPending` such that:

```
1  # Before
2  r(...,l',t') :- delay((...),t'), ... # existing logic
3  # After
4  rPending(...,l',l,t) :- ... # existing logic
```

Note that *rPending* has one more attribute than r as we do not replace the attribute bound to l' and instead add an additional attribute for l.

Persist *rPending* while the node is waiting for an ACK from C_2:

```
1 # Includes the timestep the batch is created
2 awaitingAck(l,t) :- currentBatchSize(size,l,t), size>0
3 # Conditional persist as long as the batch hasn't
    been sealed
4 awaitingAck(l,t') :- awaitingAck(l,t),
    !batchAckIn(l,t), t'=t+1
5 # If we have a batch with no ACK, block outputs
6 blockOutputs(l,t) :- awaitingAck(l,t),
    !batchAckIn(l,t)
7 # Persist blocked outputs in rPending
8 rPending(...,l,t') :- rPending(...,l,t),
    blockOutputs(l,t), t'=t+1
```

Note that `blockOutputs` and `awaitingAck` are subtly different: `blockOutputs` evaluates to false the timestep an ACK arrives, whereas `awaitingAck` only does so on the next timestep. `awaitingAck` is created to allow `blockOutputs` to unblock outputs the same timestep it receives an ACK from C_2.

Finally, create a new output channel *rSealed* that outputs persisted facts in *rPending* once the ACK has been received:

```
1 rSealed(...,l',t') :- rPending(...,l',l,t),
    !blockOutputs(l,t), delay((...),t')
```

C_1 additionally *freezes*—and delays the processing of inputs—starting the timestep after C_1 creates a new batch and prevents the sending of outputs, only unfreezing the timestep after it receives an ACK and sends the pending outputs. For each input relation r, we introduce the relations *rFrozen* and *rReleased*:

```
1 # The interval C_1 freezes is one timestep behind the
    interval it blocks outputs
2 freeze(l,t') :- blockOutputs(l,t), t'=t+1
3 # rFrozen takes inputs from r
4 rFrozen(...,l,t) :- r(...,l,t)
5 # rReleased receives facts from rFrozen when not
    frozen
6 rReleased(...,l,t) :- rFrozen(...,l,t), !freeze(l,t)
7 # Otherwise, persist facts in rFrozen
8 rFrozen(...,l,t') :- rFrozen(...,l,t), freeze(l,t),
    t'=t+1
```

To control the access to these frozen inputs by the other relations of C_1, replace all references to input relations r with `rReleased`.

6.3 Correctness

6.3.1 Linearizability. To prove the correctness of this rewrite, we will first demonstrate that it satisfies linearizability. Specifically, this means that if the commit time t_c for an operation precedes the start time for another operation t'_s, we can pick respective linearization points T and T' for these operations such that $t < t'$. We propose the following mapping to choose linearization points for messages arriving at C_1 or C_2 in our rewritten protocol.

For messages arriving at C_2, messages unrelated to batches sent from C_1 have their linearization point T mapped to the timestep of their arrival t. Messages composing a batch sent from C_1 have their linearization point T mapped to $T = t_{seal}$, the timestep the batch is fully received and processed by C_2.

For messages arriving at C_1, we divide how we map their linearization points based on the possible state of C_1 upon their arrival during timestep t:

(1) *C_1 is unfrozen and doesn't block outputs*: we map the linearization point T to be equal to t.
(2) *C_1 is unfrozen, but blocks outputs*: facts in relations referenced by C_2 must have arrived during t. Thus, the linearization point T of such messages must be mapped to $T = t_{seal}$ to coincide with C_2 and preserve linearizability.
(3) *C_1 is frozen*: this must mean that a batch was created earlier at some $t_{in} < t$. Let t_{ack} be the timestep during which C_1 receives an ACK for this batch, and $t'_{in} > t_{in}$ be the next timestep facts in relations referenced by C_2 arrive at C_1. If $t'_{in} \leq t_{ack}+1$, we map the linearization point t' to $t' = t'_{seal}$, where t'_{seal} is the timestep the relevant facts arriving at t'_{in} are sealed at C_2. Otherwise, if $t'_{in} > t_{ack} + 1$, we map t' to $T = t_{ack} + 1$.

We will begin by justifying how each chosen linearization point T falls within any possible $[t_s, t_c]$ interval of an operation in that category.

For messages arriving at C_2 unrelated to batches sent from C_1, we define t_s and t_c as the timesteps during which a third party sends a request to and receives a reply from C_2, respectively. C_2 can only receive a request during timestep t after a third party sends it during timestep $t_s < t$. Additionally, any third party will only consider an operation committed once it has received a response from C_2 during timestep t_c, who can only respond after receiving the request. As such, mapping $T = t$ satisfies $T \in [t_s, t_c]$.

Lemma 6.1. *Given an operation that is sent out and considered committed by a third party during timesteps t_s and t_c, respectively, and whose corresponding request is received and replied to by component C during timesteps t_{in} and t_{out}, respectively, $t_s < t_{in} \leq t_{out} < t_c$.*

For messages received at C_2 composing a batch sent from C_1, let t_{in} denote the timestep C_1 creates and sends the batch to C_2 and t_{ack} denote the timestep C_1 receives the corresponding ACK from C_2 and outputs. As these messages are caused by requests first arriving at C_1 from a third party, $t_s < t_{in}$. As a third party can only receive a reply from C_1 after it unblocks outputs during timestep t_{ack}, $t_c > t_{ack}$. C_2 can only start receiving forwarded messages after they are sent from C_1, and thus can only seal the batch in timestep $t_{seal} > t_{in}$. C_2 will additionally not send an ACK until the batch is sealed, meaning any ACK received by C_1 must be received at C_1 during timestep $t_{ack} > t_{seal}$. Thus, $t_s < t_{in} < t_{seal} < t_{ack} < t_c$ and so mapping $T = t_{seal}$ satisfies $T \in [t_s, t_c]$.

Lemma 6.2. *Given facts in relations referenced by C_2 arrive at C_1 during timestep t_{in}, the batch of these facts formed during t_{in} is sealed at C_2 during timestep $t_{seal} > t_{in}$.*

Lemma 6.3. *Given C_2 seals a batch during timestep t_{seal}, the ACK sent by C_2 during t_{seal} will reach C_1 during timestep $t_{ack} > t_{seal}$.*

Corollary 6.4. *Any batch created at C_1 during timestep t_{in}, sealed at C_2 during timestep t_{seal}, and whose corresponding ACK is received at C_1 during timestep t_{ack} satisfies $t_{in} < t_{seal} < t_{ack}$.*

For messages arriving at C_1 during timestep t while C_1 is unfrozen and doesn't block outputs, $t_s < t < t_c$ and thus mapping $T = t$ satisfies $T \in [t_s, t_c]$ for similar reasons as Lemma 6.1.

For messages arriving at C_1 during timestep t while C_1 is unfrozen and blocks outputs, t must represent the timestep t_{in} of a batch's creation. Therefore, $t_s < t_{in}$ and $t_{ack} < t_c$ as per Lemma 6.1. Corollary 6.4 shows that $t_{in} < t_{seal} < t_{ack}$ and thus mapping $T = t_{seal}$ satisfies $T \in [t_s, t_c]$.

For messages arriving at C_1 during timestep t while C_1 is frozen, let t_{in} denote the timestep C_1 created the corresponding batch and $t'_{in} > t_{in}$ be the next timestep facts in relations referenced by C_2 arrive at C_1. First consider the case where $t'_{in} \leq t_{ack} + 1$. As per Lemma 6.1, $t_s < t$. Since C_1 is frozen during timestep t, a batch must have been sent from C_1 during some timestep $t_{in} < t$ whose ACK has not been previously received by C_1. This implies that $t \leq t_{ack}$, with C_1 next becoming unfrozen during timestep $t_{ack} + 1$. However, $t'_{in} \leq t_{ack}+1$ dictates that C_1 will block outputs in the same timestep it unfreezes, creating and sending out another batch during timestep $t' = t_{ack} + 1$ that will be sealed by C_2 during timestep $t'_{seal} > t'$ (Corollary 6.4). The outputs for messages arriving during timestep t will thus be sent during timestep $t'_{ack} > t'_{seal}$ (Corollary 6.4), with $t'_{ack} < t_c$ (Lemma 6.1). As a result, $t_s < t \leq t_{ack} < t' < t'_{seal} < t'_{ack} < t_c$ and thus mapping $T = t'_{seal}$ satisfies $T \in [t_s, t_c]$. Note here that $T \geq t$. If instead $t'_{in} > t_{ack} + 1$, no batch is created at C_1 as it unfreezes during timestep $t_{ack} + 1$. This means C_1 will not block outputs, allowing C_1 to respond during timestep $t_{ack} + 1 < t_c$ (Lemma 6.1). As such, $t_s < t < t_{ack} + 1 < t_c$ and thus mapping $T = t_{ack} + 1$ satisfies $T \in [t_s, t_c]$.

Lemma 6.5. *Given an operation that is considered started and committed by a third party during timesteps t_s and t_c, respectively, the assigned linearization point to that operation T must satisfy $T \in [t_s, t_c]$.*

Corollary 6.6. *Given any message arriving at either C_1 or C_2 during timestep t, the message's corresponding linearization point is mapped to $T \geq t$.*

Corollary 6.6 is an observation made about our linearization mapping that will aid in some of our proofs.

To prove our mapping satisfies linearizability, we will assume by contradiction that there exists two operations spanning intervals $[t_{s,1}, t_{c,1}]$ and $[t_{s,2}, t_{c,2}]$ such that $t_{c,1} < t_{s,2}$ but their corresponding linearization points violate linearizability ($T_2 < T_1$). Lemma 6.5 states that $T_1 \in [t_{s,1}, t_{c,1}]$ and $T_2 \in [t_{s,2}, t_{c,2}]$. However, as $t_{c,1} < t_{s,2}$, this implies that $T_1 \leq t_{c,1} < t_{s,2} \leq T_2$, contradicting $T_2 < T_1$.

6.3.2 Order Consistency. We will now prove that any set of outputs produced as a result of executing the rewritten protocol can be produced by some execution of the original protocol. For this portion of the proof, we borrow notation used by Chu et al. in [10]. Since C in our original protocol is a state machine [11], as are C_1 and C_2 in our rewritten protocol, we know that C_1 and C_2 will collectively produce the same outputs as C if they all process the same inputs in the same order.

Consider the facts at C_2 in relations $\overline{rSealed}$ during t_{seal}, the timestep during which C_2 seals a batch. These facts correspond 1:1 to facts at C_2 in relations $\overline{r}$ in the original protocol, had it simultaneously received the same deltas and facts at t_{seal}. As a result, there also exists a 1:1 correspondence between facts outputted by C_2 in the rewritten and original protocols during such a t_{seal}. We now argue that the facts outputted by C_1 in our rewritten protocol correspond to an execution of the original protocol had the same deltas and facts also simultaneously arrived at and been instantaneously processed by C at time t_{seal}, thus proving that our execution of the rewritten protocol corresponds to an execution of the original, single-node protocol receiving and processing the same inputs instantaneously.

As such, we will propose a 1:1 mapping of processing times between facts arriving at C_1 in an execution of our rewritten protocol and facts arriving at a unified C in an execution of the original protocol. We claim that this mapping preserves total order—that is, any two facts are processed at C_1 during timesteps $\pi_T(f_1) < \pi_T(f_2)$ in our rewritten protocol if and only if our mapping implies they were processed at C during timesteps $\pi_T(f'_1) < \pi_T(f'_2)$ in the execution of the original protocol. Additionally, any two facts are processed at C_1 during timesteps $\pi_T(f_1) = \pi_T(f_2)$ in our rewritten protocol if and only if our mapping implies they were processed at C during timesteps $\pi_T(f'_1) = \pi_T(f'_2)$ in the execution of the original protocol. As such, the execution of the original protocol we construct from an execution of the rewritten protocol must have C process the same facts in the same order and thus produce the same outputs. Note that as C_2 is independent from C_1, we only need to prove that this property holds for facts arriving at C_1.

Our mapping is simple: for any fact processed by C_1 in the rewritten protocol, we map its corresponding processing time under the original protocol to equal the linearization point of the operation it represents.

Coinciding Facts. We will first prove that $\pi_T(f'_1) = \pi_T(f'_2)$ if and only if $\pi_T(f_1) = \pi_T(f_2)$ in our execution of the rewritten protocol.

We begin by proving that $\pi_T(f'_1) = \pi_T(f'_2)$ if $\pi_T(f_1) = \pi_T(f_2)$ in our execution of the rewritten protocol: we refer to the possible states of C_1 in our rewritten protocol during any given timestep. As facts are only processed at C_1 when it is unfrozen, this means facts are processed either (1) in the same timestep C_1 blocks outputs or (2) in a timestep during which C_1 doesn't block outputs. Note that C_1 can only be in at most one of these states during any given timestep.

In the first case, $\pi_T(f_1) = \pi_T(f_2) = t_{in}$. These facts must have each arrived at C_1 either during timestep t_{in} or while C_1 was frozen for the previous batch during the interval $[t_{in,prev}+1, t_{ack,prev}]$, where $t'_{in,prev} \leq t_{ack,prev} + 1$. Regardless, our mapping dictates that these facts are processed at C_1 during $\pi_T(f'_1) = \pi_T(f'_2) = t_{seal}$ in the

original protocol, where t_{seal} denotes the timestep during which the batch corresponding to t_{in} is sealed at C_2.

Lemma 6.7. *Given C_1 in our rewritten protocol processes facts in relations referenced by C_2 during timestep $t = t_{in}$, all facts processed during timestep t will be processed by C_1 in the original protocol during timestep t_{seal}.*

In the second case, facts processed during $\pi_T(f_1) = \pi_T(f_2)$ also arrived in the same timestep. Our mapping therefore dictates that these facts arrive (and are instantaneously processed) at C_1 during $\pi_T(f_1') = \pi_T(f_1)$ and $\pi_T(f_2') = \pi_T(f_2)$ in the original protocol. Thus, $\pi_T(f_1') = \pi_T(f_2')$ because $\pi_T(f_1) = \pi_T(f_2)$.

We now prove that $\pi_T(f_1') = \pi_T(f_2')$ only if $\pi_T(f_1) = \pi_T(f_2)$ in our execution of the rewritten protocol. Let timestep t' represent a timestep during which facts for relations in C_1 are received, instantaneously processed, and immediately outputted from C in the original protocol. During any given t', C may or may not receive facts in relations referenced by C_2.

Let $t' = \pi_T(f_1') = \pi_T(f_2')$. If t' represents such a timestep during which no facts arrive at C in relations referenced by C_2, we claim that $\pi_T(f_1) = \pi_T(f_2)$ in the rewritten protocol. To do this, assume for the sake of contradiction that we have some $\pi_T(f_1) \neq \pi_T(f_2)$ in the rewritten protocol that corresponds to $\pi_T(f_1') = t'$ and $\pi_T(f_2') = t'$ in our execution of the original protocol.

We review all possible ways for a fact arriving during timestep t in the rewritten protocol to have its corresponding linearization point (and thus its processing time) T mapped to t' in the original:

(1) If t represents a timestep during which C_1 is unfrozen and does not block outputs, T is mapped to t and thus $t' = t$.
(2) If t represents a timestep during which C_1 is unfrozen but blocks outputs, this implies that facts in relations referenced by C_2 have been processed during the same timestep. Lemma 6.7 dictates that these facts will be mapped to be processed under the original protocol during timestep $t' = T$, causing a contradiction.
(3) If t represents a timestep during which C_1 is frozen, T is mapped to t'_{seal} if $t'_{in} \leq t_{ack} + 1$ and $t_{ack} + 1$ otherwise. For similar reasons as above, the first case poses a contradiction—some facts arriving during timestep t'_{in} must have been for relations referenced by C_2. Thus, this case is only possible in the event $t'_{in} > t_{ack} + 1$.

If we assume both that $\pi_T(f_1') = \pi_T(f_2')$ and $\pi_T(f_1) \neq \pi_T(f_2)$, then it poses a contradiction to have the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ both fall into the first category. This would mean that both facts arrived (and were thus instantaneously processed) during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$. As such, $\pi_T(f_1') = \pi_T(f_1)$ and $\pi_T(f_2') = \pi_T(f_2)$. However, since $\pi_T(f_1) \neq \pi_T(f_2)$, this implies $\pi_T(f_1') = \pi_T(f_1) \neq \pi_T(f_2) = \pi_T(f_2')$ and thus the contradiction that $\pi_T(f_1') \neq \pi_T(f_2')$.

Having both facts instead fall into third category also poses a contradiction. If the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ arrived at C_1 during the same period of freezing—that is, there is some batch such that $t_{in} < \pi_T(f_1) < t_{ack}+1$ and $t_{in} < \pi_T(f_2) < t_{ack}+1$—then both facts would've been processed by C_1 in the rewritten protocol during timestep $t_{ack} + 1$ (as it must be the case that $t'_{in} > t_{ack} + 1$), implying $\pi_T(f_1) = \pi_T(f_2)$. If the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ arrived at C_1 during different periods of freezing, then without loss of generality assume that $\pi_T(f_1) < \pi_T(f_2)$. Therefore, $t_{in,1} < \pi_T(f_1) < t_{ack,1} + 1$ and $t_{in,2} < \pi_T(f_2) < t_{ack,2} + 1$, where $t_{in,1}$ and $t_{ack,1}$ correspond to one batch and $t_{in,2}$ and $t_{ack,2}$ correspond to another. Therefore, our mapping would've dictated these facts be processed at C_1 during $\pi_T(f_1') = t_{ack,1} + 1$ and $\pi_T(f_2') = t_{ack,2} + 1$ in our original protocol. As C_1 processes batches serially, $t_{ack,1} < t_{in,2}$. This means $t_{ack,1} < t_{in,2}$, implying $\pi_T(f_1') = t_{ack,1} + 1 \leq t_{in,2} < t_{ack,2} + 1 = \pi_T(f_2')$ and thus $\pi_T(f_1') \neq \pi_T(f_2')$.

Lemma 6.8. *Given timesteps $\pi_T(f_1) < \pi_T(f_2)$ correspond to distinct periods during which C_1 is awaiting an ACK for a batch, there exists a $t_{in,1}$, $t_{ack,1}$, $t_{in,2}$, and $t_{ack,2}$ such that $t_{in,1} \leq \pi_T(f_1) < t_{ack,1} + 1$, $t_{in,2} \leq \pi_T(f_2) < t_{ack,2} + 1$, and $t_{ack,1} < t_{in,2}$.*

Finally, both facts distinctly falling into the first and third categories also leads to a contradiction. Without loss of generality, assume that the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ correspond to arrival times falling into the first and third categories, respectively, and that $\pi_T(f_1) < \pi_T(f_2)$. The fact processed during timestep $\pi_T(f_2)$ must have arrived during the interval $[t_{in}+1, t_{ack}]$. As such, C_1 in the rewritten protocol would have processed this fact during timestep $\pi_T(f_2) = t_{ack} + 1$; additionally, our mapping dictates that this fact is processed under the original protocol during timestep $\pi_T(f_2') = t_{ack} + 1$. Meanwhile, $\pi_T(f_1') = \pi_T(f_1)$. However, this would imply that $\pi_T(f_1') = \pi_T(f_1) < \pi_T(f_2) = t_{ack} + 1 = \pi_T(f_2')$, forming a contradiction. Now assume that $\pi_T(f_1) > \pi_T(f_2)$. Similarly, $\pi_T(f_1') = \pi_T(f_1)$, $\pi_T(f_2) = t_{ack} + 1$, and $\pi_T(f_2') = t_{ack} + 1$. However, this implies that $\pi_T(f_1) = \pi_T(f_1') = \pi_T(f_2') = t_{ack} + 1 = \pi_T(f_2)$, another contradiction.

Once again, let $t' = \pi_T(f_1') = \pi_T(f_2')$. Now, assume timestep t' represents a timestep during which facts arrive at C in relations referenced by C_2. We claim that it is still the case that $\pi_T(f_1) = \pi_T(f_2)$ in our execution of the rewritten protocol.

There are two ways for a fact arriving during timestep t in the rewritten protocol to have its processing time mapped to t' in the original: either (1) t represents a timestep during which C_1 is unfrozen but blocking outputs, or (2) t represents a timestep during which C_1 is frozen and $t'_{in} \leq t_{ack} + 1$. Note this is the complement of the cases mentioned above in the event timestep t' represents a timestep during which no facts arrive at C in relations referenced by C_2.

If we assume that $\pi_T(f_1') = \pi_T(f_2')$ and $\pi_T(f_1) \neq \pi_T(f_2)$, then it poses a contradiction to have the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ both fall into the first category. Without loss of generality, let $\pi_T(f_1) < \pi_T(f_2)$. Recall that facts falling into the first category are processed in the same timestep of their arrival. Since $\pi_T(f_1) \neq \pi_T(f_2)$, $\pi_T(f_1) = t_{in,1}$ and $\pi_T(f_2) = t_{in,2}$, where $t_{in,1}$ and $t_{in,2}$ correspond to distinct batches. Furthermore, our mapping dictates $\pi_T(f_1') = t_{seal,1}$ and $\pi_T(f_2') = t_{seal,2}$, where $t_{seal,1}$ and

$t_{seal,2}$ correspond to the batches defining $t_{in,1}$ and $t_{in,2}$, respectively. Additionally, Lemma 6.8 dictates that $t_{ack,1} < t_{in,2}$. However, since $t_{seal,1} < t_{ack,1} < t_{in,2} < t_{seal,2}$, this implies $\pi_T(f_1') = t_{seal,1} < t_{seal,2} = \pi_T(f_2')$ and thus the contradiction that $\pi_T(f_1') \neq \pi_T(f_2')$.

Having both facts instead fall into the second category also poses a contradiction. If both facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ arrived at C_1 during the same period of freezing, then our mapping dictates both facts would've been processed by C_1 during timestep t'_{seal} (as it must be the case that $t'_{in} \leq t_{ack} + 1$), implying $\pi_T(f_1) = \pi_T(f_2)$. If the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ arrived at C_1 during different periods of freezing, then without loss of generality assume that $\pi_T(f_1) < \pi_T(f_2)$. Therefore, the fact corresponding to $\pi_T(f_1)$ would have arrived during the interval $[t_{in,1} + 1, t_{ack,1}]$ and likewise $[t_{in,2} + 1, t_{ack,2}]$ for the fact corresponding to $\pi_T(f_2)$, where $t_{in,1}$ and $t_{ack,1}$ correspond to one batch and $t_{in,2}$ and $t_{ack,2}$ correspond to another. Recall that it must be the case that $t'_{in,1} \leq t_{ack,1} + 1$ and $t'_{in,2} \leq t_{ack,2} + 1$. Therefore, our mapping would've dictated these facts be processed at C_1 during $\pi_T(f_1') = t'_{seal,1}$ and $\pi_T(f_2') = t'_{seal,2}$ in our original protocol, where $t'_{seal,1}$ corresponds to the batch including facts received during timestep $t'_{in,1}$ (and similarly for $t'_{seal,2}$). It cannot be the case that $t'_{seal,1} = t'_{seal,2}$; this would require $t'_{in,2} \leq t_{ack,1} + 1$. This is impossible, since by definition $t_{in,2} < t'_{in,2}$ and due to Lemma 6.8 $t_{ack,1} < t_{in,2}$. As such, $t'_{in,1} \leq t_{ack,1}+1 < t'_{in,2}$ and thus $t'_{in,1}$ and $t'_{in,2}$ must correspond to different batches. This means $t'_{seal,1} \neq t'_{seal,2}$, implying $\pi_T(f_1') = t'_{seal,1} \neq t'_{seal,2} = \pi_T(f_2')$. Therefore, we reach the contradiction that $\pi_T(f_1') \neq \pi_T(f_2')$.

Finally, both facts distinctly falling into the first and second categories also leads to a contradiction. Without loss of generality, assume that the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ have arrival times falling into the first and second categories, respectively. Now assume that $\pi_T(f_1) < \pi_T(f_2)$. The fact processed during timestep $\pi_T(f_1)$ must have arrived (and been instantaneously processed) during the $t_{in,1}$ of some batch such that $\pi_T(f_1) = t_{in,1} < t_{ack,1}$. As such, our mapping dictates that C_1 in the rewritten protocol processes this fact during timestep $\pi_T(f_1') = t_{seal,1}$. If the other fact arrives at C_1 during a timestep before $t_{ack,1}$, then as $t'_{in,1} \leq t_{ack,1}+1$, $\pi_T(f_2) = t_{ack,1}+1$. Our mapping dictates that facts arriving during the interval $[t_{in,1} + 1, t_{ack,1}]$ at C_1 in the rewritten protocol are processed by C_1 in the original protocol during timestep $\pi_T(f_2') = t'_{seal,1}$ (assuming $t'_{in,1} \leq t_{ack,1} + 1$). However, as $t_{seal,1}$ and $t'_{seal,1}$ correspond to different batches, this implies $\pi_T(f_1') = t_{seal,1} \neq t'_{seal,1} = \pi_T(f_2')$ and thus the contradiction that $\pi_T(f_1') \neq \pi_T(f_2')$. If instead the other fact arrives during or after timestep $t_{ack,1}+1$, then as our mapping guarantees $\pi_T(f_2')$ is at least $t_{ack,1}+1$ by Corollary 6.6, we have $\pi_T(f_1) < t_{ack,1}+1 \leq \pi_T(f_2')$. This implies $\pi_T(f_1') \neq \pi_T(f_2')$, a contradiction. Now assume that $\pi_T(f_1) > \pi_T(f_2)$. Similarly, the fact processed during timestep $\pi_T(f_1)$ must have arrived (and been simultaneously processed) during timestep $t_{in,1}$ for some batch—thus, $\pi_T(f_1') = t_{seal,1}$. As the fact corresponding to $\pi_T(f_1)$ also arrived at C_1 during timestep $\pi_T(f_1)$, then $\pi_T(f_1) > \pi_T(f_2)$ implies that that the fact corresponding to $\pi_T(f_2)$ must have been processed by C_1 before the fact corresponding to $\pi_T(f_1)$ arrived. Thus, the fact corresponding to $\pi_T(f_2)$ arrived before the fact corresponding to $\pi_T(f_1)$. As $\pi_T(f_1)$ represents a timestep during which C_1 has created a batch and the fact corresponding to $\pi_T(f_2)$ arrived while C_1 was frozen, it must be the case that the facts corresponding to timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ must have arrived while C_1 was awaiting ACKs for different batches. Thus, as $\pi_T(f_2) < \pi_T(f_1)$, there must be some $t_{ack,2}$ such that the fact corresponding to $\pi_T(f_2)$ arrived during the interval $[t_{in,2}+1, t_{ack,2}]$, where $t_{ack,2} < t_{in,1}$ as per Lemma 6.8. Recall that it must be the case that $t'_{in,2} \leq t_{ack,2}+1$, leaving two possibilities. It is possible the batch following the one corresponding to $\pi_T(f_2)$ is the batch corresponding to $\pi_T(f_1)$, meaning the facts arriving during timestep $t'_{in,2}$ are processed during timestep $t_{in,1}$. It is also possible the batch following the one corresponding to $\pi_T(f_2)$ is unrelated to the batch corresponding to $\pi_T(f_1)$, meaning the facts arriving during timestep $t'_{in,2}$ are processed during a timestep earlier than $t_{in,1}$. If the first case is true, this means that the fact corresponding to $\pi_T(f_2)$ is processed during timestep $\pi_T(f_2) = t_{in,1}$. However, this implies the contradiction that $\pi_T(f_1) = \pi_T(f_2)$. If the second case is true, this means that the fact corresponding to $\pi_T(f_2)$ is mapped to be processed during timestep $\pi_T(f_2') = t'_{seal,2}$. However, as $t'_{seal,2} \neq t_{seal,1}$, this implies $\pi_T(f_2') = t'_{seal,2} \neq t_{seal,1} = \pi_T(f_1')$ and thus the contradiction that $\pi_T(f_1') \neq \pi_T(f_2')$.

Lemma 6.9. *Given two facts are processed by C_1 during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ in the rewritten protocol and processed during timesteps $\pi_T(f_1')$ and $\pi_T(f_2')$ in the original protocol, $\pi_T(f_1') = \pi_T(f_2')$ if and only if $\pi_T(f_1) = \pi_T(f_2)$.*

Non-Coinciding Facts. We will now prove that $\pi_T(f_1') < \pi_T(f_2')$ if and only if $\pi_T(f_1) < \pi_T(f_2)$ in our execution of the rewritten protocol.

We begin by proving that $\pi_T(f_1') < \pi_T(f_2')$ if $\pi_T(f_1) < \pi_T(f_2)$ in our execution of the rewritten protocol. Similarly to our proof of Lemma 6.9, we identify that C_1 either (1) blocks outputs or (2) does not block outputs as it processes any given fact.

If the facts processed during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ both fall into the first category, then $\pi_T(f_1)$ and $\pi_T(f_2)$ correspond to the creation of different batches. As $\pi_T(f_1) = t_{in,1}$ for some batch and likewise $\pi_T(f_2) = t_{in,2}$, following similar reasoning from our proof for Lemma 6.9 shows that $\pi_T(f_1') = t_{seal,1}$ and $\pi_T(f_2') = t_{seal,2}$. However, Lemma 6.8 shows that $t_{ack,1} < t_{in,2}$. Thus, Corollary 6.4 lets us show that $\pi_T(f_1') = t_{seal,1} < t_{ack,1} < t_{in,2} < t_{seal,2} = \pi_T(f_2')$, implying that $\pi_T(f_1') < \pi_T(f_2')$.

Now, if the fact corresponding to $\pi_T(f_2)$ instead falls into second category, then the fact processed during timestep $\pi_T(f_2)$ must have arrived during the same timestep. Thus, our mapping dictates that $\pi_T(f_2') = \pi_T(f_2)$. However, as $\pi_T(f_1)$ represents a timestep during which a batch is created at C_1 and $\pi_T(f_2)$ represents a timestep during which C_1 is unfrozen, $\pi_T(f_1) < \pi_T(f_2)$ means that $t_{ack,1} < \pi_T(f_2)$. Therefore, Corollary 6.4 allows us to show that $\pi_T(f_1') = t_{seal,1} < t_{ack,1} < \pi_T(f_2) = \pi_T(f_2')$ and thus $\pi_T(f_1') < \pi_T(f_2')$.

If instead both facts fall into the second category, then $\pi_T(f_1)$ and $\pi_T(f_2)$ must have arrived during the same timestep. Thus, our mapping dictates that $\pi_T(f_1') = \pi_T(f_1)$ and $\pi_T(f_2') = \pi_T(f_2)$, implying $t_{1'} = \pi_T(f_1) < \pi_T(f_2) = \pi_T(f_2')$ and thus that $\pi_T(f_1') < \pi_T(f_2')$.

If now the fact corresponding to $\pi_T(f_2)$ instead falls into the first category, then $\pi_T(f_2) = t_{in}$ for some batch created at C_1. As $\pi_T(f_1') = \pi_T(f_1)$ and $\pi_T(f_1) < \pi_T(f_2)$, Lemma 6.7 dictates that $\pi_T(f_2') = t_{seal}$ and thus by Corollary 6.4 $\pi_T(f_1') = \pi_T(f_1) < \pi_T(f_2) = t_{in} < t_{seal} = \pi_T(f_2')$. Therefore, $\pi_T(f_1') < \pi_T(f_2')$.

We now prove that $\pi_T(f_1') < \pi_T(f_2')$ only if $\pi_T(f_1) < \pi_T(f_2)$ in our execution of the rewritten protocol. Assume for the sake of contradiction that $\pi_T(f_1') < \pi_T(f_2')$ but $\pi_T(f_1) \geq \pi_T(f_2)$. As $\pi_T(f_1') \neq \pi_T(f_2')$, Lemma 6.9 prevents $\pi_T(f_1) = \pi_T(f_2)$. Thus, we only proceed assuming $\pi_T(f_1) > \pi_T(f_2)$.

Recall that during any given timestep, C_1 either (1) blocks outputs or (2) does not block outputs as it processes any given fact.

Say $\pi_T(f_1)$ and $\pi_T(f_2)$ both represent timesteps during which C_1 blocks outputs. As $\pi_T(f_1) > \pi_T(f_2)$, each timestep represents the creation of a different batch at C_1. By Lemma 6.8, this means there exists some $t_{in,2}$, $t_{ack,2}$, $t_{in,1}$, and $t_{ack,1}$ such that $t_{in,2} \leq \pi_T(f_2) < t_{ack,2} + 1$, $t_{in,1} \leq \pi_T(f_1) < t_{ack,1} + 1$, and $t_{ack,2} < t_{in,1}$. Note that in this particular case, the roles of $\pi_T(f_1)$ and $\pi_T(f_2)$ are flipped in the context of Lemma 6.8. Since Lemma 6.7 dictates that $\pi_T(f_2') = t_{seal,2}$ and $\pi_T(f_1') = t_{seal,1}$, this and Corollary 6.4 implies that $\pi_T(f_2') = t_{seal,2} < t_{ack,2} < t_{in,1} < t_{seal,1} = \pi_T(f_1')$ and thus the contradiction that $\pi_T(f_1') > \pi_T(f_2')$.

If now $\pi_T(f_2)$ represents a timestep during which C_1 does not block outputs, we claim that this poses a contradiction. Since $\pi_T(f_1)$ represents a timestep during which C_1 creates a batch, Lemma 6.7 dictates that $\pi_T(f_1') = t_{seal}$, where t_{seal} corresponds to the same batch created during timestep $\pi_T(f_1) = t_{in}$. As $\pi_T(f_2)$ represents a timestep during which C_1 is not awaiting an ACK for any batch, we know the fact processed during $\pi_T(f_2)$ additionally arrived during the same timestep. As such, $\pi_T(f_2') = \pi_T(f_2)$ and thus $\pi_T(f_1') = t_{seal} > t_{in} = \pi_T(f_1) > \pi_T(f_2) = \pi_T(f_2')$. This implies the contradiction that $\pi_T(f_1') > \pi_T(f_2')$.

Say $\pi_T(f_1)$ and $\pi_T(f_2)$ both represent timesteps during which C_1 does not block outputs. This means $\pi_T(f_1)$ and $\pi_T(f_2)$ must have arrived during the same timestep. Thus, our mapping dictates that $\pi_T(f_1') = \pi_T(f_1)$ and $\pi_T(f_2') = \pi_T(f_2)$, implying $t_{1'} = \pi_T(f_1) > \pi_T(f_2) = \pi_T(f_2')$ and thus the contradiction that $\pi_T(f_1') > \pi_T(f_2')$.

If rather $\pi_T(f_1)$ and $\pi_T(f_2)$ represent timesteps during which C_1 does not block outputs and blocks outputs, respectively, we claim that this also poses a contradiction. As $\pi_T(f_2)$ represents the timestep during which a batch is created, $\pi_T(f_1)$ represents a timestep during which C_1 does not block outputs, and $\pi_T(f_1) > \pi_T(f_2)$, there must exist some t_{ack} such that $\pi_T(f_2) = t_{in} < t_{ack} < \pi_T(f_1)$ (Corollary 6.4). Additionally, $\pi_T(f_1') = \pi_T(f_1)$ following similar reasoning as above and Lemma 6.7 dictates that $\pi_T(f_2') = t_{seal}$. This implies that $\pi_T(f_2') = t_{seal} < t_{ack} < \pi_T(f_1) = \pi_T(f_1')$ and thus the contradiction that $\pi_T(f_1') > \pi_T(f_2')$.

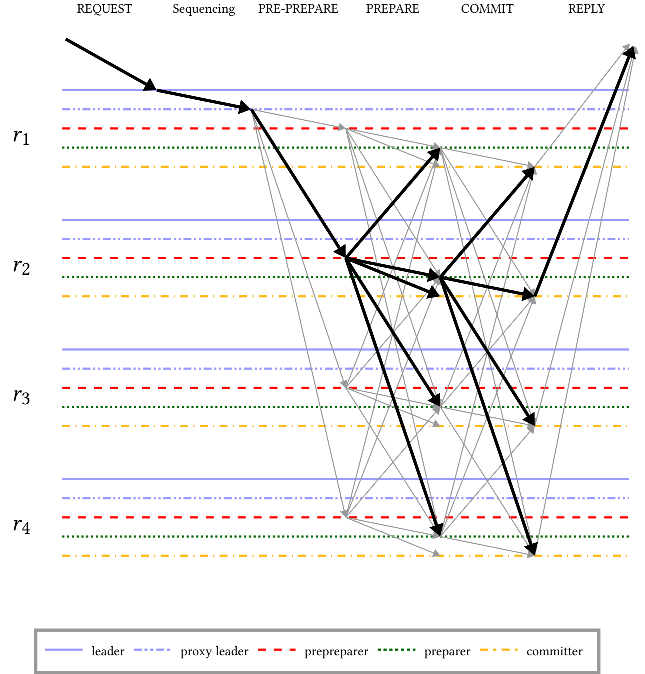


Figure 1: The critical path of ScalablePBFT, with the message path through r_2 bolded. r_1 is the primary.

Lemma 6.10. *Given two facts are processed by C_1 during timesteps $\pi_T(f_1)$ and $\pi_T(f_2)$ in the rewritten protocol and processed during timesteps $\pi_T(f_1')$ and $\pi_T(f_2')$ in the original protocol, $\pi_T(f_1') < \pi_T(f_2')$ if and only if $\pi_T(f_1) < \pi_T(f_2)$.*

Thus, because C is a state machine and our rewrite maintains the total order of inputs, C under the rewritten protocol must output the same outputs as C under our proposed execution of the original protocol.

7 EVALUATION

Chu et al. evaluates the effectiveness of rewrites in [9] by applying them by hand to the critical path of PBFT [8]. In these evaluations, Chu et al. compares a rewritten version of PBFT's critical path, decoupled and partitioned across multiple nodes, with an unmodified implementation of PBFT's critical path run on a single node.

Their experiment setup follows that of Chu et al. [11] and be found at https://github.com/rithvikp/autocomp [10]. The Dedalus [4] implementations of PBFT's critical path and its rewritten version are compiled to Hydroflow [25] (a Rust dataflow runtime for distributed systems) and deployed on Google Cloud. Each node runs on 2-standard-4 machines with 4vCPUs, 16 GB RAM, and 10 Gbps network bandwidth. State machines, clients, and replicas are all run on separate machines. Throughput and latency metrics are measured over one minute runs, following 30 seconds of warmup. Performance is measured by continually increasing a set of active clients—each of which sends 16 byte, unbatched commands in a closed loop—until throughput saturates.

Henceforth, we will refer to the standard implementation of PBFT's critical path as **BasePBFT** and the rewritten implementation as **ScalablePBFT**.

Chu et al. runs evaluations on deployments tolerating $f = 1$ failures and measures performance on the critical path, assuming no failures in the system.

The performance of ScalablePBFT configured to replicate applicable components across 1, 3, and 5 partitions was measured by Chu et al. as part of the evaluations. The deployment of ScalablePBFT partitioned 3 ways achieves a maximum throughput of 55,000 commands/s [9], 5× that of BasePBFT [9]. Latency did not noticeably increase as a result of the rewrites, and partitioning by a factor larger than 5 did not significantly improve throughput in the experiments [9].

7.1 Construction

The deployment of BasePBFT runs on $n = 3f + 1$ nodes as described in Section 2.2. Replicas, upon committing a request, send client requests to their corresponding state machine nodes for execution; state machines respond directly to the client. Chu et al. creates an optimized version of the protocol by manually applying the *mutually independent decoupling*, *functional decoupling*, *monotonic decoupling*, and *partitioning with co-hashing* rewrites [9] to the critical path of BasePBFT. Under these rewrites, each replica is decoupled into *leader*, *proxy leader*, *prepreparer*, *preparer*, and *committer* components. Each component, excluding the leader, is hash partitioned on the sequence number of the request.

Clients send REQUEST messages to the leader component corresponding to the replica they believe to be the current leader. The leader component sequences the request and forwards it to their corresponding proxy leader, which broadcasts PRE-PREPARE messages (alongside the corresponding REQUEST message) to all prepreparer components in the system. Prepreparers, upon accepting a valid PRE-PREPARE message, broadcast PREPARE messages to all preparers in the system. Prepreparers additionally forward the associated REQUEST message to their corresponding committer to allow it to execute the client request once it is committed. Preparers accept PREPARE messages from prepreparers and broadcast COMMIT messages to committers upon receiving a quorum of $2f + 1$ matching PREPARE messages. Committers accept COMMIT messages from preparers, sending the client request alongside its sequence number to their corresponding state machine after receiving a quorum of $2f + 1$ matching COMMIT messages. The state machine then sends a REPLY message to the client, who accepts the result upon receiving a weak quorum of $f + 1$ matching REPLY messages. A visual representation of this process is illustrated in Figure 1.

We additionally apply the *partial independent decoupling* rewrite to support additional functionality off the critical path, proposing an extended version of ScalablePBFT capable of view changes and stable checkpoint garbage collection. The Dedalus implementation of PBFT before and after our rewrites can be found at https://github.com/git-doge/dedalus-implementations.

To create our rewritten version of PBFT, we first designate a component of the protocol to be the *state manager*. The state manager maintains the replica's current view, low watermark, and a list of views that it is currently attempting a view change into. The state manager is additionally responsible for creating and reaching consensus on checkpoints, collecting VIEW-CHANGE messages, as well as collecting and creating NEW-VIEW messages. The state manager is decoupled from the rest of the protocol using *partial independent decoupling*; changes to the current view, low watermark, and the list of current view change attempts prompt the state manager to freeze and are forwarded to the rest of the protocol as described in the rewrite. Stable checkpoint proofs and PRE-PREPARE messages from NEW-VIEW messages also similarly require a freeze and are forwarded to the rest of the protocol as described.

We then divide the other component of the protocol in two: the critical path and a component we will call the *view-change collector*. The view-change collector is responsible for signing over the collection of PREPARE and CHECKPOINT messages required to create a VIEW-CHANGE message. The view-change collector additionally broadcasts the signed VIEW-CHANGE messages to all state managers in the system upon attempting a view change. We use state machine decoupling [10] to decouple the view-change collector from the critical path; the view-change collector thus only accepts inputs in batches sent from the critical path component.

The critical path component contains the logic to handle REQUEST, PRE-PREPARE, PREPARE, and COMMIT messages—it additionally contains logic to send the view-change collector the appropriate PREPARE and CHECKPOINT messages when a view change is triggered. Similarly to Chu et al., we decouple the different stages of the critical path from each other into the *leader*, *prepreparer*, *preparer*, and *committer* components [9]. Each component handles the logic for its corresponding messages, and the prepreparer additionally forwards client requests from the committer [9]. However, we now decouple these components from each other using partial independent decoupling; changes to the current view, low watermark, and the list of ongoing view change attempts are thus propagated through all components by a cascading series of freezes. The preparer component is designated as the one to send its corresponding view-change collector the appropriate PREPARE and CHECKPOINT messages upon learning a view change has been triggered—as such, it receives the stable checkpoint proofs forwarded from the state manager. The leader component similarly receives the PRE-PREPARE messages (originating from NEW-VIEW messages) forwarded by the state manager. Note that messages sent on the critical path do not require freezing any components.

As with Chu et al., we additionally decouple the *proxy leader* component from the rest of the leader component [9]. The proxy leader is responsible for broadcasting PRE-PREPARE messages sent to it by the leader; the leader now only sequences incoming REQUEST messages. The proxy leader is decoupled using functional decoupling [11]. We also decouple the *state machine* from the committer, responsible for executing requests in order, replying to clients, and overwriting state with snapshots when applicable. The state machine is decoupled from the committer via monotonic decoupling [11].

We can now use partial partitioning to partition the prepreparer, preparer, and committer–these components are hash-partitioned on the sequence number of the request, and changes to values shared with the state manager are thus replicated across all partitions as described in [10]. We additionally use partitioning with co-hashing [11] to partition the proxy leader.

When a view change is triggered, the state manager updates the list of ongoing view change attempts to include the new attempt. This list is relevant to the components on the critical path, as a non-empty list signifies that replica should no longer accept consensus messages. Additionally, the addition of a new view change attempt prompts the preparer to send the relevant PREPARE and CHECKPOINT messages to the view-change collector. There are several possible ways a view change can be triggered:

(1) *Request Timeout*: in PBFT, a replica attempts a view change into the next view if it has a client request that has taken too long to commit in the current view. Specifically, the committer starts a timer upon receiving a client request and stops it upon receiving $2f + 1$ corresponding COMMIT messages for the request, attempting a view change if it expires before then. In our rewritten ScalablePBFT, the timer is started by the committer, who contacts the state manager if it expires. Note that we implement this timer such that it notifies itself asynchronously upon expiring—this prevents the state manager from being dependent on the critical path, but may have ramifications on the liveness of the system.
(2) *View Change Timeout*: in PBFT, a replica can also attempt another view change if an ongoing view change attempt has taken too long to complete. Specifically, a replica starts a timer upon receiving $2f + 1$ VIEW-CHANGE messages for a given view. If the timer expires before a corresponding NEW-VIEW message arrives, the replica attempts a view change into the next higher view. In ScalablePBFT, this logic is handled by the state manager, which updates the list of ongoing view change attempts upon the timer's expiration.
(3) *View Change Quorum*: in PBFT, a replica additionally attempts a view change if it ever receives $f + 1$ VIEW-CHANGE messages for higher views. In ScalablePBFT, this logic is also handled by the state manager.

There additionally may be cases in which replicas may need to request client request or state machine snapshot from its peers:

(1) *Client Requests*: in PBFT, a replica requests a client request from its peers if it has $2f + 1$ COMMIT messages for a digest, but not the client request corresponding to the digest; it does this by contacting the replicas participating in the COMMIT quorum. In our rewritten ScalablePBFT, this logic is handled by the commmitter.
(2) *Snapshots*: in PBFT, a replica requests a snapshot from its peers if it has a stable checkpoint but not the snapshot corresponding to its digest; it does this by contacting the replicas participating in the checkpoint's proof of stability. In our rewritten ScalablePBFT, this logic is handled by the state manager.

8 CONCLUSION

This work demonstrates how the capabilities of generalized, local protocol optimizations shown to scale throughput in CFT environments [11] can similarly be applied to BFT systems, with only minor modifications. We additionally introduce a new partial independent decoupling rewrite to adapt to common design paradigms found in many BFT protocols, further expanding the scope and flexibility of our suite of rewrite rules.

This work builds on the contributions of Chu et al. [11], which motivates automatic rewrites as a means to effectively scale distributed protocols without the need to create clever insights or partake in ad-hoc reasoning of correctness. Our work here demonstrates how these principles pose interesting challenges to safety in the face of Byzantine actors, and hints at a future where optimized compilers for distributed protocols can exist just as they do for single-node programs. It additionally follows that other modifications to our rewrites may be possible to further expand the domain of fault models under which they can be performed.

ACKNOWLEDGMENTS

To Natacha Crooks, for advising me throughout my research. To David Chu, for his constant support, encouragement, and advice—and for introducing me to the joys of distributed systems. To Joseph M. Hellerstein, Shadaj Laddad, and other members of the wonderful Hydro team for their additional support. This work was supported by gifts from AMD, Anyscale, Google, IBM, Intel, Microsoft, Mohamed Bin Zayed University of Artificial Intelligence, Samsung SDS, Uber, and VMware.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of databases*. Vol. 8. Addison-Wesley Reading.
[2] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. 2017. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367* (2017).
[3] Peter Alvaro, Tom J Ameloot, Joseph M Hellerstein, William Marczak, and Jan Van den Bussche. 2011. A declarative semantics for Dedalus. *UC Berkeley EECS Technical Report* 120 (2011), 2011.
[4] 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.
[5] Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. 2010. Prime: Byzantine replication under attack. *IEEE transactions on dependable and secure computing* 8, 4 (2010), 564–577.
[6] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 355–362.
[7] Vitalik Buterin, Diego Hernandez, Thor Kamphefner, Khiem Pham, Zhi Qiao, Danny Ryan, Juhyeok Sin, Ying Wang, and Yan X Zhang. 2020. Combining GHOST and Casper. arXiv:2003.03052 [cs.CR]
[8] Miguel Castro, Barbara Liskov, et al. 1999. Practical byzantine fault tolerance. In *OsDI*, Vol. 99. 173–186.
[9] David CY Chu, Chris Liu, Natacha Crooks, Joseph M Hellerstein, and Heidi Howard. 2024. Bigger, not Badder: Safely Scaling BFT Protocols. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data*. 30–36.
[10] 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. *Proceedings of the ACM on Management of Data* 2, 1 (March 2024), 1–25. https://doi.org/10.1145/3639257
[11] 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

[12] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 277–290.

[13] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, Mirco Marchetti, et al. 2009. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. The USENIX Association.

[14] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)* 35, 2 (1988), 288–323.

[15] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. 2019. SBFT: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 568–580.

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

[17] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*. 45–58.

[18] Leslie Lamport. 2019. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*. 277–317.

[19] Leslie Lamport, Robert Shostak, and Marshall Pease. 2019. The Byzantine generals problem. In *Concurrency: the works of leslie lamport*. 203–226.

[20] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. 2016. {XFT}: Practical fault tolerance beyond crashes. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 485–500.

[21] JW Lloyd. 1994. Practical advantages of declarative programming. 3 – 17. Conference Proceedings/Title of Journal: Joint Conference on Declarative Programming.

[22] C Mohan, Bruce Lindsay, and Ron Obermarck. 1986. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)* 11, 4 (1986), 378–396.

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

[24] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. 2015. Visigoth fault tolerance. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–14.

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

[26] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 1–17.

[27] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*. 347–356.

Received 15 May 2024