Cerberus

A Parallelized BFT Consensus Protocol for Radix

Florian Cäsar
florian@radixdlt.com

Daniel P. Hughes
dan@radixdlt.com

Josh Primero
josh@radixdlt.com

Stephen J. Thornton
steve@radixdlt.com

3rd March, 2020
v1.01

Abstract

We present Cerberus, a sharded Byzantine fault-tolerant (BFT) solution to the state machine replication (SMR) problem. While classical SMR protocols ensure global ordering of commands, Cerberus introduces a partial ordering regime that enables a novel state sharding approach. With this approach, the consensus process for commands may be massively and safely parallelized across many nodes while retaining optimistic responsiveness and linear message complexity. Cerberus is designed as a suitable core for Radix, a distributed ledger technology (DLT) platform intended for a variety of deployments including global-scale decentralized public networks.

1 Introduction

Consensus protocols are necessary to conduct state machine replication (SMR) across a network of unreliable machines. While broadly applicable and with a long history, consensus protocols designed for SMR within distributed ledger technology (DLT) networks are of intense interest. These networks range from global-scale public decentralized platforms to critical private business back-ends and often have very high transactional throughput demands. Extending and adapting existing consensus protocols to meet this demand for scalability has proved challenging.

We present Cerberus as a new scalable consensus protocol of particular utility in these high-demand DLT networks – including the Radix DLT network.

Scale Through Parallelism

Most consensus protocols today come to an agreement about the ordering of all commands (e.g. transactions or events) along a *single* timeline, stored in agreed chronological order in a blockchain or similar. This is also known as "global ordering". Global ordering is generally required for any DLT offering traditional, imperative, "Turing complete" smart contract virtual machines where correct smart contract execution state may depend unpredictably on *any* other state within the system. Coordinating the consensus process for large numbers of commands generally incurs high communication overhead and consequently limits achievable throughput.

State sharding is a technique that can be used to parallelize consensus agreement to increase throughput. However, this parallelization is only possible if the method of sharding used enables separation of commands that are unrelated and thus may be safely processed in parallel. Sharding in this way is in direct opposition to the single timeline implied by a global ordering requirement.

Cerberus addresses this shortcoming by using *multiple concurrent timelines* that only enforce ordering of related events. To achieve this, Cerberus takes a combined approach to the consensus and application layers that does not require global ordering even for smart contract-like functionality. This opens up the possibility of safe parallelism of consensus through state sharding that substantially reduces throughput limitations.

Application Layer Interaction

Eliminating the need for global ordering in the application layer requires an alternative to the typical "smart contract" development method. We believe that a different approach is not only possible, but that it also has benefits (covered outside this paper) that go beyond removing the global ordering burden from consensus. This alternative approach breaks useful DLT application functionality into components that are functionally independent.

An exemplar of an application layer that works in this way is the Radix Engine, the application layer for Radix. The Radix Engine allows a developer to specify transactional functionality in an "asset-oriented" fashion. Useful elements of everyday transactions such as identities, money, products, or property are modeled not within "unbounded" smart contract code, but as functional, separate components that may be updated independently. The Radix Engine is able to translate these components into discrete finite state machines, which then easily translate to a shardable language Cerberus understands. The Radix Engine is, therefore, able to express a transaction as a command to Cerberus consisting of a collected set of state machine updates with explicit dependencies.

Cerberus is designed to take advantage of the highly granular partial ordering made possible by this type of application layer, enabling massive parallelism of consensus.

Related Work

The emergence of Bitcoin, distributed ledger technologies (DLT), and smart contract platforms have fueled a new era of lively research in consensus protocols. Recent developments in Bitcoin-style "Nakamoto" consensus [1][2], PBFT-like consensus [3][4], randomized consensus [5][6], and others represent advancements in fundamental consensus approaches that have diverse tradeoffs in the guarantees they can provide to users.

Cerberus' advancement in parallelization of consensus is based on a leader-based Byzantine fault-tolerant (BFT) consensus approach. BFT protocols for state machine replication (SMR) have the advantage of strong consistency (as opposed to availability), with a long history reaching back to the work of Lamport, Shostak, et al [7]. A definitive milestone in practical BFT protocols was reached with PBFT [8].

PBFT uses a two-phase consensus process of *prepare* and *commit*. Nodes (or participants in consensus, e.g., computers in a decentralized network) select from their ranks a "leader". A leader proposes a command by broadcasting it to the other nodes. When 2f+1 nodes (where f is the maximum tolerated number of Byzantine faulty nodes) acknowledge the command's validity by responding with their signature, the signatures are aggregated into a "quorum certificate" (QC) and broadcasted to the nodes as proof. The 2f+1 requirement ensures that even double-voting faulty nodes cannot win a majority. Once 2f+1 nodes acknowledge and sign the proof, the command is considered "committed" and final.

PBFT provided the first practical implementation of a partially synchronous BFT protocol, but it did not grapple with today's demand for scalability in open public networks or global cloud deployments. Of particular concern for these large network deployments is the high message complexity required for each "view change", or leader replacement, due to PBFT's requirement for 2f+1 QCs as a proof of safety before making progress.

More recent BFT work such as Tendermint [3] extended the state of the art in the context of large blockchain networks by reducing the messaging overhead required for view changes. Rather than requiring 2f+1 QCs, a new leader simply "extends" its highest QC (using it as the basis for a new proposal). While eliminating substantial messaging overhead, this opens the possibility that there is another node with a higher QC. Therefore a leader must wait a defined maximum network delay time before it can be sure that it has the highest QC, preventing deadlocks. This reduces responsiveness of the network by introducing a fixed latency and therefore limits throughput by the fixed time window rather than the actual speed of the network.

More recently, the HotStuff protocol [4] presented an important extension to previous BFT approaches. HotStuff maintains optimistic responsiveness while keeping low messaging complexity by adding a third phase between prepare and commit: *pre-commit*. The pre-commit phase allows leaders to "'change their mind' after voting "in the phase", meaning that consensus can safely proceed on new proposals and the network no longer has to wait a defined network delay time. A commit occurs as soon as QCs for pre-commits have been collected successfully. Combined with threshold signatures, this provides “optimistic responsiveness” and linear messaging complexity even for view changes. We leverage many of HotStuff’s extensions, and corresponding advantages, as we construct the sharded Cerberus BFT protocol.

Scope of this Paper

We begin by describing the Cerberus model as a 3-phase BFT consensus protocol with a sharding technique for partially-ordered commands. We then briefly analyze the model to show that it achieves linear messaging complexity while balancing the requirements for node capabilities, parallelization and security. A detailed specification of the Cerberus protocol, as well as formal proofs of safety and liveness, will follow in a later revision of this paper.

We follow the Cerberus model with topics relating to the implementation of Cerberus in practical DLT networks, focusing on the Radix public network. We describe methods to provide protocol functions outside of Cerberus consensus that are required in order to create and deploy a useful Cerberus-based network.

This paper is a working document for the Radix team and still under active research; as a result some areas will require further explanation or revision. As always, we welcome comments, collaboration and feedback.

2 Model

Overview

Cerberus solves the state machine replication (SMR) problem [15]: the fault-tolerant distribution of state across many nodes. Cerberus builds on top of BFT-style consensus, retaining two main properties of BFTs:

1. Safety under asynchrony, liveness under synchrony
2. Favoring consistency over availability (i.e. safety over liveness in periods of asynchrony)

We also make the following common BFT assumptions in our basic model. While we make these assumptions for the purposes of describing a simple form of the Cerberus model, some of these restrictions may be lifted in practical implementation (notably a static node set). We discuss some of these in the *Implementation* section of this paper.

- A static, globally-known set of nodes
- Partial synchrony (i.e. after some global stabilization time there is an unknown bound within which all messages arrive)
- *2f+1* nodes follow protocol, where *f* is the number of Byzantine nodes (per shard in the case of Cerberus)
- Adversaries that are computationally incapable of breaking standard cryptographic primitives
- The availability of a practical threshold signature method to aggregate votes from nodes
- Communication between correct nodes is peer-to-peer, authenticated, and reliable

In classical BFT, a log of client commands is globally ordered so that correct nodes can agree on the final state. Performance improvements have traditionally relied on “vertical” scaling or “pushing more commands down a single pipe”. In practice, applications generally do not require global ordering of all commands; they only require ordering of commands that are interdependent.

Cerberus requires an application layer that specifies dependencies so that it may implement *partial ordering*, only ordering commands which are related. Cerberus’s “shards” represent these dependency relationships between commands. Commands that do not share a partial ordering relationship also do not share shards and can be safely executed in parallel.

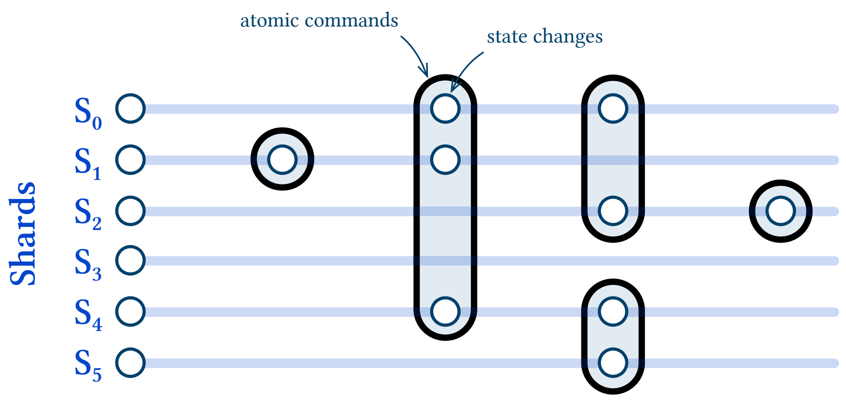
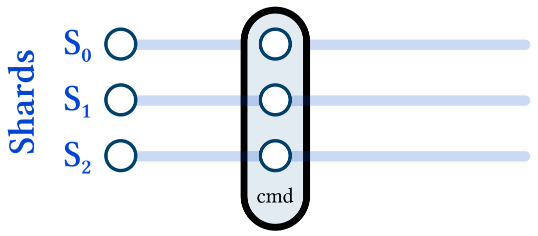
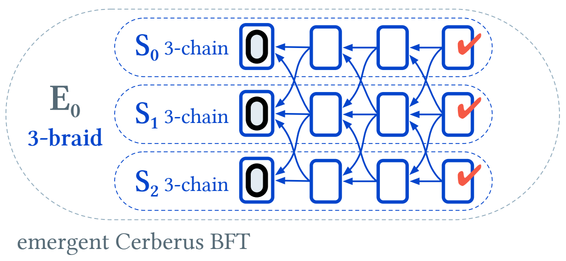


Figure 1: *Partial ordering created by commands (thick edged pill shapes) grouping related shard state changes (thin edged circles) across shards s_0-s_5*

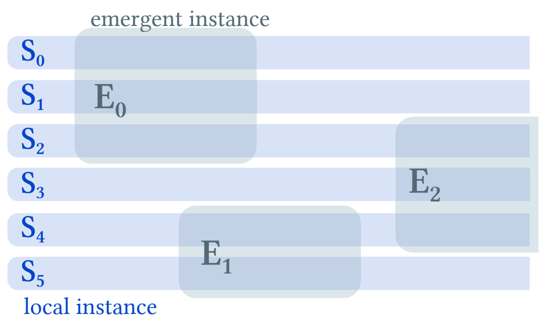
With commands sharded in this way, Cerberus conducts consensus by treating each shard as a separate "local Cerberus" BFT instance, with atomic commits of commands across arbitrary sets of shards provided by an "emergent Cerberus" BFT. We construct our description of the Cerberus protocol in three major steps:



A) Partition the application layer by formalising the partial ordering requirements of each command in the application layer.



B) Combine multiple "local Cerberus" BFT instances to form an "emergent Cerberus" instance across shards with analogous properties to a simple traditional BFT.



C) Parallelize Emergent Cerberus instances across shards, enabling massive and safe parallelization of the consensus process across the network.

A) Partition the Application Layer

Cerberus requires that its application layer specify the partial ordering requirements of its commands so that consensus may be effectively parallelized. Specifically, the implementer must define the following *pure* functions for the application layer:

1. Specify the initial state of each shard

```
INIT() -> map of shard -> initialShardState
```

2. Partition the command into its set of shards

```
PARTITION(command) -> shards
```

3. Map a command to its per-shard intermediate results

```
MAP(command, shard, shardState) -> localResult
```

4. Reduce the local intermediate per-shard results across all shards

```
REDUCE(command, shard, shardState, localResults) -> nextShardState
```

`INIT` defines the initial state of the application. `PARTITION` yields the shards a command must be synchronized with to maintain correct partial ordering. When a command is to be executed, it is first mapped to per-shard intermediate results via `MAP`. The intermediate results are then reduced to a global result and applied across all shards of the command via `REDUCE`. This approach is similar to that of contemporary large-scale distributed computing models [9]

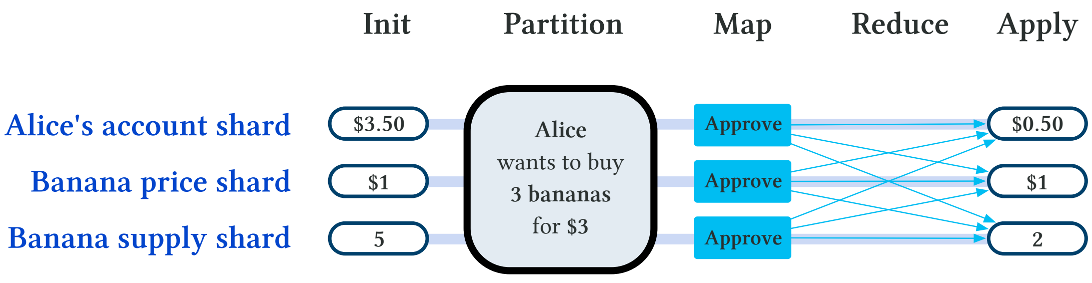


Figure 2: *Example of using the application layer to partition consensus*

For example, consider a simple application layer consuming a command saying "*Alice wants to buy 3 bananas for $3*". This command could map to shards representing the available banana supply, current banana price as well as Alice's current account balance[1]. Upon receiving the command, the shards will map the command of it to get their individual intermediate result. Cerberus then distributes these local results to all shards and applies them in the reduction step to yield the next shard states.

B) Local Cerberus Instances into Emergent Cerberus

We now describe how Cerberus can execute a command atomically across all shards it is partitioned to, starting with the operation of consensus within a single shard and moving on to operation of consensus across multiple shards.

Local Cerberus

We start with a single BFT instance patterned after the BFT design of HotStuff [4]. We call this a local Cerberus instance on a single shard that, operated in isolation, provides the same optimistic responsiveness and linear messaging complexity of HotStuff (as applied to a single blockchain). These advantages will serve as a useful foundation as we construct Cerberus as a multi-shard consensus protocol.

[1] Note that Radix's mapping and usage of shards are different than the example shown in figure 2. See "Radix Engine" in the Implementation section of this paper for more information.

Specifically, we start with a "3-chain" BFT composed of prepare, pre-commit, and commit phases. A quorum certificate (QC) – an aggregation of signatures from validating nodes – is required for each phase.

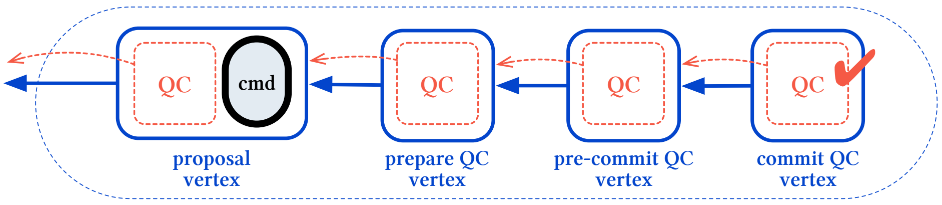


Figure 3: *A local Cerberus 3-chain*

Progression through each phase of consensus is marked by the creation of a "vertex" (shown as the enclosing rounded squares in figure 3) composed of that phase's QC and a reference to the Vertex's parent, similar to blocks in a blockchain. In a local Cerberus instance, the vertex representing a new proposal (the beginning of a new 3-chain) includes the command.

Multiple Local Cerberus Instances

We now introduce multiple instances to the model with *n* independent local Cerberus instances operating in parallel. For simplicity, we assume that each of these instances are served by a distinct set of nodes (i.e. each node is assigned to exactly one instance). This bi-map between nodes and instances are assumed to be static and well known to every node[2].

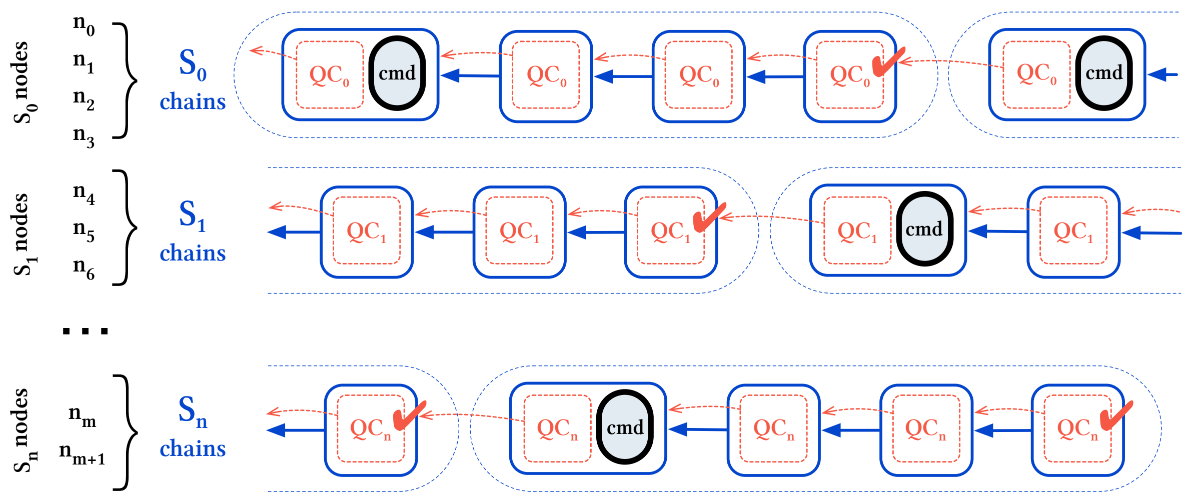


Figure 4: *Nodes distributed over n parallel instances of local Cerberus (denoted S)*

These parallel instances are not very useful yet however as there is no way to synchronize them.

Emergent Cerberus Instance

We now synchronize multiple local Cerberus instances. To do this, we add synchronization primitives between multiple local Cerberus instances and model the synchronized combination as a single emergent Cerberus instance which behaves like a single cross-shard BFT instance.

[2] We discuss practical mapping of nodes to shards and management of dynamic node sets in the Implementation section of this paper.

We implement this emergent level by mapping analogous concepts from Traditional BFT (i.e. local Cerberus instances) to multi-shard concepts in emergent Cerberus:

Traditional BFT (Local Cerberus)	**Emergent Cerberus**
Leader	Leader Set
Proposal	Merged Proposal
QC	QC Set
3-chain	3-braid

Emergent Leader: Leader Set

A local Cerberus instance designates a leader to drive progress in each new view via *new-view* messages from that shard's nodes. At the emergent instance level, the set of leaders elected by a set of shards is considered, in aggregate, to be the emergent leader called a "leader set".

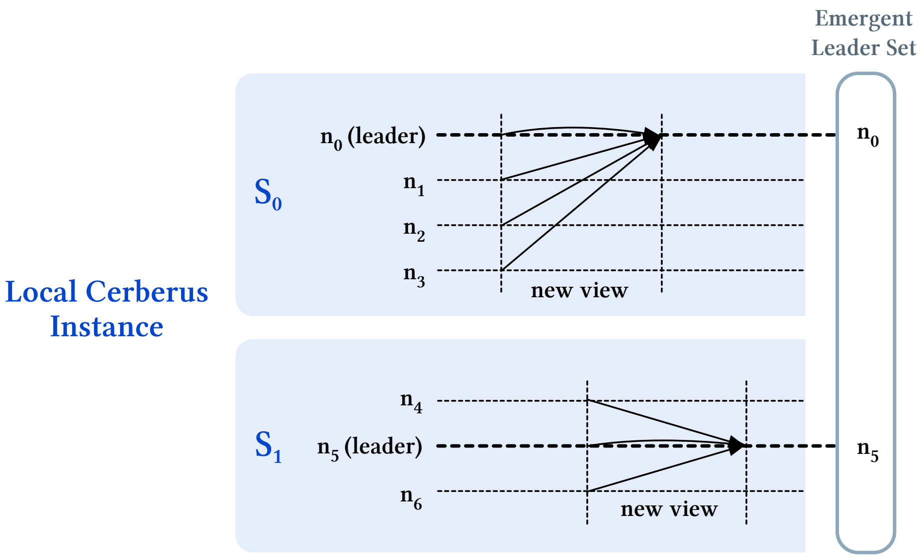


Figure 5: *Leader set implicitly formed as a set of local leaders*

In the example shown in Figure 5, two Cerberus shards independently elect local leaders (n0 and n5, respectively) via new-view messages. The set {n0, n5} is then considered the leader set across the two instances. Note that at this stage no node is aware of the leader set, it is only a theoretical construct for the purposes of describing and analysing the protocol.

Emergent Proposal: Merged Proposal

We now add the mechanism by which a leader set can propose a new command, or a "merged proposal".

To explain by example, each local leader in a leader set, {n0, n5} in our example, broadcasts a local proposal to all nodes in the emergent Cerberus instance (the shards of which are specified by the application layer's `PARTITION` function). Each node then takes each proposal from each local leader in the leader set and merges them to form a merged proposal.

As commands are sharded, each local proposal also includes the local intermediate result provided by the application layer's `MAP` function.

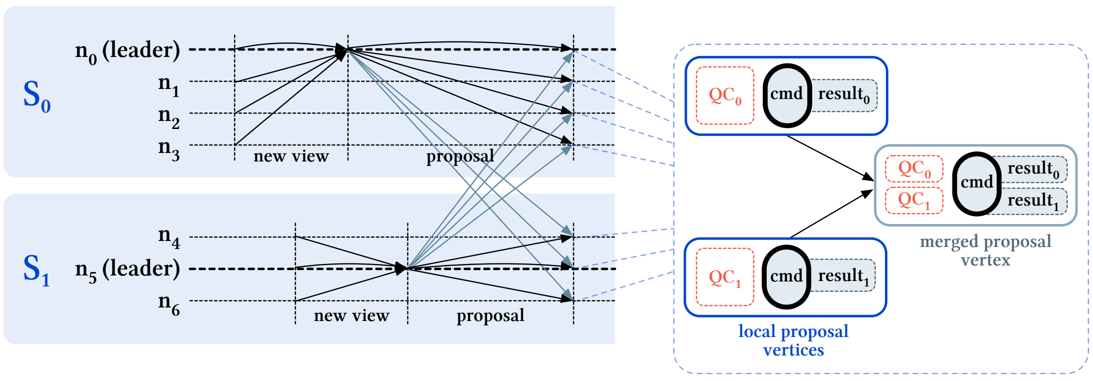


Figure 6: *An emergent merged proposal is formed from a set of local proposals with equivalent commands*

This merged proposal then acts as a vertex at the emergent level which begins the 3-braid between shards S_0 and S_1.

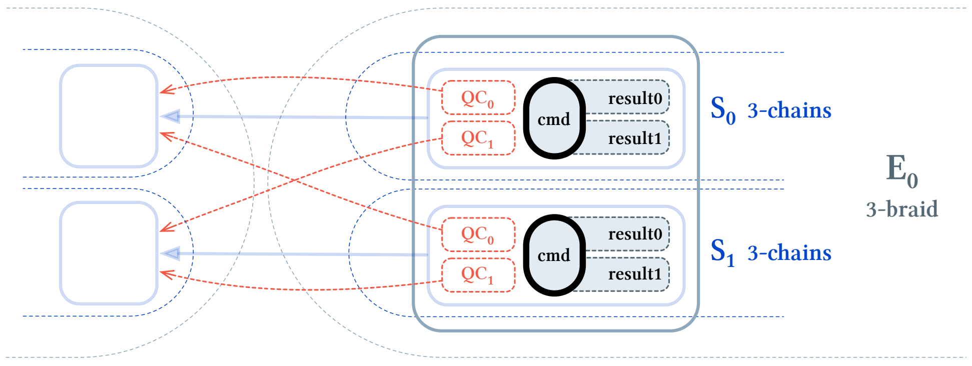


Figure 7: *The merged proposal is added to the end of both chains*

Emergent QC: QC Set

We now add a mechanism for the leader set to collect votes for the vertex. When a node receives an merged proposal, it sends its vote to its local leader. When the local leader has *2f+1* votes, it creates a QC. From the emergent view, the leader set creates an emergent QC, or "QC set", made of one QC from each shard, when each of its local leaders creates its local QC.

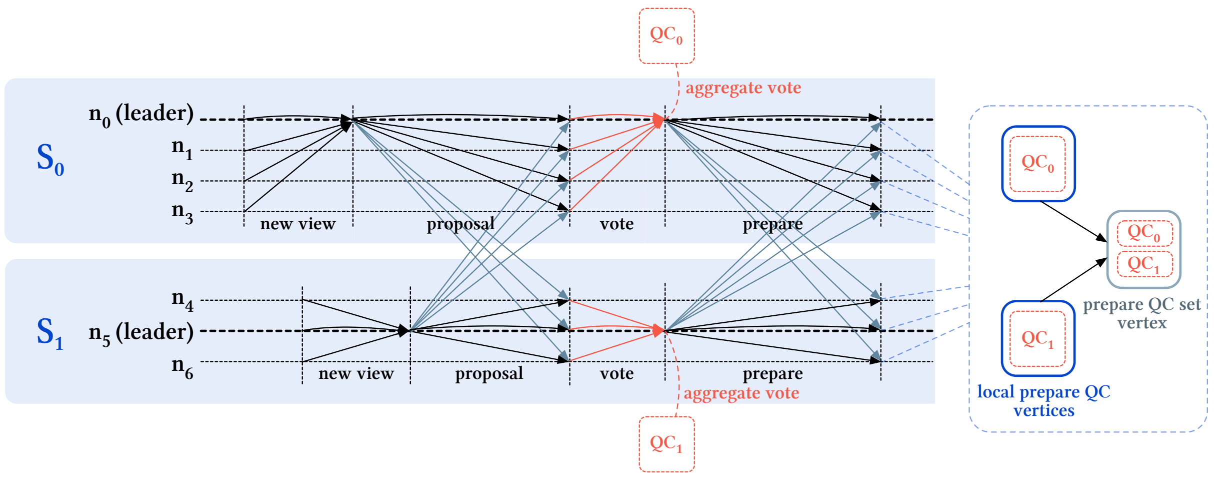


Figure 8: *Emergent QC set for cmdA collected from QCs from each shard, for a single phase*

Emergent 3-chain: 3-braid

The QC set is then passed on to the leader set in the next view where the process starts again repeating the process until a 3-braid is formed for cmdA.

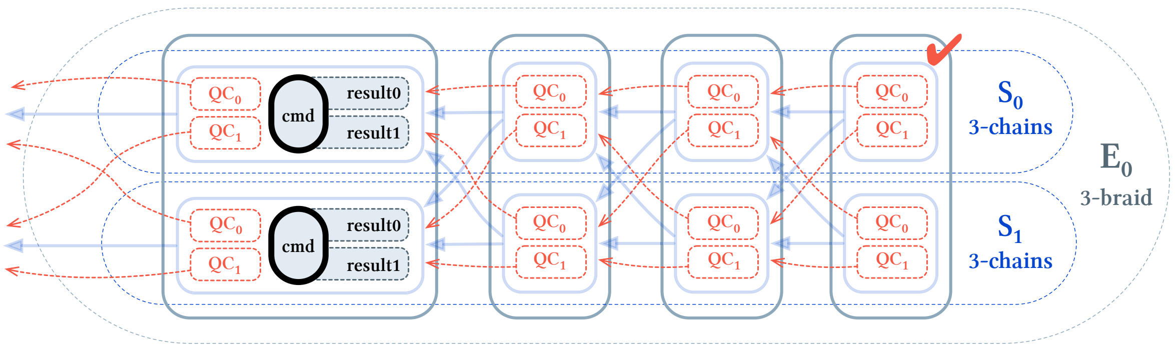


Figure 9: *3-braid in an emergent instance across two local instances*

Once a 3-braid is created for a command, the command is considered "committed".

Emergent Failure Mitigation

We now consider how the emergent Cerberus instance handles leader failures, forks, and merges.

Emergent Leader Failures

In a local Cerberus instance if a vertex fails to get created at some view, a blank vertex is added in its place so that the parent/justification equivalence check is false. In an emergent Cerberus instance, every parent of each shard must be checked.

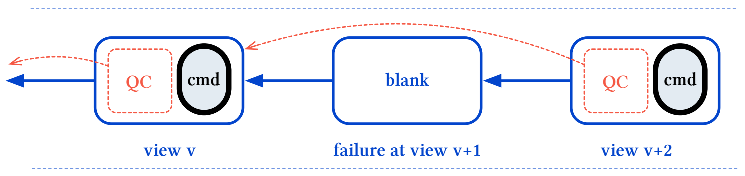


Figure 10: *Local Cerberus instance failure, the parent of v+2 vertex (vertex at v+1) is not equivalent to the justification of v+2 (vertex at v).*

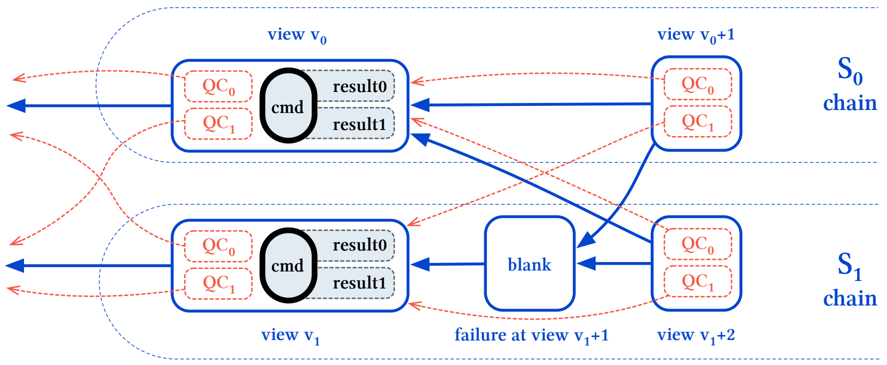


Figure 11*: Cerberus failure, parent0 is not equivalent to the rest of the parents or justifications.*

Emergent Forks

In a single local Cerberus instance, safety and liveness are preserved in the face of forks with checks on locked vertices and/or QC view numbers. With an emergent Cerberus instance, there is additional complication since a fork may occur outside of a local instance's view.

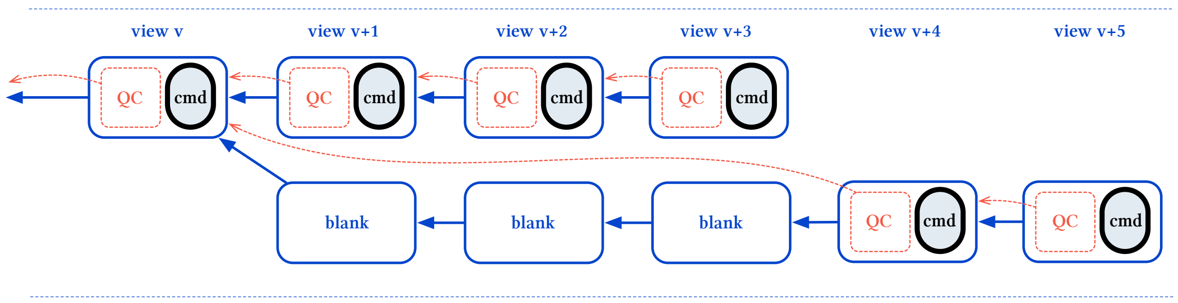


Figure 12: *Local Cerberus instance fork which requires an "unlock" of the vertex with cmd at view v+1*

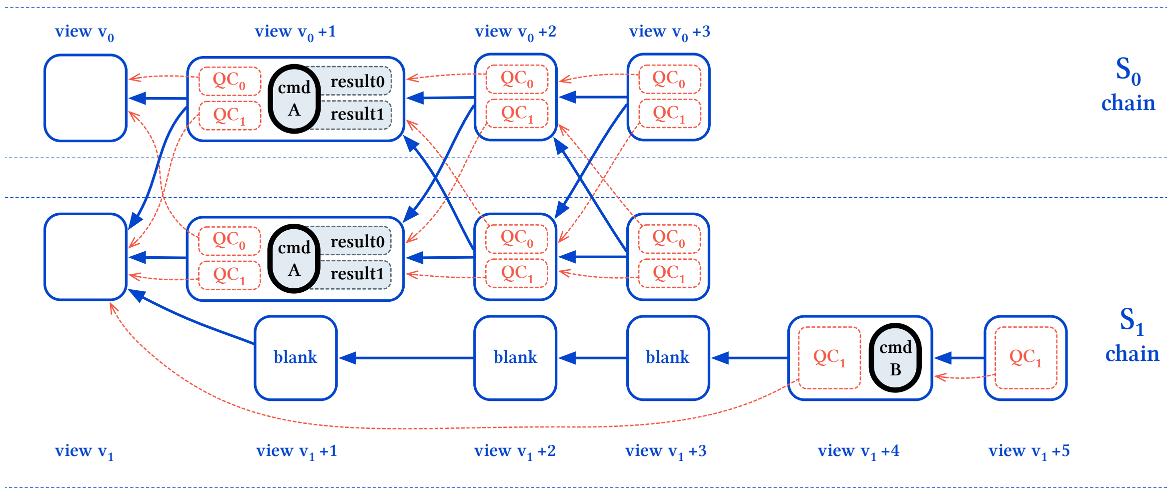


Figure 13: *Emergent Cerberus fork which requires shard S_1 to "unlock" shard S_0 from the vertex with* `cmdA` at view v_0+1

We extend HotStuff's check in the emergent instance by including the forked instance's higher QC in the emergent QC. This is detected when a forked instance receives a proposal that has an older, conflicting QC. The forked instance then sends the forked QC as a message to the other shards.

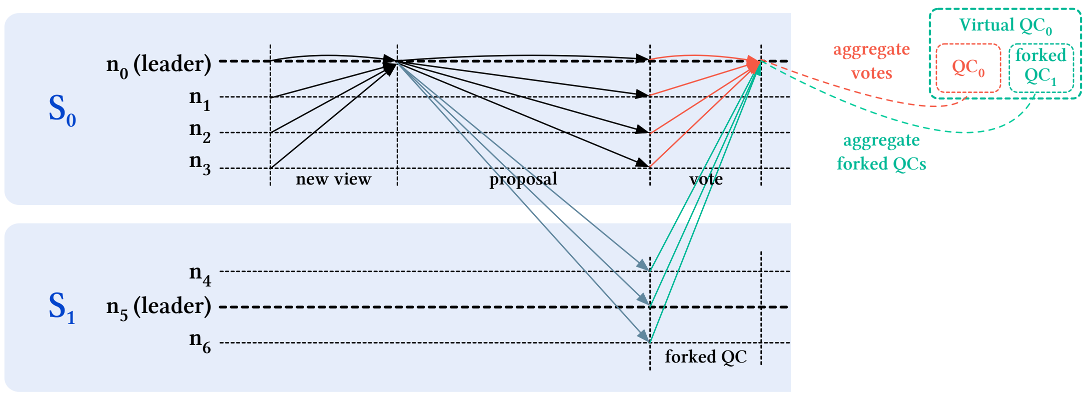


Figure 14: *Recovery of a cross-instance fork in an emergent instance through forked QC*

This forked local QC is then included in the emergent QC and is proof of a fork. A "virtual QC" is then created from this proof which "virtually" has a higher `viewNumber` than the current QC and extends back to the point where the forked QC is no longer forked.

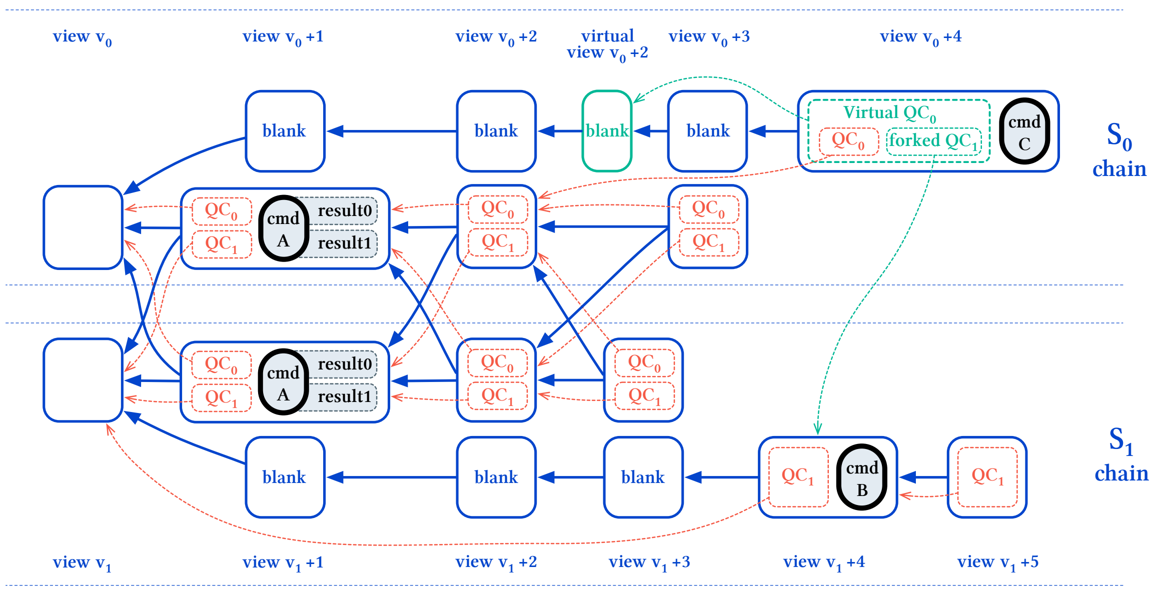


Figure 15: *A Virtual QC with a virtual higher* `viewNumber` *is used to "unlock" a forked Emergent Cerberus instance*

Virtual QCs thus act as a level of indirection. We now refer to QCs which actually contain signatures as *physical QCs*. A *virtual QC* is composed of a primary *physical QC* and an optional *forked physical QC*. A *virtual QC* composed of just the primary QC without the forked QC acts like the primary. For the purposes of simplicity, for virtual QCs with no fork, we will represent this as a single blue box. And for virtual QCs with a fork, we will represent this as a blue box with a justify arrow pointing to a virtual node. For example, Figure 15 can be reduced to the following diagram:

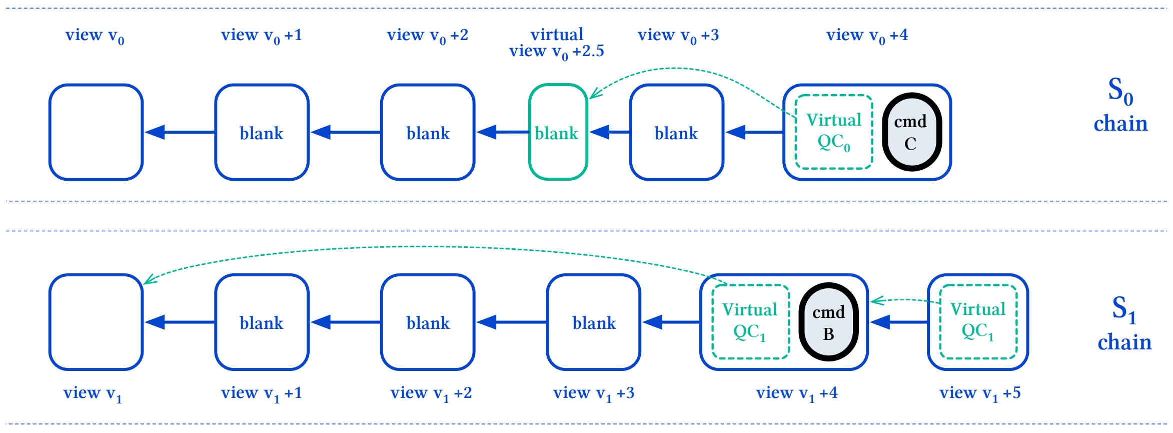


Figure 16: *Virtual QCs operate as a layer on top of physical QCs and substantially reduce the complexity of fork management*

Emergent Merges

As long as the leader set is correct and proposes a command, progress should always be made in order to ensure liveness. A complication arises in the case of forked local proposals, which are possible when dealing with separate independent Cerberus local instances.

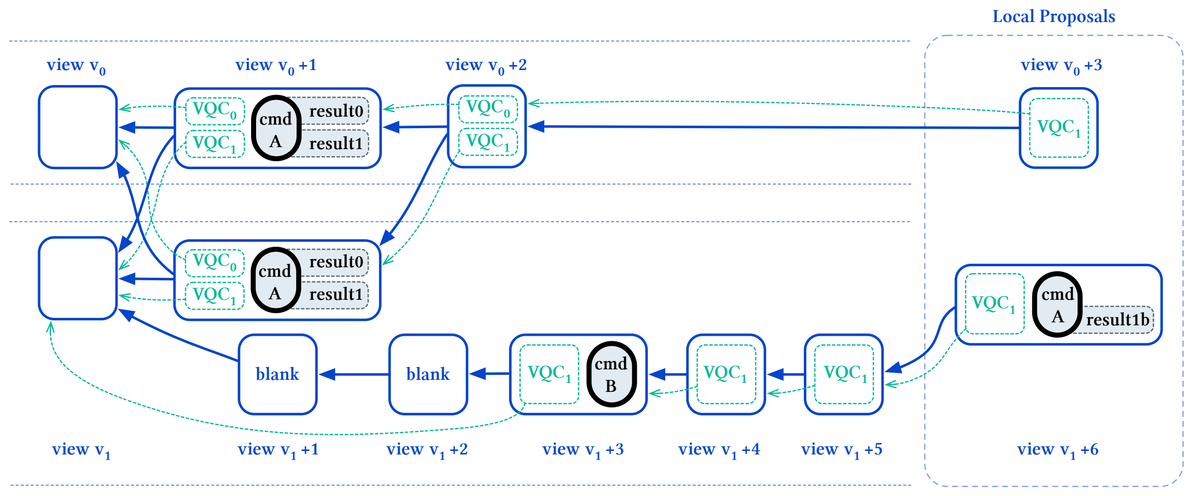


Figure 17: *Local Proposals may be forked - in this example there are two vertices connected at view v_1+1 on S_1 and thus is forked*

To resolve forked proposals, we modify the merging of local proposals so that if there exists a fork amongst the local proposals, we reuse the Virtual QC method to effectively unlock the locked shard.

As an example, in Figure 17 VQC0 and VQC1 in views v_0+3 and v_1+6 respectively fork. To resolve this, the physicalQC of VQC1 is added as a forkedQC to VQC0. This effectively removes the fork.

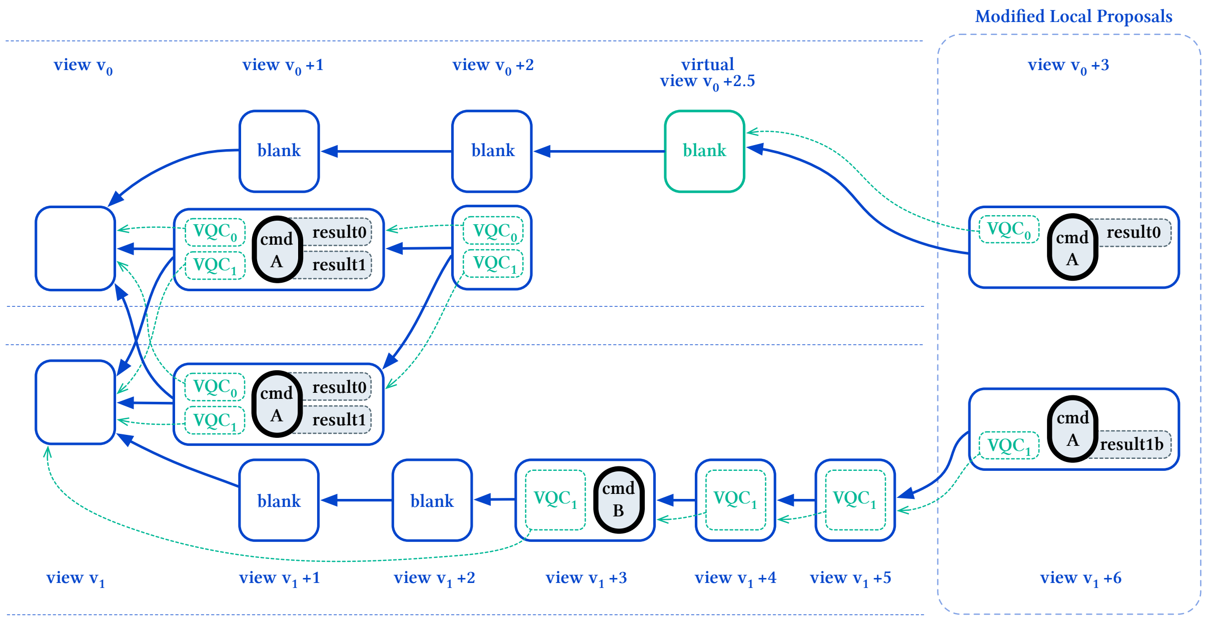


Figure 18: *As the original local proposed virtual QCs forked, we modify the proposals by including forked QCs until the fork disappears. Normal merging of proposals can now continue.*

The modified local proposals are then able to merge as normal without forks. The result can be seen in Figure 19.

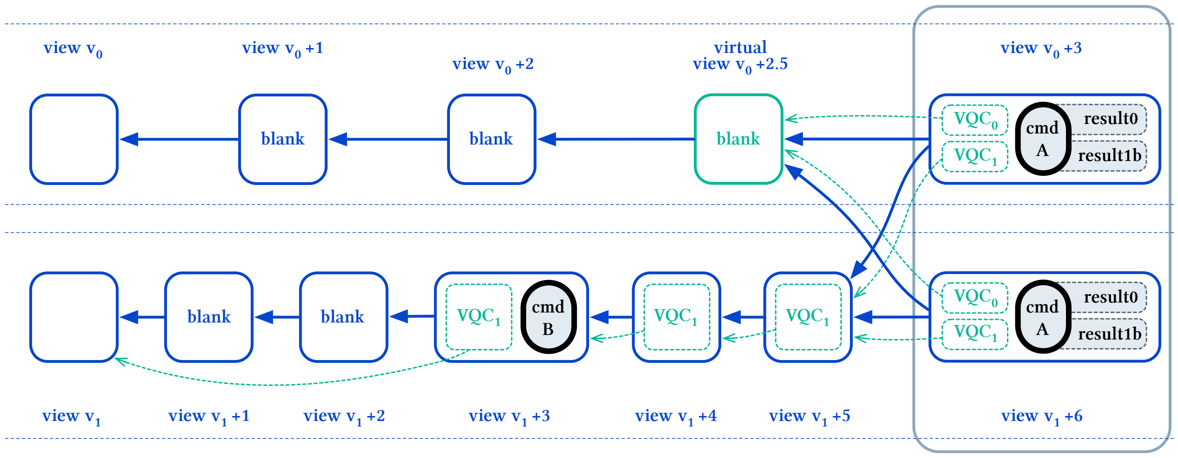


Figure 19: *The merged proposal post-modification cleanly attaches to both chains*

C) Parallelize Emergent Instances

Now that we have described a mechanism that combines multiple local Cerberus instances into an emergent Cerberus instance, we can now parallelize emergent instances across a fixed set of shards.

Given a set of shards *S={S_0, S_1, …,S_n}* we can create an emergent instance across any subset from *S*. Furthermore, we can create a set of emergent instances in parallel (a set of subsets of shards from S) as long as each instance is disjoint (no element in common) from others. For example, *{{S_0, S_1}, {S_2}, {S_3, S_4, S_5}}*.

The creation of emergent instances is dictated by the set of commands to be processed and the result given by the application layer's `PARTITION` function across these commands.

In the example below, `cmd0-4` defines a set of emergent instances. The partitions of `cmd0` and `cmd1` are disjoint shard sets thus they are able to create two separate emergent instances in parallel. On the other hand, `cmd1` and `cmd2` intersect at shard S_3 thus one command must be executed before the other (`cmd1` in this case).

```
PARTITION(cmd0) -> {S₀, S₁, S₂}
PARTITION(cmd1) -> {S₃, S₆}
PARTITION(cmd2) -> {S₂, S₃, S₅}
PARTITION(cmd3) -> {S₄, S₅}
PARTITION(cmd4) -> {S₆, S₇}
```

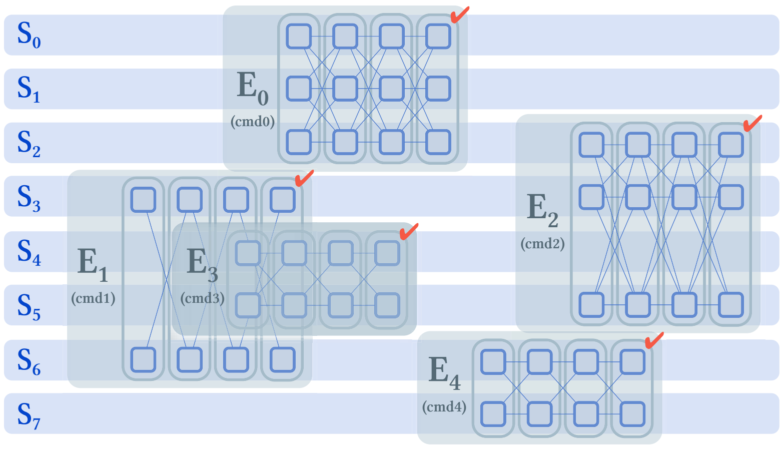


Figure 20: *Example parallelization of Cerberus for commands cmd0-4 across shards S_0-S_7*

The set of shards can be arbitrarily sized, enabling massive parallelization of the consensus process. Notably, this form of parallelization introduces a tunable tradeoff between security and scalability: reducing the number of shards each node covers increases parallelism and scale, but decreases the validator set size of a given shard. This capability allows the system designer to select a minimum threshold of per-shard node coverage that achieves high practical security, while maximizing parallelism and scale by allowing a large number of independent nodes. This is unique to Cerberus and discussed further in the analysis section.

Analysis

As with HotStuff, the messaging complexity of Cerberus is measured in authenticator complexity, the number of authenticators received by any given node in order to reach a consensus decision. In the nominal case using threshold signatures, HotStuff achieves authenticator complexity of $O(n)$ – linear complexity with the number of nodes n[3].

In Cerberus, we must consider sharding. Given an evenly distributed node-to-shard mapping and single-shard commands, Cerberus behaves like HotStuff and the authenticator complexity given a correct leader is $O(n / T)$ where T is the number of shards. As T approaches 1, or a single shard consensus instance, authenticator complexity approaches O(n), the same as HotStuff.

With emergent Cerberus instances, cross-shard consensus requires a separate QC per shard. Thus the authenticator complexity with correct leaders is $O(s * n / T)$ where s is the number of shards in a given command (and s must be $1 <= s <= T$).

This definition lets us consider the tradeoffs of deploying and using a Cerberus-based network. In effect, s / T is the burden placed on the network by the application layer, while n / T defines a tradeoff between security (more nodes per shard) and parallelism (fewer nodes per shard). For example:

- In a practical HotStuff network, a certain set number of nodes might be selected to provide the amount of per-shard node overlap needed to provide high practical security, without creating too much messaging overhead. If we apply that same set number to the number of nodes per shard in a Cerberus network, the throughput of the Cerberus network can be scaled linearly with the number of nodes, without compromising per-shard security.
- If we assume an application layer creating commands with an average low number of dependencies (relatively simple transactions), s burdens messaging very little. One can imagine implementing a public network in which greater fees are paid only for unusually complex transactions.

3 Radix DLT Network Implementation

While Cerberus is a consensus protocol that may be applied to a variety of applications, an application area of particular interest is that of scalable DLT networks such as Radix. This section describes a set of conservative and reasonably extensible potential solutions to implementation issues in deploying Cerberus in Radix, and potentially other DLT networks. While we attempt to cover some of the most important aspects of implementation, the solutions provided are not intended to be fully formed, nor is the listing of issues exhaustive.

Radix is intended for a range of network deployment types, the most challenging of which is a public permissionless network. The solutions proposed in this section are generally intended to be applicable across Radix network deployment types, but with a particular focus on a very large network with open participation.

Radix Engine Application Layer

To provide its parallelization benefits, Cerberus requires an application layer that specifies the dependencies between commands. Here we present the key interactions between one such application layer, the Radix Engine, with Cerberus consensus.

A Radix node combines a Cerberus-based Radix Ledger with an application layer called the Radix Engine. A collection of Radix nodes make up a Radix network called a "universe", and transactions created by the Radix Engine are called "atoms". Atoms contain a set of discrete updates to finite input/output state machines called "particles". This is similar to the common blockchain UTXO model but extended for greater flexibility. As might be expected, atoms must be committed or rejected by Cerberus consensus atomically; either all particles within the atom are valid and the atom is committed, or one or more particles are invalid and the atom is rejected.

Radix Engine atoms and their particles map neatly to Cerberus commands that specify related state changes. The particles themselves define the mapping of the atom to shards, with each particle mapping directly to a shard (the particle/shard addressing scheme is described below in Particle Identifiers).

[3] Similar to other leader-based BFT protocols, the linear message complexity of Cerberus relies on the use of a signature aggregation scheme (e.g., using BLS signatures).

Each particle/shard is, therefore, a simple state machine that looks like figure 21.

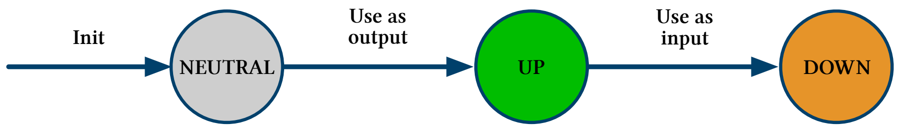


Figure 21: *The defined sequential states of "particles": neutral (default), up (active), and down (consumed)*

Given the UTXO-based logic provided by the Radix engine, which dictates which input/output pairs are valid, we use Cerberus to enforce the atomic commit of updating the state of multiple particles (either up or down). An example ledger is shown in figure 22.

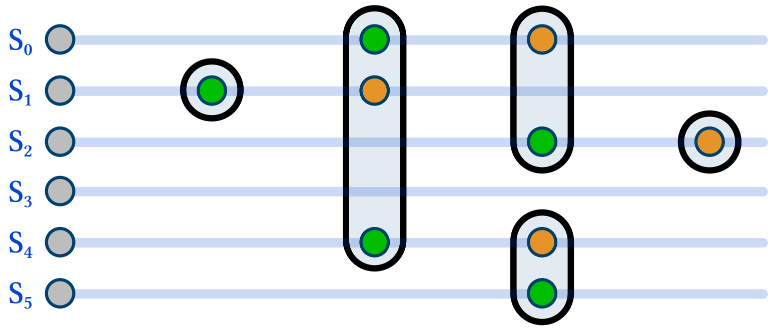


Figure 22: *Thick edged pills represent commands that group related particle state changes (thin edged circles) in their respective shards.*

Particle Identifiers

Particles, being the fundamental unit of the Radix Engine, must be uniquely addressable by an identifier. Inspired by Chainspace's "Hash-DAG structure" [11], we identify a particle by the atom in which it was created to create a high-integrity data structure of transactions in which it is both hard to forge a transactions history but easy to verify it.

A particle's identifier uses a combined hash of a containing atom's identifier and the particle's (deterministic) index within that atom to identify particles. A particle's identifier is "created" when it first appears in an atom, and any subsequent changes to that particle use this same identifier.

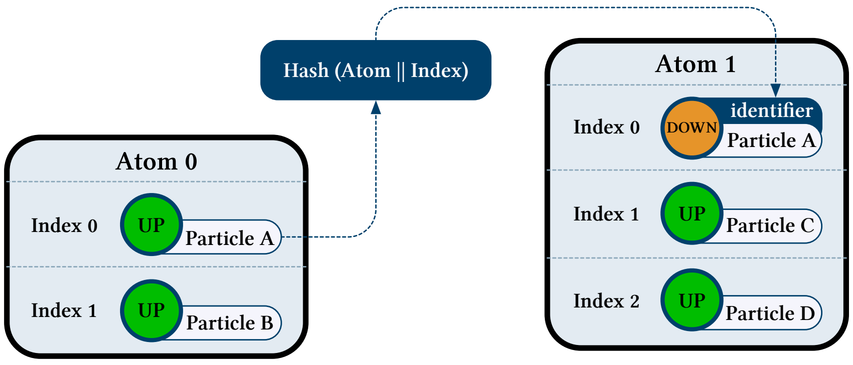


Figure 23: *Using a particle's identifier (right) as defined at the particle's creation (left)*

This particle identification scheme produces immutable, auditable chains of hashes that may be traced back to the creation of each particle. Further, since the particle identifier is based on the atom it was created in, rather than its content, there is no need for nonces to ensure uniqueness amongst particles in different atoms with identical content.

Shard Mapping

We use a one-to-one mapping of particles to shards based on particle identifiers. Distributing particles across the shard space in this manner has advantageous properties:

1. The shard space can be fixed-width, static and well-known ahead of time, removing the need for complex dynamic sharding procedures.
2. The consensus process is naturally "load-balanced" across all shards, as particle identifiers are hashes that can be expected to be nearly uniformly distributed.
3. The potential for attack or disruption for an adversary of taking over any single shard are low, as a single shard contains the state of only a single particle.

This mapping provides good consensus performance and security on transaction writes. *Transaction read* queries, however, such as requesting a complete transaction history for a given account, are more difficult since every transaction is addressed by content rather than by a single identifier.

A comprehensive solution to the contradictory requirements of consensus (e.g., shard safety, load balancing) and applications (e.g., easy account querying) requires two separate solutions, each tuned to the needs of its domain and freed from the constraints of the other. We decouple the shard mapping used by consensus from that used by applications with a proposed *query layer* which operates independently on top of Cerberus. The architecture is similar to other distributed, scalable systems such as search engines on top of the web, domains on top of the internet, or queries on top of IPFS [12] or BitTorrent [13].

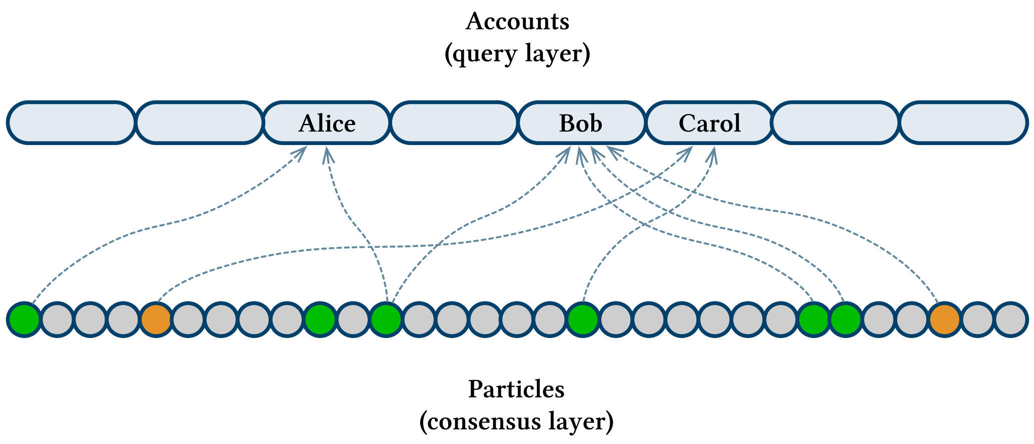


Figure 24: *A query layer utilizing a secondary index on accounts*

A simple query layer could be one based on a secondary index on accounts as shown in figure 24. This can be supported as long as the query layer can retrieve up-to-date particles.

Some practical solutions to retrieving the latest state are:

1. Let the query layer eavesdrop on every message in the consensus layer. The query layer can then tell which transactions have been committed to.
2. Have the query layer "crawl" through transactions. Since practically every transaction is linked to another, a crawler can easily walk the chain of transactions.

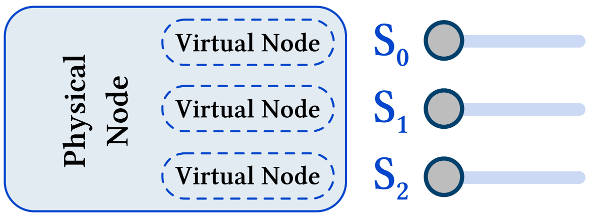
Once the latest state has been received it can be submitted to any of the many available distributed key/value databases or a decentralized DHT such as Kademlia [14].

A more specific description of the precise query layer mechanism to be used in Radix is left for future work.

Virtual Single-shard Nodes

As a theoretical model, Cerberus assigns one shard to every node. In practice, mapping nodes to *machines*, with only one shard per machine, would be an inefficient use of resources since shards represent particles that are both ephemeral and typically inactive. Most machines will have hardware capable of serving a very large number of shards simultaneously.

To retain the simplicity and correctness of Cerberus, we *virtualize* an arbitrary set of single-shard nodes in every physical node (i.e. machine). The virtual nodes behave exactly as their physical counterparts would, with the only difference being that multiple virtual nodes map to the same physical machine (with the same address and public key).



Global Shard

In Cerberus, shards allow different nodes to participate in only a subset of the network's total consensus activity in processing general client request commands (i.e. processing transactions). However in a practical DLT network, certain functions outside of general client requests require consensus and visibility of all nodes (e.g. churn management). We therefore introduce the concept of a "global shard" that can support these functions. As opposed to regular shards that are mapped to subsets of nodes (and machines) as described above, the global shard is mapped to *all* nodes. All nodes participate in consensus on special-purpose commands there, and store the global shard's state.

Consensus in the global shard operates identically to other shards, using the same decentralized consensus process in the form of a single local Cerberus instance. Rather than conducting consensus on general client commands, the global shard conducts consensus on the state of a set of global parameters that are required for network operation, such as the current node set and state of staking as described below. It may also be used for useful cross-network functions such as network time.

While regular shards in Radix may be considered ephemeral (i.e. related only to short sequence of up/down particle updates), the global shard can be thought of as a sort of narrow-purpose blockchain that is operated continuously. Compared to command throughput of the general network, the required rate of consensus state changes in the global shard is low and is therefore safe to be conducted globally without significant impact to performance.

Node Sets and Churn

Cerberus requires that all nodes must know the set of all nodes. However a practical DLT network requires tolerance of churn as nodes choose to enter and leave the network. To maintain this dynamic node set, we use an identity register that is updated in the global shard.

The identity register uses the notion of an *epoch* that captures the node set and shard mapping at a certain point in time. Epochs are globally known time periods (e.g., 1 hour or 1 day) during which the node set is fixed. Each node keeps track of the current highest epoch and includes it in all messages and client commands. At the end of an epoch, all nodes must reach consensus on the next node set by aggregating *leave* and *join* requests into a special command on the global shard.

Sybil Resistance

As a public network, we must protect against the possibility of an attacker gaining undue control of the node set and gaining the ability to break safety/liveness at will (the equivalent to the 51% bitcoin attack). We plan on employing Proof of Stake (PoS) in the BFT setting similar to Casper, Tendermint, and others to protect against Sybil attacks.

Rather than having equal voting power across all nodes, each node will have voting power relative to the amount of stake they have registered in order to compete for selection as a validator. This state will be maintained by the global shard using special *stake* and *unstake* commands. As all nodes must be a part of the global shard, each node will have complete information on the mapping between nodes, their shards, and their stakes. This allows every node to correctly execute consensus in Cerberus using validator sets selected using stake.

An in-depth description of the application of PoS to Cerberus within the Radix public network is left for future work.

Leader DOS Resistance

With our proposed leader election mechanism, future leaders are known well in advance and thus particularly vulnerable to denial-of-service (DOS) attacks. There are several possible mitigation strategies:

- Use sentry nodes as full proxies between validators (as in Tendermint [10])
- Use a leaderless model at the cost of quadratic rather than linear message complexity
- Use randomness to make the next leader less predictable (and thus give adversaries less time). This strategy would require the use of a verifiable random function, an area of ongoing research.

A deeper analysis of possible applications of these strategies within Cerberus for Radix public or private network deployments is left for future work.

Client Command Requests

Cerberus assumes that client command requests are known by leaders for them to select for proposal. A mechanism is required to reliably convey requests to leaders. A few possibilities exist, depending on the specifics of anticipated network usage.

The most common choice is an approach wherein client requests are broadly shared and a pool of pending requests is loosely synchronized (commonly referred to as a “mempool”).

Command Proposal Selection

Cerberus, being a leader-based consensus, requires a mechanism for reliable selection of client requests for proposal. We propose a random selection biased towards the oldest unapproved client requests that will eventually select all client commands. For liveness, we rely on the majority of nodes following this protocol to prevent attackers from taking advantage of predictable selection. However, it should be noted that the choice of proposal selection mechanism has no impact on safety, only liveness.

4 Acknowledgements

We would like to thank Zalan Blenessey, Alexander Cyon, Matthew Hine, Edgars Nemše, and Martin Sandiford for helping create this paper.

5 References

[1] Bitcoin: A Peer-to-Peer Electronic Cash System, 2008: https://bitcoin.org/bitcoin.pdf

[2] Casper the Friendly Finality Gadget, 2017: https://github.com/ethereum/research/blob/master/papers/casper-basics/casper_basics.pdf

[3] Tendermint: Byzantine Fault Tolerance in the Age of Blockchains, 2016: https://allquantor.at/blockchainbib/pdf/buchman2016tendermint.pdf

[4] HotStuff: BFT Consensus in the Lens of Blockchain, 2019: https://arxiv.org/pdf/1803.05069.pdf

[5] Randomized Protocols for Asynchronous Consensus, 2002: http://disi.unitn.it/~montreso/ds/syllabus/papers/randomized-consensus-survey.pdf

[6] Scalable and Probabilistic Leaderless BFT Consensus through Metastability, 2019: https://avalanchelabs.org/QmT1ry38PAmnhparPUmsUNHDEGHQusBLD6T5XJh4mUUn3v.pdf

[7] The Byzantine Generals Problem, 1982: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/The-Byzantine-Generals-Problem.pdf

[8] Practical Byzantine Fault Tolerance, 1999: http://pmg.csail.mit.edu/papers/osdi99.pdf

[9] MapReduce: Simplified Data Processing on Large Clusters, 2004: https://static.googleusercontent.com/media/research.google.com/en//archive/mapreduce-osdi04.pdf

[10] Tendermint Explained – Bringing BFT-based PoS to the Public Blockchain Domain, 2018: https://blog.cosmos.network/tendermint-explained-bringing-bft-based-pos-to-the-public-blockchain-domain-f22e274a0fdb

[11] Chainspace: A Sharded Smart Contracts Platform, 2017: https://arxiv.org/pdf/1708.03778.pdf

[12] IPFS - Content Addressed, Versioned, P2P File System: https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf

[13] Peer-to-peer networking with BitTorrent, 2005: http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf

[14] Kademlia: A Peer-to-peer Information System Based on the XOR Metric: https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf

[15] Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, 1990: https://www.cs.cornell.edu/fbs/publications/SMSurvey.pdf