University
of Stavanger

VIDAR ANDRÉ BØ

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Reconfiguration of OnceTree

Master's Thesis - Computer Science - June 2024

```
func (m *Manager) NewConfiguration(opts ...gorums.ConfigOption) (c *Configuration, err error) {
    if len(opts) < 1 || len(opts) > 2 {
        return nil, fmt.Errorf("wrong number of options: %d", len(opts))
    }
    c = &Configuration{}
    for _, opt := range opts {
        switch v := opt.(type) {
        case gorums.NodeListOption:
            c.Configuration, err = gorums.NewConfiguration(m.Manager, v)
            if err != nil {
                return nil, err
            }
        case QuorumSpec:
            // Must be last since v may match QuorumSpec if it is interface{}
            c.qspec = v
        default:
            return nil, fmt.Errorf("unknown option type: %v", v)
        }
    }
    // return an error if the QuorumSpec interface is not empty and no imple
    var test interface{} = struct{}{}
    if _, empty := test.(QuorumSpec); !empty && c.qspec == nil {
        return nil, fmt.Errorf("missing required QuorumSpec")
    }
    return c, nil
}
```

I, **Vidar André Bø**, declare that this thesis titled, "Reconfiguration of OnceTree" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master's degree at the University of Stavanger.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

*“If debugging is the process of removing software bugs,
then programming must be the process of putting them in”*

– Edsger Dijkstra

Abstract

We designed and implemented a fault tolerant version of the tree-based OnceTree CRDT protocol using the Gorums framework. The recovery protocol was designed around using groups composed of a replica's neighbours to restore the tree structure, and thereby recover from a failure. The system was deployed to a data centre where its fault recovery performance was tested. As our testing demonstrated minimal service disruption under normal workloads, the system's performance should be good enough for many types of deployments.

Acknowledgements

I would like to thank my supervisor Hein Meling for his invaluable help and feedback. The discussions during our weekly meetings has undoubtedly shaped this thesis for the better.

I would also like to thank Leander Jehl and the rest of the staff and students at the Reliable Systems Lab at UiS. Their feedback during the development and writing process has been very valuable.

Contents

Abstract iii

Acknowledgements iv

Acronyms vii

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Approach and contributions 2
- 1.3 Outline 2

2 Background 4
- 2.1 CRDTs 4
- 2.2 OnceTree 4
- 2.3 gRPC 6
- 2.4 Gorums 6

3 Related work 7

4 Design and implementation 9
- 4.1 Design 9
  - 4.1.1 Challenges and limitations 9
  - 4.1.2 Group membership 10
  - 4.1.3 Move operations 11
  - 4.1.4 Reconfiguration message exchange 12
  - 4.1.5 State inheritance 14
- 4.2 Implementation 15
  - 4.2.1 Event bus 16
  - 4.2.2 Gorums provider 17
  - 4.2.3 Failure detector 17

4.2.4 Node manager 18
4.2.5 Storage service 18
4.2.6 Gossip sender 20

5 Experimental evaluation **23**
5.1 Experimental setup 23
5.1.1 Correctness testing 23
5.1.2 Performance testing 24
5.2 Experimental results 25
5.2.1 Correctness 25
5.2.2 Gossip throughput 25
5.2.3 Write throughput 26
5.2.4 Latency 28
5.2.5 Side effects of the gossip sender 32

6 Discussion **33**
6.1 Paxos similarities 33
6.2 OnceTree as a fault tolerant CRDT protocol 33
6.3 Complexity of fault tolerance 34
6.4 Consequences of increasing fanout during recovery 34
6.5 Gorums' suitability to OnceTree 34

7 Conclusion and future work **36**
7.1 Conclusion 36
7.1.1 Future work 36

A Source code **38**

Acronyms

BFT byzantine fault tolerant 8

CPU central processing unit 24

CRDT conflict-free replicated data type 1, 4

RAM random access memory 24

RPC remote procedure call 6, 17, 18, 21, 23, 34

Chapter 1

Introduction

1.1 Motivation

In the time of cloud services and infrastructure where the ability handle borderline unlimited traffic is expected, we sometimes end up in situations where one server is not enough to handle traffic from clients. Services are therefore horizontally scaled, essentially making copies (replicas) to handle more traffic or to facilitate fault tolerance [1]. Some of these solutions do however impose the limitation that only one of the replicas can handle write operations. Conflict-free replicated data type (CRDT)s can provide a solution to the problem of writing to different replicas of a service, but certain protocols have been problematic with regards to scaling to a large number of replicas [2].

Fault tolerance and recovery is of the utmost importance in distributed systems. In some systems, a failure at one of the replicas will render the system unable to make progress, so quick detection and recovery is essential. This could be the case for systems where replicas rely on communication with other replicas to process events. System halts can also occur in any application that overlays a logical network infrastructure on top of the physical one. In that case, the failed replica will effectively create a network partition, making sets of replicas unable to communicate with each other.

OnceTree [2] presents itself as a solution to some of these challenges. With its tree-based architecture and a focus on scalability to a large number of nodes, without the memory and transmission costs associated with existing approaches it can yield great performance for some workloads. However, a well defined reconfiguration and recovery process for when failures occur is yet to be defined.

1.2 Approach and contributions

In this thesis we have done the following:

- Implemented the OnceTree protocol using Gorums.
- Designed and implemented a method for tree restructuring.
- Designed and implemented a method for state inheritance.

Our implementation of the OnceTree protocol and its recovery mechanisms has been designed with the intention that it will act as a service that can scale up and down in number of replicas as required by the system load. The system has been designed around being able to handle large amounts of traffic without interruption and minimal service disruption under failure. The system design has therefore been dictated by these requirements or goals in mind:

- Fast tree path restoration after a failure.
- Minimise operations that block the progress of either the system or individual replicas.
- All nodes should be fully functional during a recovery process as long as a tree path exists.

The contribution of this thesis is finalising the OnceTree protocol design, making it fault tolerant. An implementation of the complete protocol is created using the Gorums framework.

1.3 Outline

- Chapter 2 covers the operation of the OnceTree protocol and touches on some other relevant background topics.
- Chapter 3 covers related work, and it relation to our objectives.
- Chapter 4 is split into two main sections, those being the design process and the implementation. The design process section will go into details of the design and why we made the design choices that we made. The implementation section will then cover how this was integrated into an application with some code examples to demonstrate the key parts.

- Chapter 5 covers how the correctness and performance of the OnceTree implementation and the accompanying reconfiguration protocol is evaluated.
- Chapter 6 discusses of how the design choices have impacted the results and how it might have been handled differently. It also discusses shortcomings of the protocol itself and some implementation challenges.
- Chapter 7 covers conclusion and suggest some future enhancements to the reconfiguration approach.

Chapter 2

Background

2.1 CRDTs

CRDTs are a way of replicating a state across several replicas without coordination. Updates happen without conflict and all updates are eventually applied to all replicas making the state consistent. An example of a simple CRDT is an add-only set. Since sets are unordered and do not contain duplicates, we can apply updates in any order to all replicas, and still end up with an eventually replicated state [3].

2.2 OnceTree

OnceTree is CRDT protocol that shares state in a tree-structured network of nodes. The core idea of the protocol is to have an O(1) storage requirement in an n-node network. It supports the data types that can be aggregated into a single item, e.g. sums, counters [2]. The nodes in the network will store values in fragments, one for each of its neighbours and one local fragment. The joined value of these fragments represents a stored state. A requirement for these fragments is that the combined fragments should require the same amount of memory as a single value. This limits the application to support values like counters if the mathematical properties of the protocol should be preserved. This by no means prohibits storing types like sets using the protocol, but the space-requirements of the update messages would not be constant. Each fragment (apart from the local) contains the aggregated value of the given neighbour's subtree. Since each node will have a unique position in the tree, the fragments will differ from one node to the next, while the aggregated value of these fragments will be identical. When changing a state,

the node will update its local state (its fragment), then send the aggregated state to all neighbours. The state that is sent to a neighbour will however not include the subset of the state that was received from that neighbour. This ensures that the state is not duplicated. When a node receives a state update, it will write the update to storage if the state is newer than the stored state. It will then forward the new state to all its neighbours except the origin of the update. This way we ensure that we create an infinite storm of messages. Throughout this thesis we will refer to client communication as ”read” or ”write” operations, while internal data transmissions between replicas are ”gossip” operations unless specified otherwise.

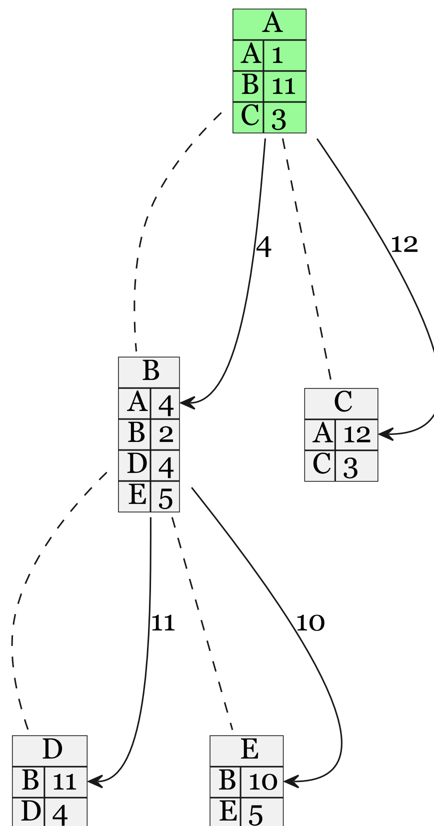

Figure 2.1: Gossip procedure

Figure 2.1 shows the state of each node and the flow of gossip messages in a simple 5 node network after the value stored at node A has been changed. At node A, the state has been changed to 1. A will have to send a gossip message to node B and C with this update. For node C, we combine A and B’s fragment and send the value 12, while for B we combine A and C’s fragment and send the value 4. Node C has no other neighbours than the message’s origin, so it

will not have to forward this update to any node. Node B however has D and E as neighbours and will send the values 11 and 10 respectively in the gossip messages.

2.3 gRPC

gRPC is a remote procedure call (RPC) framework that uses Protocol Buffers as a serialisation format. The general idea is that a client can call methods on a server over a network almost as if the server was a local object. gRPC provides the protoc tool which generates the aforementioned methods for the client to call based on the Protocol Buffer definitions [4].

2.4 Gorums

The application is built using Gorums [5], a framework built for simplifying gRPC calls between sets or quorums of replicas. Instead of the RPC client generated by the protoc tool in the normal gRPC implementation, Gorums provides configuration objects where RPCs can be performed on all or a subset of nodes. The configurations are provided by a manager which is responsible for maintaining the gRPC connections to the nodes. Gorums RPC calls can be invoked with a couple different modes of operation. Unicast sends messages to a single replica, essentially working as a standard RPC call. Multicast RPC calls are invoked on all the replicas present in the configuration, while ignoring their responses. Quorum RPC calls are also invoked on all replicas in the configuration, but a function specifying how their responses should be merged into a single response must be provided. Gorums will not create more than one gRPC stream to a server within a single client configuration. This enables the framework to ensure that messages arrive in order for application where that functionality is needed [5, 6].

Chapter 3

Related work

For the tree restructuring part of the reconfiguration, there have been published a couple of papers. Among those is ”Self Adjusting Binary Search Trees” by Daniel Dominic Slator and Robert Endre Tarjan [7]. There are however two core differences with how tree restructuring works for search trees and the tree structure in OnceTree. The first one is that there is no fixed entry point to the tree in OnceTree. Read and write operations can be performed on any node in the tree, and as long as the tree fan-out is not excessive, the performance of the read and write operations should be similar across nodes. Binary search trees however can benefit greatly if the most commonly accessed items are closer to the root of the tree. All restructuring that optimises for node positions and tree height differences are therefore only beneficial for value propagation latency, and not for outright speed. Secondly, changing a nodes position in the tree will have adverse effects on the shared state if not coordinated properly. It will also require re-transmitting much of the shared state, so particularly for large states, it is infeasible to move nodes often. All of this will be explained in detail in Chapter 4.

In addition to the OnceTree protocol itself, some tree re-configuration strategies were proposed by C. Power et al. [2]. These strategies do however require the tracking of much more state than strictly needed, as the replicas would track the state of neighbours two hops away. In a network with large fan-out and a large state, this would be problematic. Since one of the primary objectives of the protocol is to minimise memory usage, we elected to use an approach that only stores the state of direct neighbours. Another difference in the approach is that some of the replicas participating in the re-configuration will ignore updates from certain paths until the reconfiguration if complete, effectively stopping value propagation from one subtree. This is still partially

true for our implementation, but we don't ignore updates outright.

Kauri [8] is a byzantine fault tolerant (BFT) communication abstraction where nodes are organised in a tree structure. Its reconfiguration strategy is designed with the goal of creating a tree structure were there are safe edges between the leader and a quorum of correct processes. In Kauri however, the whole tree structure might change, while in OnceTree, the stored state is dependent on the tree structure and we can not freely move replicas without consequence.

Chapter 4

Design and implementation

4.1 Design

4.1.1 Challenges and limitations

Due to the way OnceTree stores and updates state, it imposes some limitations to how we can reorganise the tree structure. To illustrate this, consider a binary tree of replicas where the state in the tree is simply an integer, and node ”x” currently has the local state of 5 and is positioned as a leaf node in the tree. All other nodes have a local state of 0, meaning the aggregated state is also 5 across all nodes. As part of the reconfiguration strategy we will want to move the rightmost node ”x” in the tree to be a child of the leftmost node ”y”. If we just perform the move operation without any synchronisation, node ”x” will then begin to propagate the state 5 from its new position in the tree. As a result of this, nodes will have received the same state twice, meaning we have double counted the state. We will want to perform our reconfiguration so that we maintain the original properties of the protocol to the highest degree possible. This for instance means we cannot perform move operations that require external coordination of the nodes. We also do not want to have invalid state such as double counting of a state or missing state as a side-effect of the reconfiguration. We chose to design the reconfiguration process with the intent of inheriting the state of the failed node. This might not be applicable for all deployments of OnceTree, but it certainly can be useful.

To demonstrate how the reconfiguration works, we will use a binary tree with 15 nodes as depicted in Figure 4.1. For the demonstrations in this section, we will introduce a failure at replica 2.

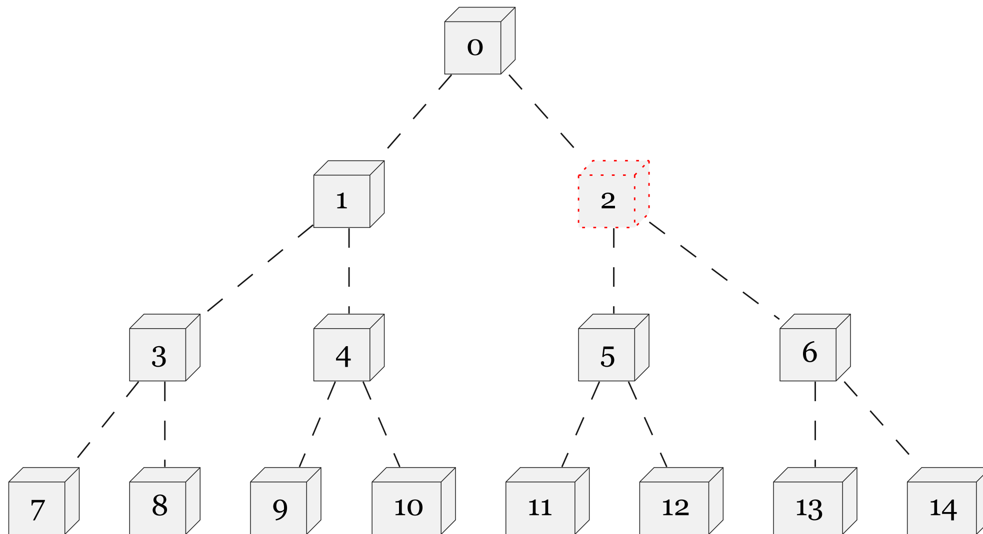

Figure 4.1: Before reconfiguration

4.1.2 Group membership

The reconfiguration process is based around each replica having a recovery group. This group consists of the closest neighbours of the replica. Replicas that are leaf nodes will only be a member of one recovery group, since it only has one direct neighbour. Replicas in the middle of the tree will be members of the recovery groups of its parent and its children. Whenever the neighbours of a replica changes, the replica itself it responsible for informing group members of who is a member of the group. When a replica sends out these updates, it includes an epoch number so that group members can ensure they have identical information of the groups composition. All neighbours of a replica will receive a heartbeat periodically, so all group members will be able to detect if a replica has failed. Whenever a replica fails, the recovery group is responsible for reconnecting the network to a spanning tree.

Figure 4.2 shows the recovery group of node 2 marked in green. Those are the nodes that are responsible for reconnecting the network to a spanning tree when node 2 fails.

With how the groups are setup, the system has information for how to reconnect the tree regardless of which node fails. What it cannot do is reconnect if two adjacent nodes fails at the same time.

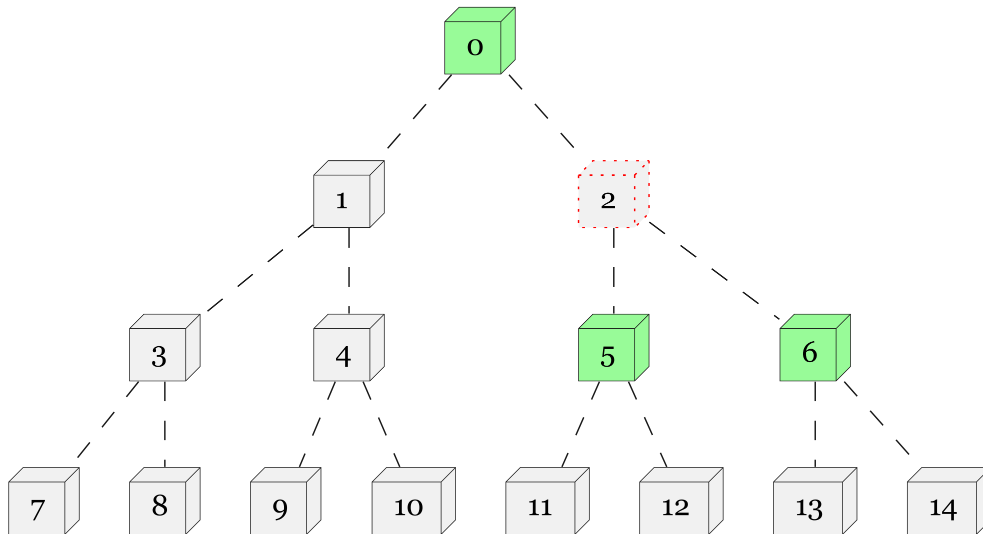

Figure 4.2: Recovery group for node 2

4.1.3 Move operations

Several mode operations are possible to reconnect the tree, but few will have a simple way of ensuring that we don't end up with invalid state. As explained in Section 4.1.1 we cannot simply move a node freely around in the tree without consequence. If we are to reconfigure the tree without synchronisation with the remaining parts of the tree, two methods emerge. We can either chain the nodes in the recovery group to each other with the leader as the first in the chain, or we can connect all nodes in the group directly to the leader. If keeping the fanout low is important, then node-chaining might be the best option since it will only increase the fanout by 1 for each node in the group except for the leader. The big caveat here is that the height of the tree will grow immensely for trees with large fanouts if we want to keep the fanout growth limited. It will also make the state inheritance process (described in Subsection 4.1.5) a multi-jump process, whereas it can be done as a broadcast when connecting all group members directly to the leader. We therefore chose to connect all group members directly to the leader. When the reconnection is completed the fanout of the leader will have increased by the number of nodes that were connected to the failed node. The performance implications of this will be shown in Chapter 5.

Figure 4.3 shows the network of nodes after the reconfiguration process has been completed.

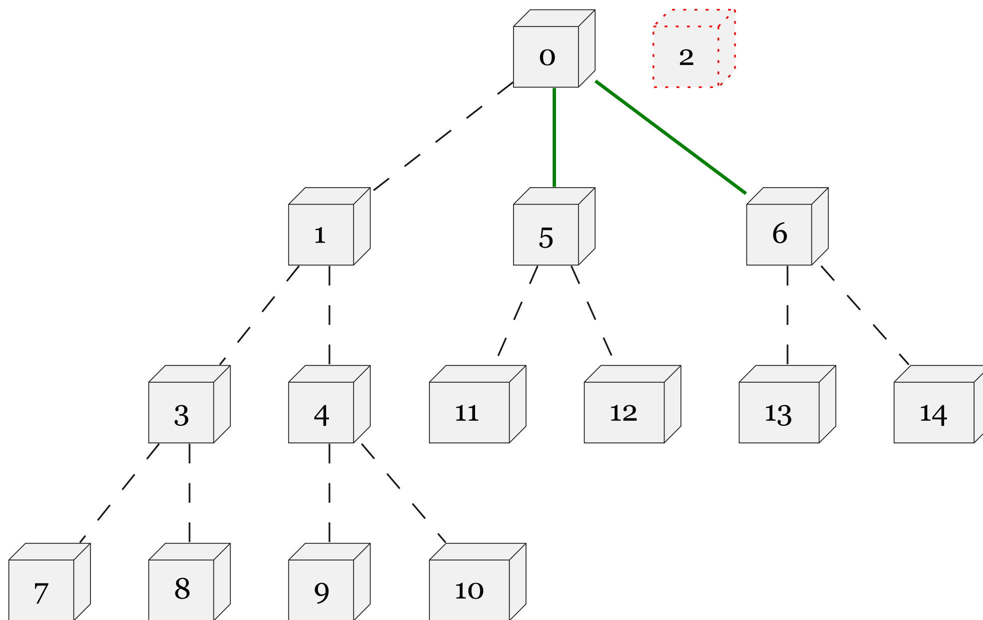

Figure 4.3: After reconfiguration

4.1.4 Reconfiguration message exchange

The message exchange of the reconfiguration phase draws inspiration from the message flow in Paxos [9]. All replicas start off sending a prepare message as a group multicast where the current epoch for the group is included. If epoch differs across replicas, then we know that the group information differs across replicas and we cannot successfully recover. The replicas will return a promise to the group member which has the highest position in the tree. Receiving a promise means that replica is the leader of the group. In the case where the root node fails, the group will sort the group members' ids and pick the first as the leader. Once a replica has received a promise from all group members it will enter the next stage as a leader.

The elected leader will in the next stage send out proposed tree changes in an accept message. Included in the message is a map showing which replica should connect to which. In our case, all map entries point to the leader, but another scheme is possible. The group members will respond with a learn message if the proposed changes are possible to perform. Once the leader has received a learn message from all group members, it knows all replicas can perform the changes and it moves on to the next stage.

Finally the leader will send a commit message to the group. This will make all replicas in the group apply the changes to the tree structure.

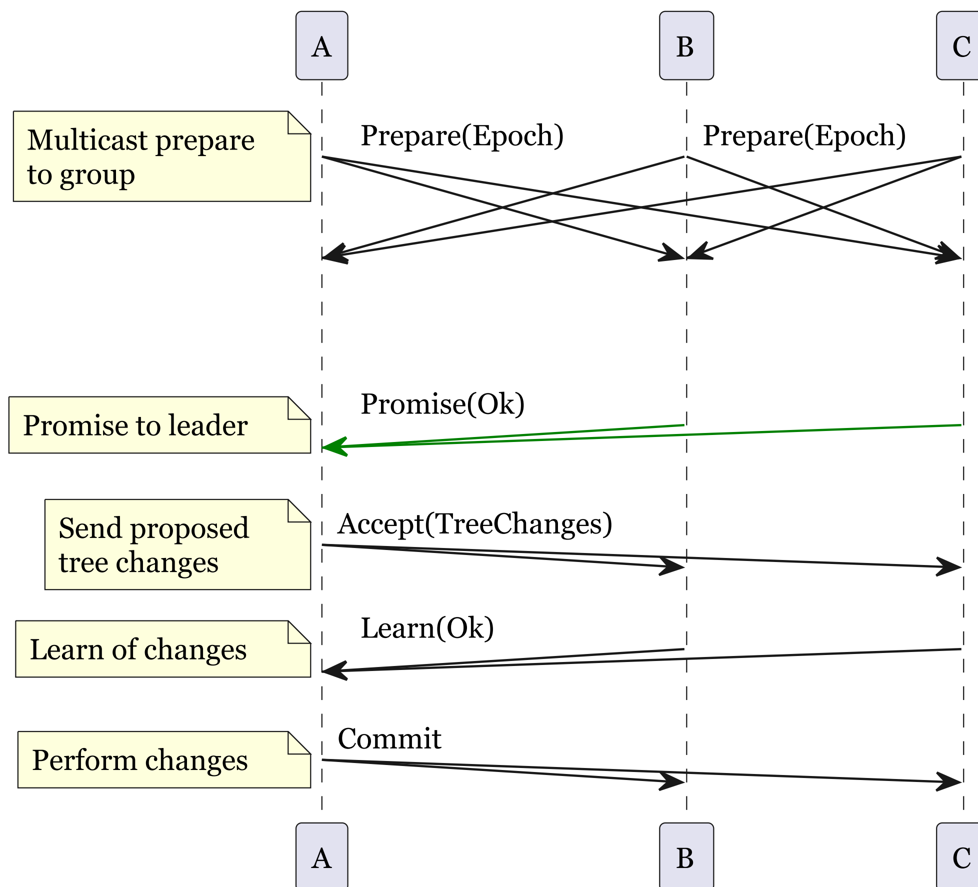

Figure 4.4: Successful tree reconfiguration

After the reconfiguration of the tree structure has occurred, the replicas' new neighbours will propagate updates through gossip messages as normal. These updates will be integrated into the state of the replicas, despite the state of the failed replica still existing. To avoid double counting the state, we need to impose the following ignore rules for the recovery group at this stage:

- Read operations will ignore the state from other replicas in the recovery group.
- Gossip messages to replicas in the recovery group will ignore the state of the failed replica.
- Gossip messages to replicas outside the recovery group will ignore the state of replicas in the recovery group.

These rules will persist until the local state of the failed replica has been integrated into the leader of the recovery group. Figure 4.4 shows the Paxos

inspired message flow for the reconnection process.

4.1.5 State inheritance

Since the state stored in OnceTree is dependent on the tree structure, we cannot completely remove the state of a failed node after the reconfiguration. If we did, we would also lose the state from all replicas that branch of the failed node until the new neighbours has propagated their state. As a consequence of this, the failed node will effectively be converted into a leaf node on replicas that were part of its recovery group (its closest neighbours before failure). An illustration of this behaviour is shown in Figure 4.5.

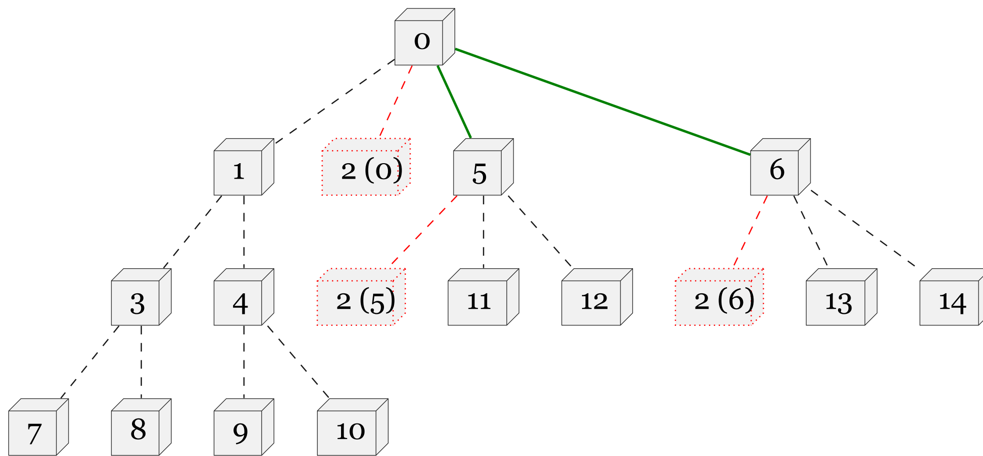

Figure 4.5: Logical tree state after reconfiguration

Whenever the leader of the reconfiguration process has finished reconnecting the tree, it will start the state inheritance process. It is the leader that will integrate the state of the failed node into its own state. In a non fault-tolerant implementation of OnceTree, the only state that a replica will ever receive from its neighbour is its aggregated state. To facilitate fault-recovery, we must however include its local state in the gossip messages as well. Replicas will store this data, but separate from the actual stored state as it will only be read as a part of the recovery process. It is theoretically possible to deduce the local state of the failed replica based on the aggregated state stored by its recovery group, but if one replica has received a newer state than the others, the computed state will be incorrect. The replicas in the OnceTree network would have to perform the first hop of the gossip operation atomically to its neighbours to ensure that all have the same state if it ever needs to be recovered. This is impractical for a number of reasons.

The procedure for state inheritance starts at the leader from the tree restructuring process. Note that the leader will include all its neighbours in this process, not just the recovery group. This greatly simplifies the process as we do not have to create custom Gorums configurations and trigger sending of gossip messages to nodes outside of the recovery group to keep those nodes up to date.

The leader will start by broadcasting a prepare message containing the timestamp of the failed node's local state. The replicas will respond with a promise message with a field set to true if they have a newer version of the failed replica's local state. Included in the promise will also be this newer version of the failed node's local state, and additionally the replicas local and aggregated state. Up until now, the leader does not possess its new neighbours' state unless it has received it through a gossip message, so including this is a necessity for the next stage. If the leader receives newer version of the failed node's state in the promises, it will integrate this into its own state.

Finally, the leader will integrate the failed replica's local state into its own state. At the same time it will remove both the local and aggregated state for the failed replica. It will also remove the read and gossip exclusions that were put in place at the earlier stages. The leader will then compute gossip values to send to its neighbours and send it as an accept message. Neighbours receiving this accept message will then update the state accordingly and remove all state stored for the failed replica including the aforementioned read and gossip exclusions. The failed replicas state has now been successfully inherited by the leader and then distributed. From this point, the neighbours will handle this as any other gossip message - compute and send gossip to next hop neighbours.

Figure 4.6 shows the Paxos inspired message flow for the state inheritance process.

4.2 Implementation

In this section we will describe the most relevant and interesting parts of the implementation. Up until now the state has mostly been referred to as one single unit, as this is how it is described by C. Power et al. [2]. To simulate a more real world scenario we chose to implement a key-value store. For the most part this only really adds a for loop around some operations and requires indexing with a key on others. All writes and gossips therefore happen on a key by key basis, and not the entire state.

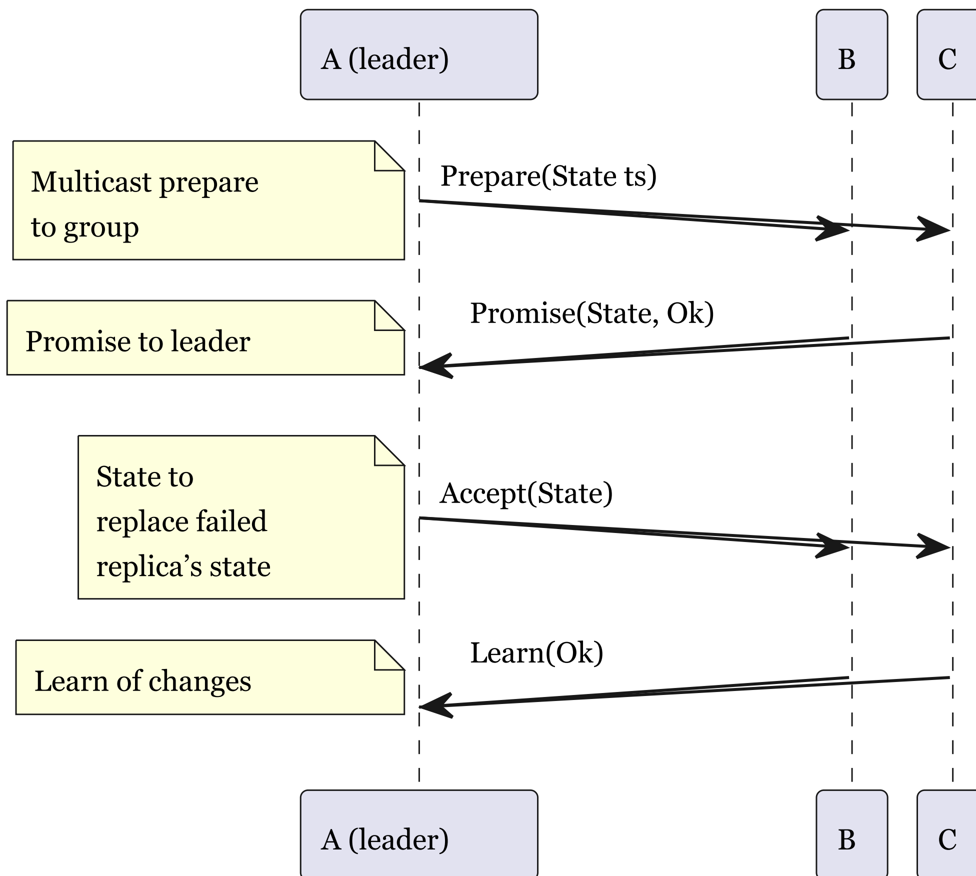

Figure 4.6: Message flow for value inheritance

4.2.1 Event bus

To notify of and handle cross-domain events, we implemented an event bus. Modules in the application are able to register event handlers, and the handler functions will be executed whenever an event of the appropriate type has been pushed to the event bus. The event type is extracted from the event using Go's reflect package. When an event is pushed to the event bus, the event bus will fetch all the event's handlers and execute them on by one. The event bus uses a fixed, configurable number of goroutines to execute these event handlers. This is accomplished by spawning goroutines at start up that listen to the pending events channel and execute the event handlers. An alternative approach to limit the degree of concurrency here could be Go's semaphore functionality and then spawn a new goroutine for each event handler instead.

4.2.2 Gorums provider

As of writing, the Gorums framework does not have a built in way of handling the configurations for multiple services [6]. We therefore created a package that given a collection of nodes as input, will provide the application with configurations for the various services. If a module in the application needs a configuration it will call a function on the provider which will return an appropriate configuration object.

When a node is removed from the OnceTree network, we not only need to remove it from configurations, but also from the Gorums managers. This is because it is the managers that actually maintains the gRPC connections, and we do not want to maintain a connection to failed or otherwise non-required nodes. The problem with that is that the only way to remove a node from a Gorums manager is to delete the manager itself, then recreate it with the desired nodes. Configurations that were created by this manager will as a consequence of this be deleted, something that might cause RPC failures to nodes that are otherwise healthy. To solve the problem of removing nodes, we simply implemented a scheduled deletion of the manager. It works by creating a new manager which will hand out new configurations when required and then spawning a goroutine which after a certain timeout deletes the old manager. As long as the rest of the application retrieves configuration from the Gorums provider when required instead of storing then for later use we can ensure that the configurations is actually functional.

4.2.3 Failure detector

Each node in the network will send out a heartbeat to each of its neighbours once every second. The failure detector is set up as its own service in the protocol buffer definitions, and has a multicast RPC enabled by Gorums called Heartbeat. The nodes maintain a map of the number of ”strikes” a given node has. A strike will be added once every second to each node. When a heartbeat is registered, the nodes strike count is set to zero. If the number ever gets above a certain threshold, a node failure event is published. In our deployment of the OnceTree network, this threshold was set to 5, but it might have to be set higher if the network conditions requires it.

The failure detector will publish a NodeFailedEvent to the event bus if a node has not sent out a heartbeat within the last 5 seconds.

4.2.4 Node manager

The NodeManager module is responsible for managing the node's local view of the tree. It will call the GorumsProvider to create configurations with the required neighbours whenever changes to the tree occurs. When the failure detector publishes a NodeFailedEvent, the NodeManager module will be responsible for initiating the recovery process. To keep complexity down we did impose a limitation in the NodeManager that it can only handle a failure with one group at a time. This way, keeping track of reconfiguration progress and ignore rules for the state is much simpler.

Join RPC

The Join RPC handled by the NodeManager and is used on startup by replicas joining the tree. Replicas will call the Join RPC on one of the already known replicas in the tree. This replica will check if its fanout has reached the maximum allowed and attach the replica as a child node if the fanout threshold has not been reached. If the fanout threshold is exceeded, it will return the address of one of its child nodes. To ensure a balanced tree structure, it will send the child addresses in an alternating fashion. Joining replicas will repeat calling the Join RPC until they are given a place in the tree. After a successful joining of the network, the pre-existing replica will trigger a share operation where it will transfer its state to the new replica.

4.2.5 Storage service

The StorageService module is responsible for handling the incoming read, write and gossip operations, writing to storage, and then send the appropriate gossip messages. It contains a key-value store, the struct for which is shown in Listing 4.1.

```
1 type KeyValueStorage struct {
2   data              map[string]map[int64]TimestampedValue
3   local             map[string]map[int64]TimestampedValue
4   readExclusions    map[int64]hashset.HashSet[string]
5   gossipExclusions  map[int64]map[string]hashset.HashSet[string]
6   mut               sync.RWMutex
7 }
```

Listing 4.1: KeyValueStorage struct

Note the read and gossip exclusions on lines 4 and 5. Each key in the key-value store will have its exclusions set after a reconfiguration has occurred and

will be checked during read operations and gossip calculations, the former of which is shown in Listing 4.2.

```
func (kvs *KeyValueStorage) ReadValue(key int64) (int64, error) {
  kvs.mut.RLock()
  defer kvs.mut.RUnlock()
  agg := int64(0)
  found := false
  for nodeID, values := range kvs.data {
    if value, exists := values[key]; exists {
      found = true
      // we skip the key for that node if it is excluded
      if excludedNodes, ok := kvs.readExclusions[key]; ok &&
    excludedNodes.Contains(nodeID) {
        continue
      }
      agg += value.Value
    }
  }
  if found {
    return agg, nil
  }
  return 0, fmt.Errorf("keyvaluestorage does not contain key %v",
    key)
}
```

Listing 4.2: Read process

Gossip value calculation is shown in Listing 4.3. It is for the most part the same as the read operation, except that we exclude the value stored for the gossip's destination. There is also a more complex exclude pattern during the state-inheritance phase as shown on line 10-14.

```
func (kvs *KeyValueStorage) GossipValue(key int64, destId string) (
    int64, error) {
  kvs.mut.RLock()
  defer kvs.mut.RUnlock()
  agg := int64(0)
  found := false
  for nodeID, values := range kvs.data {
    if value, exists := values[key]; exists {
      found = true

      // exclusions contain the key
      if destinations, hasExcludeForKey := kvs.gossipExclusions[key
    ]; hasExcludeForKey {
        // exclusions contain the destination of gossip
```

```
13        if excludes, hasExcludeForDestination := destinations[destId
    ]; hasExcludeForDestination && excludes.Contains(nodeID) {
14          continue
15        }
16      }
17      if destId != nodeID {
18        agg += value.Value
19      }
20    }
21  }
22  if found {
23    return agg, nil
24  }
25  return 0, fmt.Errorf("keyvaluestorage does not contain key %v",
    key)
26 }
```

Listing 4.3: Gossip value calculation

For completeness the write procedure is included in Listing 4.4. It does however not require any real changes to enable fault tolerance.

```
1 func (kvs *KeyValueStorage) writeValue(key int64, value int64,
    timestamp int64, nodeID string) bool {
2   if _, containsNode := kvs.data[nodeID]; !containsNode {
3     kvs.data[nodeID] = make(map[int64]TimestampedValue)
4   }
5   if storedValue, exists := kvs.data[nodeID][key]; exists {
6     if storedValue.Timestamp >= timestamp {
7       return false
8     }
9   }
10  kvs.data[nodeID][key] = TimestampedValue{Value: value, Timestamp:
    timestamp}
11  return true
12 }
```

Listing 4.4: Write process

When a leader has successfully reconfigured the tree structure, it will publish a TreeRecoveredEvent to the event bus. The StorageService module's handler for this event will then start the value inheritance process.

4.2.6 Gossip sender

The sending of gossip messages is by far the most common type of outgoing traffic in the application. These send operations need to happen concurrently

to ensure adequate performance, but there should also be some form of limit to the degree of concurrency. If there are no bounds to concurrency, network congestion will cause a large number of messages to be queued for sending. This can result in severely high memory usage and then potentially a crash. There is however a potential of deadlocking when limiting the number of concurrent send operations since the gossip process is recursive across nodes. The Gossip RPC takes the incoming data and writes it to memory if the data is newer than what is already stored. After the write operation, the node calculates what data to gossip to all its adjacent nodes except the message origin, and then spawns new send gossip operations. The Gossip RPC returns right after the send gossip operations are spawned. If we waited for the response from the next hop, the write latency would increase along with the height of the tree structure. To illustrate the problem with a bound degree of concurrency, take a tree of nodes and pick two adjacent nodes placed in middle of the tree heightwise. If these two nodes spawn a gossip operation to each other concurrently, and the number of concurrent send operations are bound (assume a limit of 1 for instance), there might not be enough free capacity for the receiving node to spawn its own new send operations. The node will block until there is a free slot, and you end up with a deadlock. An additional problem with this approach is that when a node fails, traffic to all other nodes will be held up by attempts to transmit to the failed node.

A way around the deadlock while still not having unbounded concurrency is to split the limitation by traffic origin. Essentially we impose a limit where we don't handle more gossip operations from one part of the tree until it has been propagated to the next-hop neighbours in the other branches of the tree. The limitation itself is simply one of Go's buffered channels making the RPC block until the new gossip message has been pushed to the channel. We utilise one channel per origin-recipient pair, but this is mostly so that we can have a single goroutine responsible for sending to each neighbour. It could have just as easily been a single channel per origin, and a sender which sends to all neighbours. The difference here is that if some replicas are slower than others, they might receive updates much later than the other neighbours, since the sender has a smaller backlog of gossip messages for those nodes. This is not problematic for the OnceTree protocol itself, as it is only supposed to provide eventual consistency.

```
1 func (gw *GossipWorker) sendGossip(gossip PerNodeGossip) {
2   cfg, ok, _ := gw.configProvider.StorageConfig()
3   if !ok {
```

```
4       gw.logger.Error("no cfg, skipping gossip",
5         slog.String("target", gw.target))
6       return
7     }
8     gorumsID, ok := gw.gorumsID(gw.target)
9     if !ok {
10      if !gw.hasSkipped {
11        gw.logger.Error("node not in manager, skipping",
12          slog.String("target", gw.target))
13      }
14      gw.hasSkipped = true
15      return
16    }
17    node, ok := cfg.Node(gorumsID)
18    if !ok {
19      gw.logger.Error("node not in config, skipping",
20        slog.String("target", gw.target))
21      return
22    }
23      .
24      .
25      //...send gossip
```

Listing 4.5: Gossip worker

When a failure is recovered from, the gossip message queue to the failed replica might be full, blocking the sending of gossips to other replicas. Listing 4.5 shows how the worker will quickly process messages if it can no longer find the target of the gossip message.

Chapter 5

Experimental evaluation

In order to test the performance and correctness of the application, we devised two types of tests. The first and simplest are integration tests using Go's built in testing functionality. These tests were used to verify the correctness. The second and more involved type of test is a deployment of the tree on a data centre, then collecting latency and throughput data during operation.

5.1 Experimental setup

5.1.1 Correctness testing

Software setup

We built our correctness testing around Go's built in testing functionality and Stretchr's testify package. For these tests, we created a collection of OnceTree replicas in their own goroutine where they listened for traffic on their own separate port on the loopback interface. The replicas were each provided with a specific address to join to when entering the network such that we had a consistent network layout between tests.

Testing methodology

The tests were based around creating some random data to write to each replica in the network, performing the write RPC calls, then checking that the changes were reflected in the network by performing read RPC calls. We wrote the following test scenarios:

- Let a replica join an existing network, then check that the new replica has received the existing state.

- Write a collection of random key-value pairs to all replicas, then verify that the aggregated values of those key-value pairs are correct.
- Write a collection of random key-value pairs to all replicas, then verify that the local values of those key-value pairs are correct, and only the required replicas store them.
- Write a collection of random key-value pairs to all replicas, fail a replica, and then verify that the correct replica has inherited the local state.
- Write a collection of random key-value pairs to all replicas, fail a replica, and then verify that the aggregated values of those key-value pairs are correct.

5.1.2 Performance testing

Hardware setup

Each replica in the OnceTree network was run on its own dedicated server. The servers each had an Intel Xeon E-2136 6 Core central processing unit (CPU) and 32 GiB random access memory (RAM). They were equipped with 10Gibps network interfaces and we typically observed less than 0.2 ms of round trip latency between servers. In total, 27 servers were used, 15 of those being OnceTree replicas and the remaining 12 being clients to generate traffic.

Software setup

To generate load on the system, we created clients that performed read and write operations on the cluster of OnceTree replicas. The clients were operating as either a reader or a writer and would would perform a new operation once every 1000 ms. Each client had a Gorums configuration with all of the replicas in the network. For each of the replicas, the client spawned a goroutine that was responsible for sending messages to that replica. We recorded latency of read and write operations on the clients, and recorded throughput metrics on each of the replicas.

The benchmark run consists of first writing a state with 5000 random key-value pairs to each of the replicas. When this phase is complete, we enter a 60 second phase of read and write operations at a constant rate where a failure is introduced for one replica at $t = 20$.

Test parameters

Number of write servers	Number of read servers	Instances per server	Request rate	Tree fanout	Gossip buffer size	Number of OnceTree replicas
3	9	1-10	1000/s	2, 4	10k	15

Table 5.1: Testing parameters

5.2 Experimental results

Where applicable, all plots will be labelled with total read and write load and fanout for that benchmark run. Throughput metrics are recorded at the replicas, so here the y-axis is a per-replica metric.

5.2.1 Correctness

The correctness testing was essential in the development phase of this thesis. Having integration tests to rely on when developing this type of system played a major role in ensuring a smooth integration of features. The list of tests was not comprehensive, but covered enough failure scenarios that we could feel reasonably confident that the system functioned as intended.

5.2.2 Gossip throughput

The gossip throughput numbers can give us an indication of whether or not the system can keep up with the internal demand. If the throughput starts off very high, then slows down a lot, it can indicate that the gossip message buffer was filling rapidly and that the replica is unable to transmit the outgoing gossip messages. Figure 5.1 show how the introduction of a failure at $t = 20$ impacts the rate of gossip operations. The system is here under a load of 12000 writes per second and 36000 reads per second. The plot shows the gossip throughput of all the replicas, and we can clearly see that some of them reaches a rate of 0 after about 20s into the benchmark. This is because some of the replicas are leaf nodes who's only connection to the rest of the network is through the failed node. The other replicas are able to process some traffic, as the message buffer was large enough to handle the short period of a missing

network path. As long as the gossip message buffers in the direction of the failed node is not full, a replica can send to the other neighbours.

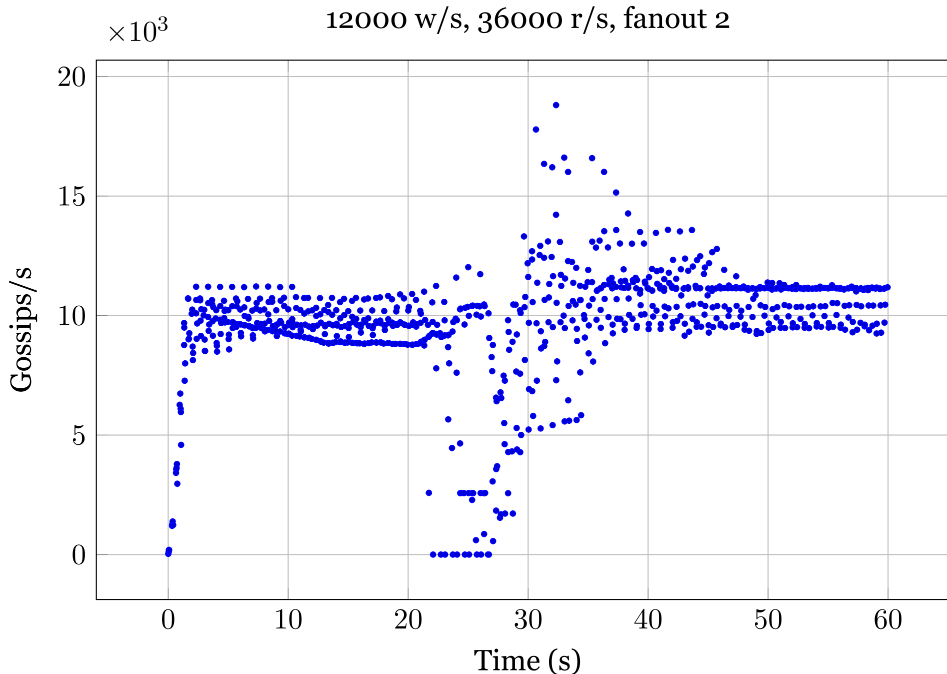

Figure 5.1: Per node gossip throughput

In Figure 5.2 we have doubled the load on the system. 24000 writes per second and 72000 reads per second was the highest that could reliably be handled by the system. We tested up to 30000 writes and 90000 reads per second, but the system was not able too keep up with that demand.

Note the quite clear separation of lines in the throughput graph, we will explain this in Subsection 5.2.5.

Changing the fanout to 4 in Figure 5.3 while keeping the same system load we see a clearer dip in the gossip throughput until the failure has been recovered. OnceTree deployments with larger fanouts are expected to perform worse in terms of throughput due to there simply being more replicas to transmit updates to for central tree nodes. Incoming traffic should occur in the same quantities regardless of fanout, just split across more neighbours.

5.2.3 Write throughput

The write throughput can give us an indication of whether or not the system can keep of with the external demand. Ideally the write throughput should be

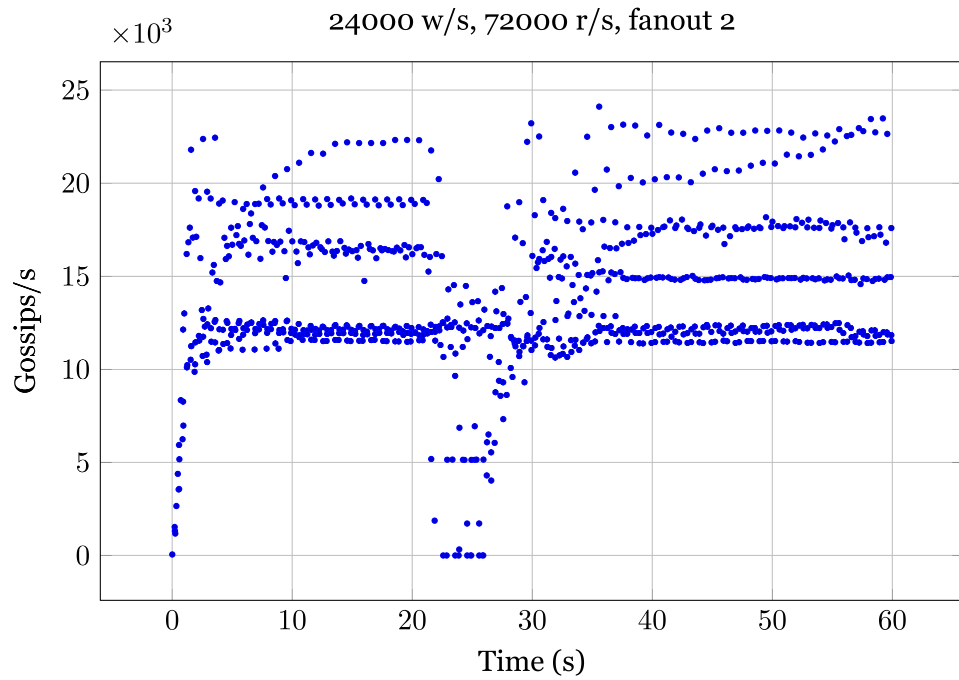

Figure 5.2: Per node gossip throughput

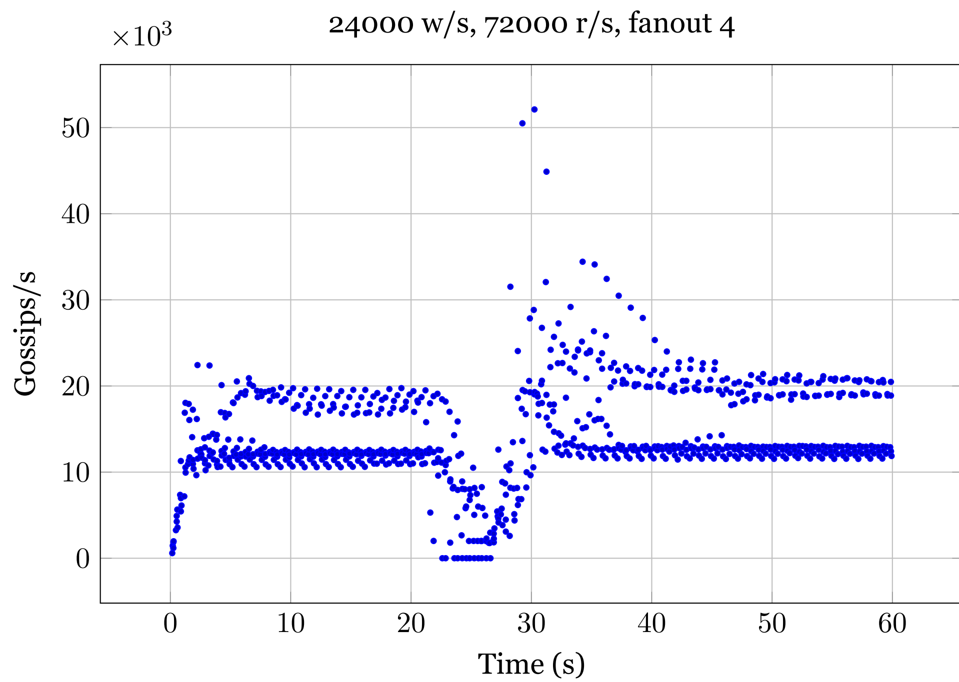

Figure 5.3: Per node gossip throughput

constant since the clients are providing a constant load evenly distributed to the system.

Figure 5.4 shows the write throughput each replica is experiencing under a 12000 writes per second system load. As there is one less replica in the system after 20 seconds, there is an increase in the write throughput. Apart from that, no other impact of the replica failure can be observed here.

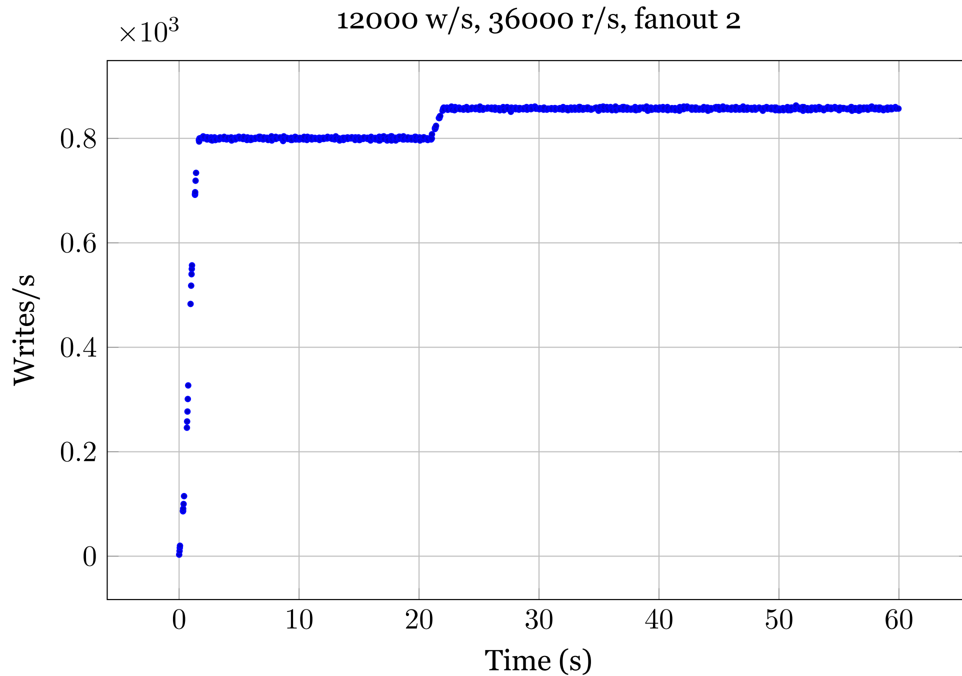

Figure 5.4: Per node write throughput

Figure 5.5 shows the write throughput when the write load is increased to 24000 writes per second. Here we finally see a real impact on the write throughput due to the failure. The culprit here is a full gossip message buffer in the direction of the failed replica at one or more of the failed replica's neighbours. When the reconfiguration is complete, the gossip worker quickly skips queued message to the failed replica, allowing other gossip messages to be pushed to the gossip send queues.

5.2.4 Latency

Figure 5.6 and Figure 5.7 show the write and read latency under a load of 12000 writes per second and 36000 reads per second. Under this system load, a failure's impact on the read and write latency is minimal.

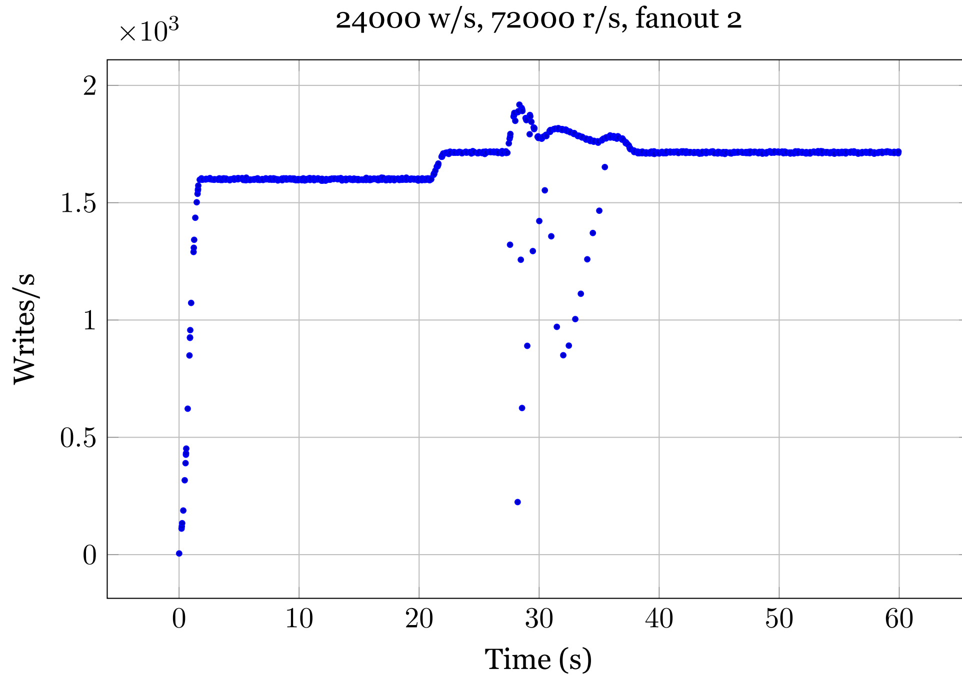

Figure 5.5: Per node write throughput

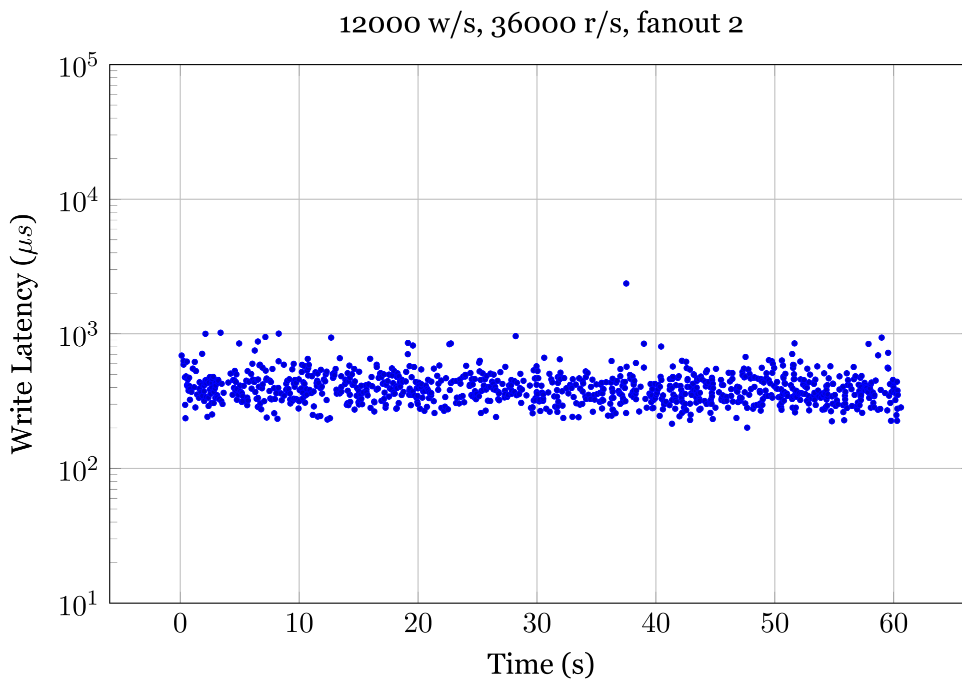

Figure 5.6: Write latency observed at clients

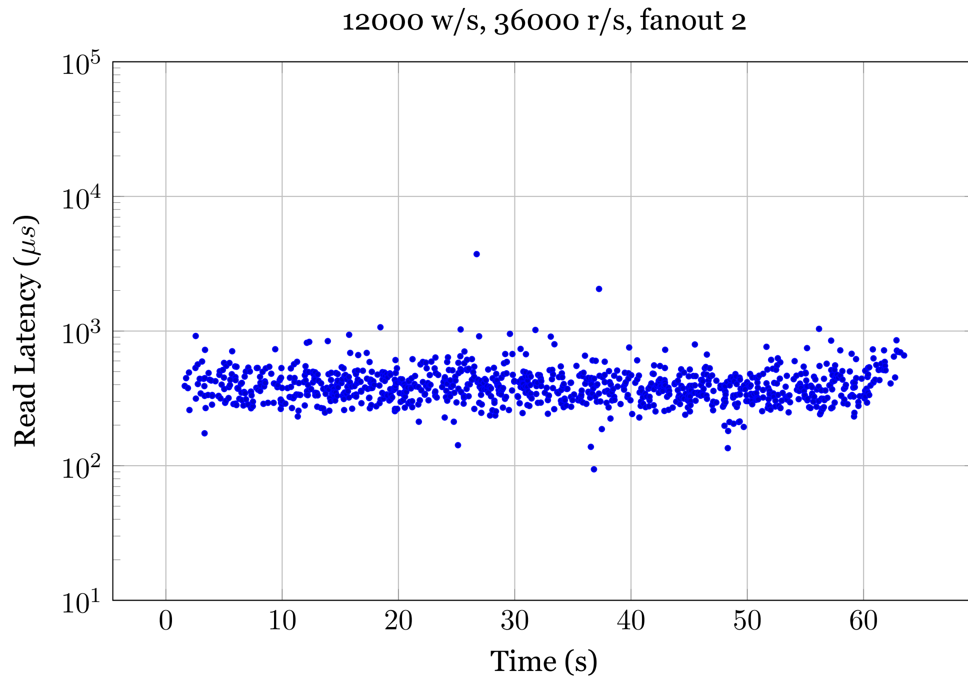

Figure 5.7: Read latency observed at clients

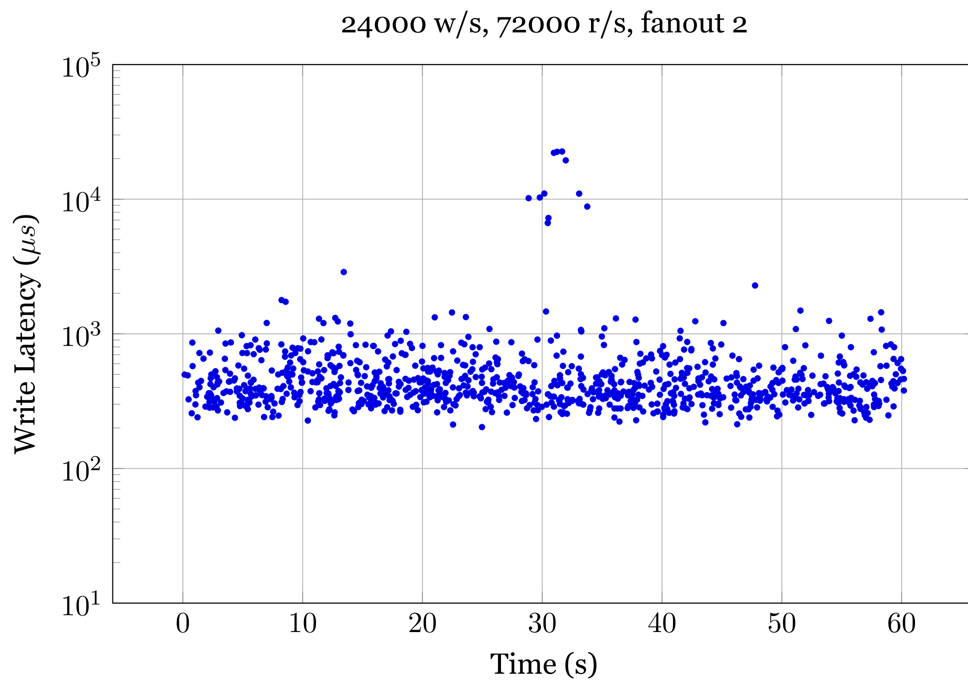

Figure 5.8: Write latency observed at clients

Increasing to 24000 writes per second and 72000 reads per second in Figure 5.8 and Figure 5.9 we start to see some impact of the failure. The clients are experiencing some spikes in write latency some time after the failure has been introduced, while the read latency remains largely unaffected. This is as expected, as the read operations never rely on communication with other replicas. Write operations will have a spike in latency due to the full gossip message buffer discussed in Subsection 5.2.2. The points on the latency plots are timestamped after the client has received its response, so this is why the spikes in latency appears so much later than when the failure is introduced.

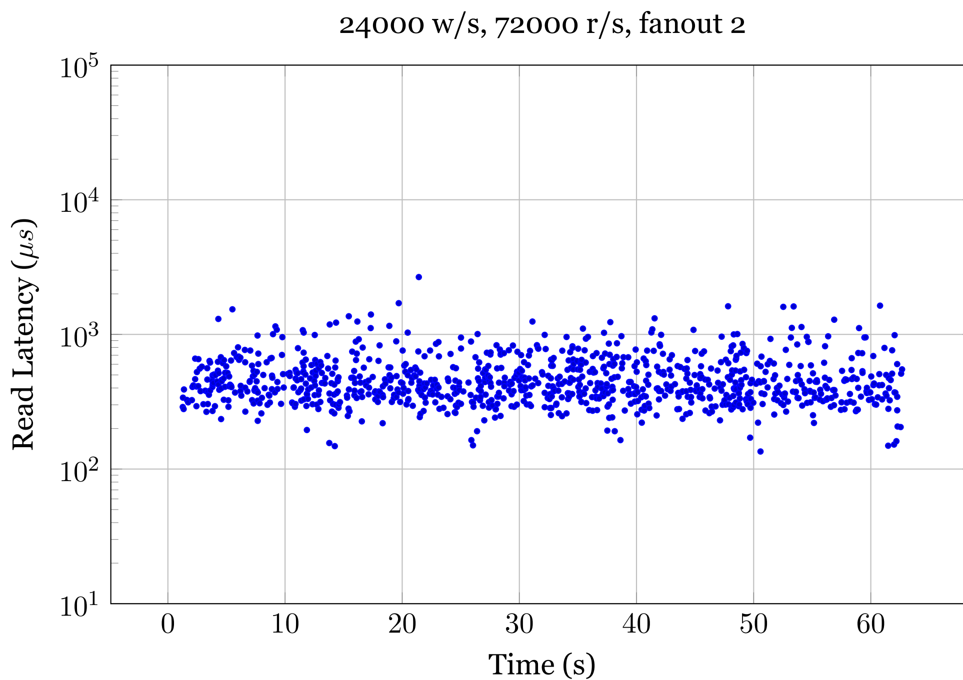

Figure 5.9: Read latency observed at clients

Figure 5.10 shows the impact that the system load has on the read and write latency. Apart from the 3 first data points which we do not have a good explanation for (even though it was consistent across benchmark runs), we generally see an increase in the latency as the system load increases. This is not a useful statistic for the reconfiguration process itself, but more of an indication of the performance under normal operation.

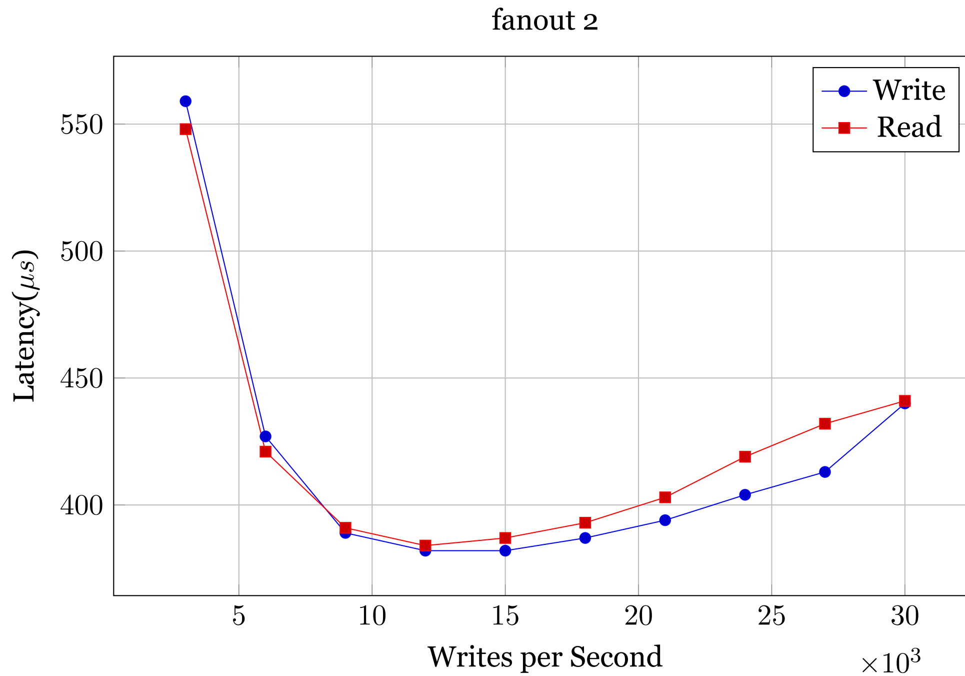

Figure 5.10: Read and write latency at varying load

5.2.5 Side effects of the gossip sender

As described in Section 4.2.6, the GossipSender module has one queue for each origin next-hop pair. This means that, especially under high load, gossip messages might not be sent to the next hop in the same order that the state was updated. If the gossip messages that got shuffled was for the same key in the state, the one with the lowest timestamp will get discarded at the next hop. This is why we see distinct levels of gossip throughput on the graphs in Subsection 5.2.2. Effectively what's happening is an accidental batching of updates because of the reordering. In a scenario with a slow network and a large send buffer, we could also manually inspect the messages, and discard updates if a later update for the same key is queued.

Chapter 6

Discussion

6.1 Paxos similarities

In the start of the development process we imagined that the similarities to Paxos were more numerous than what it ended up actually being. Paxos generally operates under the constraint that a majority of replicas need be involved in the consensus process [9]. For OnceTree however we will always need full participation to ensure that the protocol functions as intended. Since a single node is able to create a network partition, we cannot allow only a majority to participate and still make progress. This is especially true for the reconfiguration processes, where if a single replica is not included, the whole state of the tree might end up incorrect, or a portion of the tree is not reconnected.

6.2 OnceTree as a fault tolerant CRDT protocol

If fault tolerance is of the highest importance, as opposed to minimising memory footprint, then OnceTree with its tree based architecture might not be the best option. As there are no redundant links to other replicas, the worst case scenario is that we can disconnect one half of the network from the other half if the root node fails (for a tree with fanout of 2). The system we implemented only appears to still function properly to the clients because of the built in buffers, but there is potentially a massive backlog whenever the tree is restored. A protocol that would allow redundant links would not have the same problem, but likely at some other cost.

6.3 Complexity of fault tolerance

During the development process of a fault tolerant OnceTree implementation we noticed how fault tolerance impacts the complexity of the application. Algorithms such as the OnceTree algorithm that are so simple they can be explained in 20 lines of pseudocode [2] can become unwieldy by the introduction of fault tolerance. As a simple example, the GossipMessage used in the Gossip RPC (the majority of internal traffic) has to include two extra fields of data just to facilitate fault tolerance.

Although we definitely perceive the code as more complex, it is difficult to quantify the complexity vs a non fault tolerant version. This is because we likely would not design the applications modules the same way for a fault tolerant as for a non fault tolerant version of the protocol. Nevertheless, a substantial amount of the application modules' code such as in the NodeManager could be removed outright if it did not require fault tolerance.

6.4 Consequences of increasing fanout during recovery

A consequence of moving all replicas in the recovery group up to be children of the leader will make the leader's fanout increase. From our testing with a relatively low number nodes and fanouts of 2 and 4, we observed little to no impact on the metrics we collected in the period after the fault had been recovered from. That's not to say that it won't make an impact if the fanout had been larger to begin with, but we would argue that unless end to end latency is exceptionally important, fanout should be kept low. A low fanout will offer better throughput as we explained in Subsection 5.2.2, and fewer nodes have to be involved with recovery in the event of a node failure.

6.5 Gorums' suitability to OnceTree

The use of Gorums for this OnceTree implementation demonstrated some of its strengths and weaknesses. Some of Gorums's abstractions are not really suited for most of the communication that occurs within a OnceTree network. This mostly comes down to gossip messages being unique for each replica, so Gorums's broadcast-like abstractions cannot be used as is. Gorums does support providing a unique message to each node into the broadcast abstrac-

tion [6], but at that point it is often just as simple to iterate over each replica and send the message as unicast to it. Message types like heartbeats are however trivially simple with Gorums, since all messages are the same.

Chapter 7

Conclusion and future work

7.1 Conclusion

In this thesis we have designed and implemented a fault tolerant version of the OnceTree CRDT protocol using the Gorums framework. The recovery process was designed with a goal of minimal service disruption, while preserving the strengths of OnceTree. We based the recovery process around groups of adjacent nodes in the tree who are responsible for reconnecting the tree in the event of a failure. The recovery process also handles the inheritance of the failed node's state, so no state is lost due to failures. Our testing showed that the system was capable of recovering from a fault while under a constant load, with minimal service disruption. It also demonstrated that tree structure changes caused by our reconfiguration process yielded negligible impact on the systems performance.

7.1.1 Future work

Subtree rebalancing

As the protocol currently operates, if the tree goes through many reconfiguration, some parts of the tree might have a much higher fanout than intended. A possible solution to this could be to initiate a recursive leave and re-join operation on the subtree that is imbalanced. Only leaf nodes will leave the network, so the recovery operation when leaving will be inexpensive since no network transmission except for one round of gossip is needed.

Generic data types

An extension to support any data type could be useful, as the application is currently limited to integers. The most challenging part of this would be to encapsulate the data inside a protobuf message, since this would require manual serialisation and deserialisation. As long as the data supports a merge operation it should be possible to integrate into the system.

Appendix A

Source code

Source code for the OnceTree implementation as tested can be found at https://github.com/vidarandrebo/oncetree/tree/v1.0

Bibliography

[1] The PostgreSQL Global Development Group. PostgreSQL documentation. [Online]. Available: https://www.postgresql.org/docs/current/high-availability.html

[2] C. Power, S. Laddad, C. Douglas, S. Achalla, L. Katahanas, J. M. Hellerstein, and D. Suciu, "Once upon a tree: Distributed idempotence in o(1) space."

[3] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer, pp. 386–400.

[4] gRPC Authors. What is grpc. [Online]. Available: https://grpc.io/docs/what-is-grpc

[5] T. Lea, L. Jehl, and H. Meling, "Towards new abstractions for implementing quorum-based systems," pp. 2380–2385.

[6] H. Meling, J. I. Olsen, T. E. Lea, and L. Jehl. Gorums github page. [Online]. Available: https://github.com/relab/gorums

[7] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," vol. 32, no. 3, pp. 652–686. [Online]. Available: https://doi.org/10.1145/3828.3835

[8] R. Neiheiser, M. Matos, and L. Rodrigues, "Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. Association for Computing Machinery, pp. 35–48. [Online]. Available: https://doi.org/10.1145/3477132.3483584

[9] L. Lamport, "Paxos made simple."

4036 Stavanger
Tel: +47 51 83 10 00
E-mail: post@uis.no
www.uis.no

Cover Photo: Hein Meling

© 2024 Vidar André Bø