Miguel Ramos Duque

Licenciado em Engenharia Informática

Specification of a partial replication protocol with TLA+

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática

Orientadora: Carla Ferreira, Prof. Auxiliar,
Universidade Nova de Lisboa

Júri

Presidente:	Doutora Maria Cecília Farias Lorga Gomes
Arguente:	Doutor Manuel Alcino Cunha
Vogal:	Doutora Carla Maria Gonçalves Ferreira

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2015

Specification of a partial replication protocol with TLA+

Este documento foi gerado utilizando o processador (pdf)LaTeX, com base no template "unlthesis" [1] desenvolvido no Dep. Informática da FCT-NOVA [2]. [1] https://github.com/joaomlourenco/unlthesis [2] http://www.di.fct.unl.pt

Acknowledgements

I would like to start by thanking my adviser Carla Ferreira for all the support, availability and constant meetings throughout this year which were essential to the development of this work.

I would like to thank all my friends and colleagues for all the support, motivation, and countless hours spent working together.

Additionally, I would like to thank my parents and aunt Mena for all the support and for always believing in my capabilities.

Finally, I would like to especially thank Hélder Pita for all the guidance, support and encouragement throughout my academic journey.

Abstract

Nowadays, data available and used by companies is growing very fast creating the need to use and manage this data in the most efficient way. To this end, data is replicated over multiple datacenters and use different replication protocols, according to their needs, like more availability or stronger consistency level. The costs associated with full data replication can be very high, and most of the times, full replication is not needed since information can be logically partitioned. Another problem, is that by using datacenters to store and process information clients become heavily dependent on them. We propose a partial replication protocol called ParTree, which replicates data to clients, and organizes clients in a hierarchy, using communication between them to propagate information. This solution addresses some of these problems, namely by supporting partial data replication and offline execution mode. Given the complexity of the protocol, the use of formal verification is crucial to ensure the protocol two correctness properties: causal consistency and preservation of data. The use of TLA+ language and tools to formally specificity and verify the proposed protocol are also described.

Keywords: Partial replication, hierarchy, causal consistency, protocol specification, formal specification, TLA+

Resumo

Atualmente, a informação disponível e usada pelas empresas está a crescer a um ritmo muito elevado, e por isso há a necessidade de usar e gerir essa informação da forma mais eficiente possível. Para isto, as empresas replicam a informação em vários centros de dados e usam diferentes protocolos de replicação, dependendo das suas necessidades. Os custos associados com a replicação total de um conjunto de dados podem ser bastante elevados, e muitas vezes a réplica total não é necessária, ou seja, a informação pode ser dividida de forma lógica. Outro problema é que ao usar os centros de dados para guardar e processar informação, os clientes ficam bastante dependentes destas estruturas. ParTree é o protocolo proposto como solução. Consiste num protocolo de replicação parcial que replica os dados para os clientes. Os clientes estão organizados numa hierarquia de forma a permitir a comunicação entre clientes mesmo quando os servidores possam estar offline. Devido à complexidade do protocolo, a sua verificação formal é crucial para garantir que as propriedades desejadas são garantidas, consistência causal e que os dados não são perdidos. Essa verificação foi feita através da especificação do protocolo com a linguagem TLA+. A linguagem e as ferramentas usadas também foram descritas nesta dissertação.

Palavras-chave: Protocolo de replicação, replicação parcial, consistência causal, especificação formal, verificação formal, TLA+

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Nowadays software systems provide services that require shared mutable data at a global scale. To achieve high availability, data is replicated into multiple data centers so it remains closer to its users. In this setting, is a well-known technique to improve availability. By maintaining consistent replicas of a database, one can improve its fault tolerance, offer low latency, and simultaneously improve system's performance by splitting the workload among replicas [33]. When developing a distributed system protocol, many trade-offs must be considered, namely the replication degree of data, the communication topology, and the properties that a system can offer and guarantee. Therefore, the proposed work aims to offer a new replication protocol, that addresses some of the drawbacks of existing replication protocols.

1.2 Problem Description

In a globalized organization, the majority of its data is not relevant for all of its branches. Given that, the concept of using full datastore replicas is questionable. By only replicating the relevant data, the required network bandwidth and local storage at each replica can be reduced. Another problem consists on the necessity to improve the responsiveness of applications. Geo-replication is a commonly used mechanism to bring the data closer to the clients. However, even with this mechanism, reaching the closest datacenter can still be considerably slow [31]. Security is also an issue that should be considered. By creating full replicas, sensitive data can be more prompt to attacks, especially when considering replication to clients. Partial replication addresses the problems aforementioned, however, the use of partial replication introduces new challenges not raised by

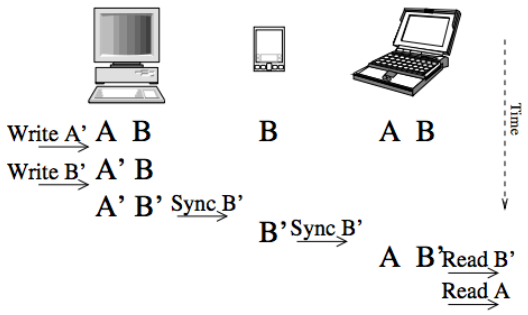

Figure 1.1: Loss of causal consistency with partial replication [4]

full replication.
One important issue is the guarantee that no data is lost. Where full replication only needs to assure that, at least, one node is available, partial replication (where each node of the system can contain a datastore with different data objects) must assure the availability of enough nodes to cover all data set (objects should not be only replicated on one node). Another side of this problem is the ability of nodes to choose the objects of interest. Protocols must prevent all nodes to drop one object at the same time, which would lead to do disappearance of that object from the system.
Other important problem aggravated by partial replication is the ability to offer data consistency between nodes. Some measures must be taken in the communication topology to assure that the desired consistency can be provided. Figure 1.1 describes an example of a loss of causal consistency due partial replication. In the example, three devices were used, a laptop and desktop computer with objects A and B, and a mobile phone with only B. The example starts with the desktop updating object A, followed by an update of object B. When the mobile phone synchronizes with the desktop, B is updated and everything is as desired. The problem arises when the laptop, that is interested in objects A and B, connects with the mobile phone that can only provide the update on B. This leads to a situation that should not happen, B updated before A. If causality between operations was to be respected, the update of B should always follow the update of A.

1.3 Proposed Solution

To solve the presented problems, a partial replication protocol was developed and verified. This protocol offers the following properties:

- nodes are organized in a tree hierarchy, where each child has a subset of the data of his parent.
- The root of the hierarchy is a full datastore replica, and sibling nodes have disjoint data partitions;

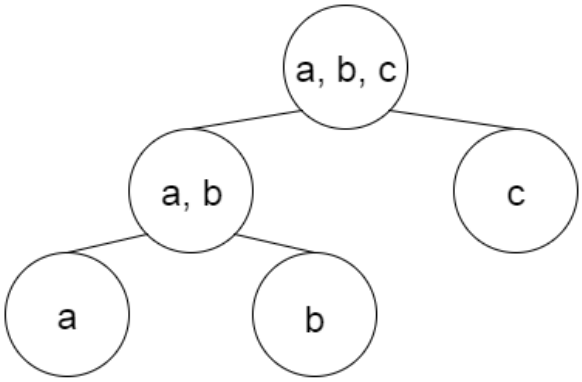

Figure 1.2: Example of a valid hierarchy.

- dynamic replication, i.e, instead of fixed number of replicas, this protocol allows operations for insertion and removal of nodes. Insertion operations will be directed to the root of the hierarchy;
- the data is stored in a key-value datastore, i.e, data will be represented as a collection of key-value pairs, such that each possible key appears at most once in the collection. This type of datastore supports two different operations, *put(key, value)* and *get(key)*;
- the operations of the datastore (*put* and *get*), can be executed anywhere in the hierarchy;
- ability to create keys anywhere in the hierarchy. The creation of keys is propagated to the top of the hierarchy.

With these properties, the system will be able to guarantee that no data is lost, causal consistency, eventual convergence of the data on the different nodes, and fault tolerance. Causal consistency is achieved by guaranteeing causality between related operations (related operations are the ones executed by a node), and fault tolerance by guaranteeing that in the event of a hierarchy node failure, the protocol will recover and keep working properly. Figure 1.2 is an example of a valid hierarchy with 3 keys.

The two main decisions about the architecture of the protocol were the node organization topology, and data partition strategy. The first decision (organization of nodes in a tree hierarchy) was due to its aid on offering causal consistency, by only allowing nodes to communicate with their connections, and the guarantee that only one of those connections will have a wider data partition. The fact that nodes only communicate with others who share their data interest, allows nodes to only receive updates of data on their interest set, avoiding the need to propagate unnecessary messages, and keeping large amounts of metadata. Other know topologies like a ring topologies where each node has exactly two connections, and a graph topology where each node is connected to a subset of other nodes, can be compared with the tree hierarchy. Although the ring topology

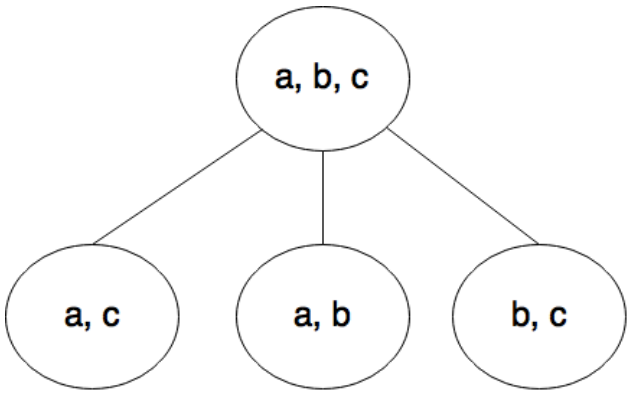

Figure 1.3: Example of hierarchy not forcing disjoint datastores between children

might require less metadata than the three hierarchy, its efficiency in partial replication protocols is inversely proportional to the number of nodes in the ring, due to the necessity to propagate all updates through all nodes (each update ends up propagating several unnecessary messages), which is not desirable when developing a protocol that might operate with many nodes. On the other hand, a graph topology requires nodes to keep and maintain large amounts of metadata to know to whom propagate updates. Every time a node enters/leaves the hierarchy, several nodes will need to change their metadata to guarantee the correct propagation of updates. The second decision, regarding the disjoint data partitions of siblings, was taken as a matter of efficiency, without this condition the protocol would still work, however, it could lead to a situation where the hierarchy would be inefficient and irrelevant. Without this condition two options would be available:

- the hierarchy would still be based on the datastore of each node, i.e, a node would still contain all data of its children. This option could lead to a situation represented in Figure 1.3, where all nodes would have datastores not contained on any other node's datastore, which would generate a two level hierarchy, with all nodes being children of the root. An hierarchy of the developed protocol would not allow the three children of the root to exist simultaneously, only one could exist;

- the hierarchy would not be based on the datastore of each node. With this option, in order to guarantee that nodes received the required updates, an update would have to be propagated through all nodes of the hierarchy, or metadata would have to be stored to identify the interested nodes.

1.4 Contributions

The presented work will have two contributions:

1. a new partial replication protocol that organizes nodes in a tree hierarchy, guaranteeing causality between related operations, convergence of data, and that no data is

lost. This protocol was developed to focus on a hierarchy between clients, to allow direct communication between clients, while keeping a server on the root of the hierarchy;

2. A formal specification of the protocol with TLA+, and the verification of its correction with the TLC model checker.

1.5 Document Structure

This document has the following structure:

Chapter 2 describes some important properties regarding data replication, and the tool used to develop the proposed protocol.

Chapter 3 describes several replication systems. Those systems implement some properties explained in Chapter 2

Chapter 4 presents the developed protocol. Explains its properties, which information is kept by a node, and the available operations assisted with pseudocode.

Chapter 5 presents the specification of the protocol with TLA+.

Chapter 6 explains how each property of the protocol is verified, and shows the tests performed to verify the correctness of the protocol.

Chapter 7 presents the conclusions of the thesis, compares the developed protocol with similar protocols, discusses its scalability, and future work.

Chapter 2

Background

The study of the area highlighted some properties on which all the related work is based on. This chapter starts by explaining those properties and then describes the tool used to develop the proposed protocol.

2.1 Replication properties

2.1.1 Replication Policies

When implementing a replication policy on a distributed system, two main solutions arise, a centralized replication policy, and a distributed replication policy. In a centralized replication policy a single node of the system is in charge of all replication decisions, therefore is easy to implement but likely to become a bottleneck. On the other hand, in a distributed replication policy multiple decisional nodes exist in the system, typically all nodes of the system[8].

The data replication can be made in a static or dynamic policy [8, 14]. These policies are used to automatically decide when and where an object has to be replicated in the system. Both policies have to decide the replication degree of a given object (number of physical replicas to be provided at a given time) and the allocation of the replicas within the system (the nodes of the systems in which replicas must be physically stored). A static replication policy decides these properties at the object creation, where a dynamic replication policy (also known as adaptive replication), takes those decisions during the object lifetime. In order to improve availability, these properties vary depending on the patterns of access.

In order to always grant availability and avoid loosing data, systems take different actions to handle nodes failures. Reactive and proactive replication are different approaches that systems can implement to deal with that situation.

Reactive replication systems react to failures. These systems define a threshold of the minimum amount of each object's replicas, and if it is surpassed due to node failures, the object is replicated in another node. Reactive replication minimize the total bytes sent since it only creates replicas as needed, however, after a failure, the network use can increase significantly, making it difficult to provision bandwidth [32].

On the other hand, *proactive replication* systems attempt to predict node failures and increase the replication level of objects that will have few replicas following a predicted failure [32]. Nodes are always replicating a new object, however, the choice of which object to replicate can generate some problems. Since objects are not uniformly distributed, after a failure, some nodes may end up replicating more objects than others. Although these systems avoid bandwidth spikes, a sequence of incorrect predictions can lead to data loss.

2.1.2 Replication Schemes

Considering the relationship between logical and physical replicas, two main schemes are possible. The active replication scheme, in which all replicas are considered at the same level and accessed without any preference (more focused to achieve high-performances); and the passive replication scheme, where there is a difference between a primary replica and a set of backup replicas to be used when they primary is not available, which is more oriented to fault tolerance.

Geo-replication is used to improve the distribution of data across geographically distributed data networks. To achieve this, systems replicate data across multiple datacenters. By maintaining data replicated, client can contact the closest datacenter, which leads to a decrease of latency and distribute the work between several datacenters. Geo-replication is specially used by globalized organizations.

2.1.3 Consistency

Depending on the consistency model implemented by a system, a read of an entity on different sites can show different values on different sites. While systems with strong consistency will show the same value, independently of the site where the read was made, weak consistency models might present different results. Before explaining the different consistency models, one property, called causality, must be explained, due it's importance to the weak consistency models.

Causality describes the dependency between operations. Systems that guarantee this property, ensure that operations are only applied, and therefore become visible, when all

versions of objects in their causal history have been applied, i.e, operations are applied in the same order everywhere. Causality between operations is transitive, for operations a, b, and c, if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$. If operations are not causally related, then they are concurrent under delivery order.

Now, the most relevant consistency models will be explained:

- Eventual Consistency – weakest consistency model because every node can independently apply operations on any order. All operations are independent of each others. With this model replicas can temporarily diverge due to concurrent updates at different sites, but they are guaranteed to eventual converge. During normal execution reads at different sites can return different values, but eventually will return the same value (an eventually consistent system can return any value before it converges [12]). This model is used with optimistic replication and is usually used with systems that want to offer high availability.
- Causal Consistency – ensures partial ordering between dependent operations, two causally related events must appear in the same order on every site. Causally consistent replication's weakness is that it doesn't prevent conflicting updates. Conflicts can make replicas diverge forever. Different servers/datacenters with conflicting updates need to communicate and use at least one round-trip-time in order to detect and resolve these conflicts. In some cases humans are needed to solve conflicts. Thus, causal consistency is not suited for systems that must maintain a global invariant. Causal consistency does not order concurrent operations, two unrelated operations can be replicated in any order, avoiding the need for a serialization point between them (normally, this allows increased efficiency in an implementation).
- Causal+ Consistency [19] – is a variation of causal consistency that is defined by causal consistency with convergent conflict handling. This requires all conflicts to be handled in the same manner at all replicas, using a handler function. This function must be associative and commutative, so that replicas can handle conflicting writes in the order they receive them and that the results of these handlings will converge.
- Strong Consistency – maintains a global, real-time ordering and does not allow conflicts.

Depending on the consistency model used by a replication system, it can be classified as pessimistic or optimistic. Pessimistic replication guarantees that all replicas are identical to each other. New/updated data is only available after a synchronization between all nodes, in order to keep all replicas with the same value. Uses a strong consistency model. Optimistic replication is a strategy for replication in which replicas are allowed to diverge. There is no need to wait for all of the copies to be synchronized when updating data, which helps concurrency and parallelism.

2.1.4 Partial vs Full Replication

Instead of replicating the whole database, in partial replication each database only keeps some of the data. This concept arise to reduce some overheads introduced by replication:

- any single logical entity consumes systems resources, thus, the higher the replication degree, the more resources need to be used, proportionally;[8]
- coordination is needed among the replicas to maintain their consistency, if the status of a physical replica changes, all other replicas of the same logical must be made aware of this change in order to maintain the coherence of the involved entity.[8]

Some work have been done to adapt a full replication protocol to a partial replication protocol, while providing the same performance [11]. This protocol guarantees causal consistency and limits the amount of metadata sent during the communication between DCs (datacenters) by sacrificing the time needed until an applied update becomes observable. After applying an update, a DC has to wait until the sending DC informs it that the update dependencies have been applied and the data is safe to be observable, i.e, will not violate causal consistency.

2.2 Goals of Replication

Ideally, a perfect replication system would be able to provide every property without concern of trade-offs. These properties are:

High-performances – replicating an entity at the site where it is mostly refereed, can grant better performance and minimize access time. In addition, replication can decrease the communication load imposed on the network;

Greater availability – if different physical copies of an entity are distributed in the system, a replicated resource is more available than a single-copy one.[8] This property is a guarantee that every request receives a response about whether it succeeded or failed. To provide this property, systems replicate data and use geo-replication, to guarantee that systems are fault tolerant, i.e, systems can answer clients requests, even though some servers/datacenters may fail. Replication improves response time and prevents bottlenecks, which would decrease availability. Geo-replication is a good mechanism to improve availability, especially in case of catastrophes that may cause an entire datacenter to become offline;

Fault tolerance – a failure that occurs to one physical replica of an entity does not compromise the access to the logical entity since other replicas exist in the system;[8]

Strong consistency - all nodes see the same data at the same time, the system maintains a global, real-time ordering between operations;[8]

Scalability – Is the ability of system to grow, the capacity to adapt to increasing load and storage demands. If a system can easily improve its performance by adding new resources (in a proportional way), is said to be a scalable system.

Unfortunately, the CAP theorem [6, 13] proves it impossible to create a system that achieves Consistency, Availability and Partition tolerance. Instead, modern web services have chosen availability and partition tolerance at the cost of strong consistency. This choice enables these system to provide low latency for client operations and high scalability. Further, many of the earlier high-scale Internet services, typically focusing on web search, saw little reason for stronger consistency[19]. Systems with these properties are called ALPS systems (Availability, low Latency, Partition tolerance, high Scalability).

2.3 TLA+

TLA+ [16] is a formal specification language based on basic set theory and predicate logic, especially well suited for writing high-level specifications of concurrent and distributed systems [34]. In order to guarantee that a system correctly implements the desired correctness properties, TLA+ is used to specify all its possible execution traces. To do this, a TLA+ specification makes use of variables, and constants, the inputs of the model.

$$Spec \triangleq Init \wedge \Box[Next]_{vars}$$

Spec defines a system specification. It starts by implying that all execution traces start with a state (state is an assignment of values to variables) that satisfies the initial condition *Init*. After that, every system transition, i.e, every following states, is defined by the formula *Next*, which changes *vars*. The values assigned to variables on each state come either from inputs (constants), or libraries like *Naturals*, *Integers*, etc. *Spec* can generate multiple execution traces due to the number of possible initial states satisfied by *Init*, and all combinations of the possible system actions, defined in the *Next State Action* with multiple disjunction clauses (*OR* clauses).

TLC is a model checker for specifications written in TLA+. It finds all possible system behaviors (a behavior is a sequence of states), i.e, exhaustively checks all possible execution traces, and verifies if any of them violates the invariant properties such as safety and liveness. Safety properties can be described as what the system is allowed to do, while liveness properties can be described as what the system must eventually do [23]. Therefore, the TLC model-checker provides a verification of the system specification and its properties [26]. The procedure used by the model checker to compute all possible behaviors uses a directed graph, whose nodes are states, and has 4 main steps:

1. Computation of all initial states, by computing all possible assignments of values to variables that satisfy *Init*;

2. For every state found in step 1, compute all possible next states by substituting the values assigned to variables, with the operations defined in *Next State Action*;

3. For every state found in step 2, if it is not already in the graph, it is added, and an edge is drawn from the state that generated it, to it.

4. Steps 2 and 3 are repeated until no new states or edges can be added to the graph.

When this process terminates, the nodes of the graph correspond to all the reachable states of the specification. The process either ends with a state that is supposed to happen and marks the end of the execution, or with a deadlock, a state from which there is no next state satisfying the *Next State Action*, but corresponds to a situation not supposed to happen.

The choice of TLA+ to specify the developed protocol was due to its increasing use by projects in the industry. Companies have been justifying its use due to the ability it gives to handle extremely rare combinations of events which are hard to imagine, find bugs, verify complex designs, make innovative performance optimizations, but mainly the fact that it allows developers to precisely test and verify the safety of the developed changes, and therefore avoid serious bugs from reaching production. An engineer in Amazon Web Services mentioned "had he known about TLA + before starting work on Dynamo DB, he would have used it from the start, and this would have avoided a significant amount of time spent manually checking his informal proofs" [23]. Finally, several projects in the industry that used TLA+ can be named, the Paxos consensus algorithm [17], the Farsite distributed file system [5], the Pastry distributed key-value store [21], the fault-tolerant real-time communication protocol Doris [26], the partial replication protocol Cimbiosys [25], in complex cache-coherency protocols [3, 18], and more recently at Amazon to specify concurrent and distributed systems [22, 23].

Chapter 3

Related Work

3.1 Introduction

This chapter describes some relevant articles regarding data replication. These articles explain systems implementing different properties, like different consistency levels, different replication levels (partial and full replication), as well as systems which replicate data to clients.

We also describe centralized models and peer-to-peer models, where each node can synchronize with any other node, and any update can be applied at any accessible node.

We present examples of different topologies to organize nodes like a ring and a tree hierarchy.A tree hierarchy has also been used to balance the load of data requests within a system [15]. The system proposed in [15] maintains a main storage site in the root of the hierarchy, and client nodes as leafs of the hierarchy. Those nodes are connected by intermediate nodes which have a dynamic replica set, and are used to balance the load of data requests within the system. The system manages the intermediate nodes by deciding when to create a replica and where to place it, in order to replicate data that receive high number of requests and improve performance.

3.2 Convergent and Commutative Replicated Data Types

Convergent and Commutative Replicated Data Types (CRDTs) [31] is an optimistic replication method (uses eventual consistency) used with full database replicas. With CRDTs a replica may execute an operation without synchronizing a priori with other replicas. The local operation is sent asynchronously to other replicas, which then execute the operation remotely possibly in a different order. Replicas of any CRDT converge to a common state that is equivalent to some correct sequential execution. Updates to replicas

are unaffected by network latency, faults, or disconnection. CRDTs are based on the convergence property which states that two replicas eventually converge if every update eventually reaches the causal history on every replicas. The communication between nodes can be made using state-based or operation-based mechanisms.

The state-based mechanism updates the state at the source, then propagates the modified state between replicas. Replicas merge their state with the received state. Operation-based mechanism propagate operations and require verification of preconditions before applying updates.

A state-based CRDT is a Convergent Replicated Data Type (CvRDT). CvRDT uses the definition of join semilattice (a partial order that has a join with a least upper bound) to prove the convergence, by saying that a state-based object whose state takes its values in a semilattice, converges towards the least upper bound (LUB) of the initial and updated values. This way, CvRDT can be summarized by two important properties:

- when a replica receives and merge a state, it is guaranteed that all the state values will be equal or greater than the previous values. This means that merges are idempotent and commutative;

- eventual consistency is guaranteed if replicas exchange (and merge) states infinitely. Due the previous property, states can also be exchanged in an indirect channel (via successive merges), which allow the communication channels of a CvRDT to have very weak properties, messages can be lost and received out of order.

An operation-based CRDT is a Commutative Replicated Data Type (CmRDT). The key idea is that if all operations commute, operation can be executed in any (causal) order leading to the same state. In this case, unlike CvRDT, the communication channel must guarantee that all updates that are ordered in delivery order (not concurrent) are delivered in the same order at every replica (reliable broadcast). Concurrent operations can commute and still lead to an equivalent state.

The main problem with CRDTs is the fact that can become heavy data structures, which leads to a waste of resources like memory and bandwidth, specially in CvRDTs. CvRDTs may be inefficient because of the need to send the entire state, which imposes a large communication overhead as the state size becomes larger. To improve this, one solution was developed that, instead of shipping the entire state, only ships deltas [1]. Some work related to partial replication of CRDTs was also initiated in a master thesis [7].

3.3 Chain Replication

Chain replication [27] is a method that organizes a set of full replicas in order to offer high availability, high throughput, and strong consistency guarantees.
With chain replication, the primary's role in sequencing requests is shared by two replicas-/nodes, the head node of the chain receives update operations, and the tail node receives query operations. The implementation of these requests distinguishes three actions, a *query processing* request that is directed and processed atomically at the tail of the chain, *update processing* request directed to the head of the chain, that after processing, forwards the request to the next element of the chain until the request is processed by the tail, and finally, a *reply generation*. This latter action replies to the source of the received operation (query or update) and is executed by the tail, after processing the update. This actions guarantee strong consistency because query and updates requests are processed serially at the tail.

Chain replication handles server failures. To do that, it uses a central service called the *master*. The *master* has the responsibility to correct the chain by removing the server that failed, and must also inform clients if the tail and head of the chain change. The *master* distinguishes three cases:

- **Failure of the head:** His successor becomes the new head. Requests received by this server but not yet forwarded to a successor are lost.
- **Failure of the tail:** His predecessor becomes the new tail.
- **Failure of some other server:** In order to not lose updates, a four messages mechanism is used to remove the server that failed and connect its predecessor and successor.

Chain replication can be compared to primary/backup protocols. Chain replication share responsibility of requests in two replicas, which partitions the sequencing task. Only a single server (the tail) is involved in processing a query and that processing is never delayed by activity elsewhere in the chain. On the other hand, the primary backup approach, before responding to a query, must await acknowledgements from backups for prior updates. When comparing the disseminating costs of updates, it can be concluded that the primary/backup approach is better (lower latency) than chain replication. While chain replication disseminates updates serially, the primary/backup approach disseminates updates to backups in parallel. The delay to detect a server failure is identical for both protocols, however, the recovery costs (after a failure) can be compared. This comparison is done using the best case and worst case outage of each approach. In chain replication the worst case outage is the tail failure where message processing is unavailable for 2 message delivery delays, while in primary/backup, the worst case is the primary server failure, which leads to a 5 message delays. The best case outage for chain replication is a middle server failure, where some delay may be possible, although

no interruption in message processing happens, and in primary/backup protocol the best case outage is a backup failure. The backup failure has two situation, if no update requests are in progress, there is no processing delay, however if an update is in progress, one message delay happens while the *master* informs the primary server that it will not receive the acknowledge of the failure server.

The problem with chain replication is that the existing replicas are not leveraged to promote load balancing among concurrent read operations, all writes use the Head server and all reads use the Tail server. In addition, fault tolerance requires a central coordination service to know about all nodes of the system. Both those characteristics limit scalability.

3.4 ChainReaction

ChainReaction [2] is very similar to ChainReplication, however, it tries to distribute the load of concurrent read requests among all replicas, and allows concurrent requests to be executed in parallel. ChainReaction offers causal+ consistency (it's conflict resolution method is based on the last writer wins rule). The ability of ChainReaction to allow concurrent update can lead to situations where replicas temporarily diverge, however it guarantees that applications never observe a state previous to one they observed before. Servers are organized in a ring, and each item can have different chains (different head and tail).

Clients execute operations (*get* and *put*) through an API provided by a client library. This library manages client metadata. This metadata is used to offer causality of the state observed by clients. It keeps an entry (*key*, *version*, *chainIndex*) for each item accessed by the client, that is not DC-Write-Stable(d), i.e, writes not yet applied to all nodes in the chain located in datacenter d that is responsible for the object targeted by the write operation.
Items can be not DC-Write-Stable(d) because while in chain replication writes are only returned by the tail, in ChainReaction writes are returned as soon as they are processed by the first k replicas (k defines the fault-tolerance of the chain). *Put* operations are only executed when all the versions that the new item causally depends, have become DC-Write-Stable(d) (these items are all the requests in the client metadata). As soon as the writes of all the objects on which the new update depends have become DC-Write-Stable(d), the proxy (each datacenter uses a proxy) uses consistent hashing to discover which server is the head of the chain of the item being updated, and forwards the *put* request to that node. After replicating the write on k elements of the chain (eager propagation phase), a result is returned and placed in the metadata. This result include the most recent version of the object, and a *chainIndex* representing the k_{th} node. After that

node, the replication is performed lazily until it reaches the tail of the chain.

When the client sends a *get* request, the *chainIndex* of the global metadata is used to decide to which data server the *get* operation is forwarded to. The proxy selects a server at random with an index from 0 (the head of the chain) to *chainIndex*. This strategy allows a distribution of read requests among the servers whose state is causally consistent.

To support geo-replication, ChainReactions uses multiple datacenters. To do this, some modifications were made. Instead of keeping a *chainIndex* in the Metadata, a *chainIndexVector* is maintained.
In *put* operations, datacenters must communicate in two situations. The first is after the update being processed by the head, starting the transfer of the update to the remote datacenters. The second is after the update being processed by the tail, where an acknowledgment is sent to the tails of the same item on the other datacenters informing that the item is DC-Write-Stable(d) on that datacenter. When an update is DC-Write-Stable(d) on every datacenter, it is called Global-Write-Stable.
As explained before, when using a single datacenter, after writing the update in k nodes, the information will be written in the metadata. The only difference when using multiple datacenters will be the *chainIndexVector*, instead of only writing in the metadata the k value (identifier of the last server which replicated the write operation), the whole *chainIndexVector* must be written. This *chainIndexVector* will have the value 0 for all the datacenters but the one where the put operations was executed, that will have the value k.

When a datacenter receives a *get* requests (assuming the existence of more than one datacenter), it may happen that the datacenter, i.e, head of the local chain, does not have the required version. The *get* operation can either be redirected to another datacenter or blocked until the update becomes locally available.

In conclusion, ChainReaction is a very interesting system due it's efficient use of the available replicas while providing guarantees to clients that the read values are always updated, without using a strong consistency model. Nevertheless, the fact of ChainReaction uses full replicas, in addition of the problems inherent to full replication, can lead to a great waste of resources because each datacenter needs to keep several full replicas.

3.5 Pastry

Pasty [20] is a decentralized protocol that supports partial replication by implementing a distributed hash table. It organizes the nodes of the system in a ring (if the system has more than 3 nodes, each node will have two neighbors), and it automatically adapts to the arrival and departure of nodes.

When an object is created, it is assigned an object id (*objId*) and is replicated across multiple nodes, however, only one node is responsible for each object. Lookup requests (object consultations) must be routed to the node responsible for the object. Object responsibility is divided equally according to the distance between two neighbor nodes, and the protocol has a correctness property guaranteeing that there is always at most one node responsible for a given object. Since nodes can fail, in order to guarantee that property, each node must store two leaf sets of size l containing its closest neighbors to either side (l nodes to the left and l to the right). Those sets will be used to correct the ring after a fail is detected, and l corresponds to the fault tolerance of the protocol, i.e, $l-1$ sequential nodes can simultaneously fail and nodes can still correct the ring.

The Pastry protocol has two main operations, the join operation to add a new node in the system, and the lookup operation to consult the value of an object.

Since every node can receive a lookup request for any object, each node implements a routing table to forward the request to the node responsible for the object. In larger systems, routing tables might not contain information about all nodes of the system, therefore, nodes use these tables to keep forwarding the lookup request message to a node closer with the one responsible for the object, until the message finally reaches its destination.

When a node joins the system, a random unique *nodeId* is assigned to it. That *nodeId* will define which objects the node replicates and is responsible to. A node is responsible for objects with *objId* numerically close to its *nodeId*. The process of adding a new node in the ring starts with the new node communicating with its closest node, asking it to enter the ring (sends 2 messages and wait for both replies, first step takes 4 messages). The new node will receive the leaf sets of its closest node to construct its own leaf sets. After that the new node will communicate with all nodes in its leaf sets (as mentioned, it contains two leaf sets, each one with l nodes) in order to confirm their presence on the ring (sends a messages to $l+l$ nodes and waits for replies, second step takes $4l$ messages). After finalizing this process, the new node will send a message to its closest neighbors to confirm the validation of its information, and after receiving a response, it will become in a "ready" state and will start executing normally (more 4 messages). In sum, the addition of a node requires $4+4l+4$ messages.

Pasty is an interesting system due to its partial replication and its mechanism of fault tolerance which does not require central coordination policy. In addition, Pastry was also specified with TLA+. Nevertheless, Pasty has some limitations, it is only focused on maintaining data, nodes cannot select which keys to replicate, the keys that each node replicates is based on its identifier, and it requires a $4+4l+4$ messages to add a new node in the system, which can compromise scalability.

3.6 Adaptive Replication

The implemented system [8] presented by this paper has as main goals throughput and fault tolerance. The system implements an active replication scheme, following an adaptive file replication policy (new replicas are created and/or deleted depending on the pattern of access).
The server of the system has three layers, the upper layer (interface layer) to receive and answer to clients requests, an intermediate layer (coordination layer) for coordination with other servers and accessing to local data, and the lower layer (replication policy layer) which is used to decide whether to create or delete a replica of an item. This decision is influenced by the number of replicas of a file in the system and the type of accesses, as will be explained later.

Each server does not keep track of which servers maintain replicas of a given file, so when a replica is not present, the request is broadcast to all other servers. However, any server knows how many replicas of the file are present in the system, so when a process has a write operation, it should contact all other servers to find which hold one replica of the same file. When at least a half plus one of the servers confirm that no other process is currently writing on the same file, the client is allowed to update the file and propagate the other servers.

The lack of global vision of this system oblige each node to maintain local independent replication policy (decisions are based on local information only), and therefore no capability to influence other nodes. The implemented replication policy has been designed with the main goal of locality. This approach has some advantages like preventing the need of coordination protocols between the replication policies of different nodes, and the robustness to failures, a failure in a site does not interfere with the activities of other policy modules. However, the lack of a global vision, obliges the propagation of all creation and deletions operation to all servers, in order to keep the current number of replicas (of that file) updated.

A formula is used to decide whenever delete or replicate a file. If few reads and many writes are issued, the local file replica should be deleted, if many reads are issued, a local file replica should be created.

This system has other problems related to the patterns of access. To deal with those problems, the parameters used in the formula referred in the previous paragraph must be adjusted carefully. If the patterns of access are highly dynamic, the parameters must be adjusted to prevent unnecessary actions (deletes and replication of files). On the other hand, if the patterns of access change slowly, the parameters must also be adjusted to increase the responsiveness of the replication policy. The problem arises because these

two cases are conflicting. Another problem is related to the communication, as the lack of global awareness increases the number of messages sent, which could be aggravated in global systems (lack of scalability).

3.7 Cimbiosys

Cimbiosys [25, 28] is a partial replication system where each node contains a filter specifying its interest set. Nodes are organized in a filter hierarchy, a tree in which one full replica is chosen as the root of the hierarchy. Cimbiosys was designed for clients, i.e, nodes of the system are clients, and uses a peer-to-peer architecture. Nodes synchronize with each other from time to time, and do not attempt to maintain any ordering between updates (eventual consistency). However, causality can be achieved if nodes only synchronize with its parent and children.
Each node keeps a set of items, and a dynamic filter so that it eventually only stores all the items that match the filter. Each node also keeps a *knowledge*, a set of version-ids that contains identifiers for any versions (items updates generate version) that match the replica's filter and are stored by the node, or are known to not match the node's filter.

An item is an XML object with information of the item to be filtered (description, rating, etc), and an optional associated file (like JPEG data to store photos). Updates produce new versions of items that are later sent to other replicas. Each replica has an *updateCount*, a *replicaID*, a *filter*, and *knowledge*. Each item has an *item identifier* which labels all versions of the item. Each version of an item has a unique *version identifier*, the *item identifier*, a *made with knowledge* which indicates which versions of the same item this version supersedes, and a *version content*. This *versionID* is created by nodes, each time an operation is performed, assigning the value of the *replicaID* coupled with the *updateCount* to the *version identifier*. Conflicts arise when two different versions of the same item were created and neither one supersedes the other.

Knowledge is a group of version vectors associated with a set of item ids with the format S:V. Each fragment S:V indicates that the replica knows all versions of items in the set S whose version-ids are included in the version vector V.

The Cimbiosys synchronization protocol has several steps:

- *Target* replica initiates synchronization with replica *Source* by sending a message that includes the target's knowledge and its filter.
- *Source* replica checks if its item store for any items whose version-ids are not known to the target replica and whose XML contents match the target's filter.
- The XML contents, file contents, and metadata for each of these items are returned to the *target*.

- If possible the *source* replica also informs the *target* replica of items that no longer match its filter and must be removed from the device (move-out notifications).
- The source replica responds with a message including one or more knowledge fragments that are added to the target's knowledge.

There are two conditions under which the *source* returns move-out notifications:

- When the *source* replica stores a newer version of an item than the *target* replica, and the content of the item do not match the *target's* filter.
- This condition only happens when the *target's* filter corresponds to a subset of the *source* replica's filter. If the *source*'s knowledge for this item is greater than the *target's* knowledge, the *target* replica stores the item, and the *source* replica does not.

To assure the permanence of data, Cimbiosys guarantees that a device only permanently remove its local replica after synchronizing with some other device.

If the replica does not have any synchronization partner whose filter matches the item, the replica can still decide to send all items in its push-out store, to be able to remove them. This partner is always forced to accept these items even if they do not match its filter.

When a device changes its filter, there are three possible situations that might occur:

- The new filter is more restrictive than the previous filter. In this case, items that no longer match the filter are moved to the replica's push-out store.
- The new filter is less restrictive than the previous filter. In this case the knowledge must be reduced to only store version of items already stored. Knowledge about items that are known to not match the replica's filter must be removed, because those items can now match the new filter.
- The new filter has no relation with the previous filter. In this case the actions of the previous two situations must be applied.

The referred mechanisms to guarantee that no data is lost are supported with the concept of authoritative. The idea behind this concept is to hold every unsuperseded version somewhere, so that updates do not disappear. To allow replicas to discard unwanted data, Cimbiosys constantly transfers the authority of the created versions. The idea is to transfers the set of a replica's authority to its parent in the filter hierarchy, with everything eventually ending up at the root (the full replica).

In conclusion, Cimbiosys was described due to its partial replication mechanisms, and the hierarchical organization between nodes. The problem with the hierarchy is that still requires the existence of a full replica, that might not be needed or wanted. In addition, Cimbiosys was also specified with TLA+.

3.8 Perspective

Perspective [29] is a storage system designed for small environments and replicates data to clients, instead of servers. It is a decentralized system, uses a peer-to-peer architecture to allow communication between nodes of the system, supports partial replication, topology independence, and guarantees eventually convergence of data.

The partial replication is a selective replication, i.e, each node can specify which data to replicate. To do this, each device holds a view. A *view* is a description of the data stored, and is expressed like a search query. Perspective ensures that any file that matches a view will eventually be stored on the node that uses that view, and allows nodes to add and remove *views*. The *view* mechanism used by Perspective to specify which data to store is very similar with the *filter* used in Cimbiosys, the main difference between both system is that Perspective does not require a node to keep the full datastore.

As mentioned, the system supports topology independence. In order to guarantee that updates are propagated to all nodes interested in the updated file, each node holds a list of views regarding the other nodes of the system. When a node executes an update, it sends a message to each device with a view that contains the updated file, and after receiving that message, if the receiver node did not change its view, it pulls a copy of the updated file (3 messages). Since nodes can concurrently execute updates, conflicts can be generated. A version vector is used to detect conflicts, and the file's modification time, or the user input is used to handle those conflicts.

In order to guarantee that no data is lost, Perspective implements some conditions to guarantee that nodes do not drop files that might need to be propagated. The first functionality is marking an updated file as "modified", until it is pulled by another node. The second functionality is marking all files as "modified" when a view is removed from the node. These conditions assure that a node will not drop a file until it has confirmed that another replica of that file exists somewhere in the system.

In conclusion, Perspective is an interesting system due to its ability to provide partial replication without keeping a master node with the full datastore, and its topology independence. However, Perspective requires nodes to keep metadata regarding all nodes of the system, which affects its scalability, and its capacity to be used in larger systems.

3.9 PRACTI Replication

PRACTI [4] is a system that provides three properties, partial replication, arbitrary consistency, and topology independence. Arbitrary consistency means that the system is flexible, particularly in the ability to enforce consistency guarantees that influence the order that updates become observable to readers. Topology independence means that the protocol works with different communication topologies.

This system has two important principles:

- The separation of control path from the data path by separating invalidation messages (messages that identify what has changed), from body messages (messages with the actual changes to the contents of files).
- imprecise invalidations, which allow a single invalidation to summarize a set of precise invalidations, and are used to provide partial replication. These invalidations allow nodes to omit details from messages sent to another node, while still allowing receivers to enforce causal consistency.

Precise invalidation messages contain two fields: $objId$, which identifies the modified object, and *accept*, which is the accept stamp assigned by the writer when the write occurs.

A node's local state contains a version vector, $currentVV$, an ordered log of updates (sorted by accept stamp) and a *per-object store* representing the current state of each object for reads. Besides storing the accept stamp and the value of an object, the *per-object store* also keeps a *valid* field for each object.

A node receives a stream of updates (invalidations) $\{startVV, w1, w2, ...\}$, where $starVV$ is the version vector of the node (sending the updates), and $w1, w2$ the different updates. The receiving node rejects the stream if $startVV_x > currentVV_x$ for any node x (this verification detects any missing updates), otherwise, it processes each w_i by inserting the write into its sorted log and updating the store. The *valid* field is assigned the value *INVALID* until the correspondent *body* (of the processed invalidation) arrives. Although invalidations must respect a causal consistency, the destribution of bodies can be done in arbitrary order. Bodies are not applied until the corresponding invalidation message has been processed. The system ensures a causally consistent view of data by having a local read request block until the requested object's *valid* field is *VALID*. To ensure liveness, a mechanism can be implemented to arrange some node to send the body of an *INVALID* object requested for read.

In order to guarantee reliability, i.e, the loss of a node or a replacement decision does not make some data unavailable, the PRACTI system enforces a policy decision about the minimum acceptable level of replication of an object. To this end, the system uses *bound* invalidations (invalidations messages with a *body*).
Until the system has a confirmation that k replicas of a file as been replicated, nodes propagate *bound* invalidations. After that, nodes start to propagate *unbound* invalidations, i.e, the system starts to work like explained before, invalidations and bodies separed.

An imprecise invalidation contains three fields, *start* and *end* (accept stamps) and *target*, which is a set with identifiers of the objects affected by the invalidation. $Start$'s value is

the earliest accept stamp and *end*'s value is the latest accept stamp of the modifications over the objects in *target*. The difference between the representation of a precise and an imprecise invalidation is that in a precise invalidation, only a single write is represented so $start = end$, and *target* only contains a single object.

The decision about which node receives *precise* or *imprecise* invalidations is made during the initial connection between nodes, where the receiving node specifies which invalidations wants to receive (the communication is not just a change of invalidation).

Nodes group system data into interest sets, and check whether an interest set is *precise*, meaning that the node's local state reflects all invalidations (all precise invalidations) overlapping the items in the interest set, or *imprecise*, meaning that the node's local state over the items in the interest set is not causally consistent due the use of imprecise invalidations. In order to guarantee that reads always observe a causally consistent view, reads are blocked until the interest set of the item being read becomes *precise*.

Although the system claims to provide partial replication, in the end, every node stores some information about every object and fails to offer one of the advantages of partial replication, the reduced coordination between replicas (every node must see all invalidations). This allows nodes to compose precise invalidations into imprecise ones, and allows nodes to recover precision for an interest set that has become imprecise, without great complexity.

3.10 PNUTS

PNUTS [10] is a massive-scale, hosted database system to support Yahoo!'s web applications focused on data serving for web applications, rather than complex queries (like joins, group-by, etc). As a web applications, PNUTS requirements are scalability, good response time for geographically dispersed users, high availability, and fault tolerance. This type of systems (web applications) usually don't require a strong consistency model, however an eventual consistency model is often too weak and hence inadequate.

Communication in PNUTS is made by propagating asynchronous operations over a topic-based publish–subscribe system called Yahoo! Message Broker (YMB). In this method replicas do not need to know the location of other replicas.

The consistency level used by PNUTS was not explained previously in this document. This level, called per-replica timeline consistency is stronger than eventual consistency but weaker than causal consistency. This level guarantees that updates of a record are applied in the same order on every server. This is done with a property called *Record-level Mastering*. This property chooses one of the replicas of the record as the master, and all

updates to that record must be forwarded to the master. To be able to forward the update to the master replica, each record maintains a hidden metadata field with the identity of the current master. The master replica of a record can be dynamically changed, depending on the number of local updates on each replica of the record. Updates are committed and published by the master in a single message broker (YMB), to be asynchronously propagated to non-master replicas. Updates over a record are all delivered to replicas in commit order, however, as different records might use different messages brokers (YMB), causality between updates on different records can not be achieved.

By using this per-record timeline consistency model, PNUTS supports different API calls with different levels of consistency guarantees:

- **Read-any:** Returns a possibly outdated version of the record, however the returned record is always a valid one from the record's history.
- **Read-critical[required version]:** Returns a version of the record, newer or the same as the required version
- **Read-latest:** Returns the latest copy of the record that reflects all writes that have succeeded.
- **Write:** Updates the value of a record
- **Test-and-set-write[required version]:** This call performs the requested write to the record if and only if the present version of the record is the same as required version.

The system architecture of PNUTS is divided into regions, where each region contains a full replica of the database. PNUTS does not have a database log, the replication and reliability is guaranteed by the delivery pub/sub mechanism. A region is composed by storage units (tables), tablet controller and routers. Storage units respond to requests, while routers and tablet controllers determine which table of the storage unit is responsible for the required record.

As said before, PNUTS was chosen as an example of a web application system. Besides not having any direct contribution to the proposed work, it introduces a different consistency model, and the implementation of an ALPS system, a system with different requirements of the systems explained before (scalability, good response time for geographically dispersed users, high availability, and fault tolerance).

3.11 Bayou

Bayou [24] is a causally consistent replicated storage systems with an "update anywhere" model for data modifications, with full replication, and support for arbitrary

communication topologies.

In Bayou, each server contains an ordered log of writes, and a database (result of the execution of the writes). This log contains contains all writes executed locally and received by other server. Bayou's writes consists in a set of updates, a dependency check and a merge procedure, where the last two are used as a conflict resolution method. The writes are only used when synchronizing with other servers (if the server is missing writes, they can be all sent).

Bayou uses a version vector with the number of updates received from other servers. This way, Bayou guarantees that new updates of a server *R* are only applied after the previous updates of that server *R* been applied.

The algorithm used for communication between servers is very simple:

- The *receiver* server sends its version vector to the *sending* server.
- The *sending* server examines its write log
- The *sending* server sends the missing writes to the *receiver* server.

This algorithm is based on an assumption that servers do not discard writes from their write-logs. However, Bayou allows servers to, independently, shrink their write-logs. This can cause servers that are too far "out of synch", to receive the full database state from a *sending* server.

In order to allow a server to remove a write from its log, that write must be a *stable write* (or *committed write*). Bayou uses a *primary-commit* protocol to stabilize writes. This protocol uses a database server as the primary replica. Writes committed by this server are assigned with a increasing commit sequence number (*CSN*). That new value represents the global order of the writes. Writes not yet committed by the primary server are called uncommitted (or tentative). A server (other than the primary server) can only receive a committed write if it already stores every committed write before that. To aid the propagation of *commited* writes, each server stores a variable with the highest committed sequence number (highest *CSN*), so that the sender can compare the two server's highest commit sequence numbers, and send the missing committed writes.

The ability to remove writes from the log can cause situations where a server's write-log may not hold enough writes to allow a successful synchronization between nodes, e.g. Server *A* only stores committed writes with *CSN* higher than 100 (i.e, the first 100 writes were deleted from the log). If the server *B*, that only has 40 committed writes, tries to synchronize with the server *A*, *A* will not be able to send the missing updates. To prevent situations like the previously explained, each server maintains another version vector

$0SN$, that represents the number of committed writes deleted (omitted writes) from the log. By keeping this version vector $0SN$ and the highest CSN, a server can easily detect whether it is missing writes needed to communicate with another server. If the $S.0SN$ is higher than the $R.CSN$ (S represents the sender, R represents the receiver), there are committed writes that the sending server deleted (omitted) from its log, that the receiver has not yet received. Under this circumstance, if the servers still decide to synchronize, S must transfer the full database, instead of the missing writes.

In conclusion, Bayou was described due to ability to provide a causal consistency with an "update anywhere" policy, without needing support of any proxy (like ChainReaction [2]) to coordinate client's writes.

Chapter 4

Protocol

4.1 Overview

The presented solution consists in a specification of the protocol ParTree, a partial replication protocol that guarantees causal+ consistency between related operations, convergence of the data, and that no data is lost. The developed solution organizes nodes in a tree hierarchy. The root of the hierarchy is a full replica, where each child has a subset of the data of its parent, and siblings have disjoint data partitions. Since the system is focused on clients, the hierarchy is dynamic, i.e, instead of a fixed number of replicas, the protocol allows insertion and removal of nodes (all nodes but the root can be removed). The datastore of every node will consist in a key-value datastore, i.e, data will be represented as a collection of key-value pairs, such that each key appears at most once in the collection. This datastore can only grow, and it supports two different operations, $put(key, value)$, to create or update the value of a key, and $get(key)$ to read a value. Nodes cannot drop or remove keys from their datastores. Since all clients can execute operations (online or offline), the protocol uses an "update anywhere" model for data modifications. As explained, causal+ consistency means that the protocol uses a handler function to resolve conflicts. On this protocol, that handler function is based on the hierarchy level of the node that sent the update, so nodes higher in the hierarchy will win the conflict. ParTree is also able to guarantee fault tolerance of nodes. When a node detects a failure of one of its neighbors, it has the ability to automatically rectify the hierarchy, without a central coordination entity. ParTree assumes that:

1. The root of the hierarchy does not fail nor disconnects;

2. The communication channel of nodes uses a FIFO buffer, ensuring that the first message sent by the sender, will be the first message received by the receiver;

3. Messages are not lost, i.e, if a node sends a message, the destination will receive it.

Figure 4.1 shows an example of an hierarchy with four nodes, X, Y, Z, W, where the root of the hierarchy contains the full datastore a, b, c, d. This hierarchy will be used in the examples that will be later explained.

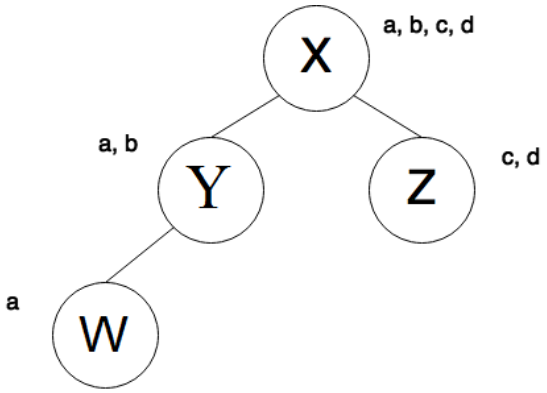

Figure 4.1: Example of a node hierarchy.

4.2 Example of use

ParTree is aiming to improve the replication and communication of data in a working environment (like a company). This solution would be specially suitable for companies with geographically distributed offices, since presumably, different regions are interested in different information (disjoint datasets for each region). As mentioned earlier, this protocol uses a hierarchy. The root of the hierarchy would be the central server, while the rest of the nodes would be clients, organized based on the set of the datastore that are interested in. The replication and communication between clients would improve the availability of the data, and therefore, allow clients to work while offline. The fact that only some clients communicate with the server, allows a diminishing dependency on the server. The clients (of each region) could be connected in a LAN environment, allowing them to keep working, as if nothing had happened, during disconnection periods like a temporary disconnection of the server or an internet failure. After the recovery of the connection, the client higher in the hierarchy would propagate all the updates to the server. Although it was stressed out that only a central server exists, each region could have its own server (second level of the hierarchy would also be servers), connected to clients in a lan environment, improving even more the availability of the data and guaranteeing that no update is lost, since servers are more reliable than clients and do not leave willingly the hierarchy (after the recovery of the connection, the local server would always be available to connect with the central server).

4.3 Node Information

All decisions made by nodes are based on information that they store and receive (messages), and this information is only altered by executing operations. Each node stores *a*) its datastore; *b*) which keys its children are interested in; *c*) a log of executed operations; and *d*) a version vector. The datastore of the node stores the value and version of the different keys, and the information about its children keys is needed to allow nodes to only propagate updates to the right destinations (only the identifier of those keys is needed). The log is needed to re-propagate operations after hierarchy changes, which will be explained later. A version vector is a mechanism used in distributed systems to establish a partial order among the different nodes of the system. Each node stores its own version vector, and by comparing it to another, it can determine if one update preceded another, or if they happened concurrently (a version vector is received with an update). A node can store an entry on its own version vector for a number of other nodes in the system, and each entry will contain a counter representing the number of operations that the current node is aware of.

Since counters increase when an update is executed, a version vector can be used to determine if one update preceded another, or if two updates happened concurrently. The version vector used by ParTree is slightly different from the majority of the version vectors that other systems use [4, 24], instead of increasing the counter for global operations, i.e, operations executed locally and applied, only increases the counter for operations executed locally. In ParTree, every time a node updates the value of a key, or applies a change in the hierarchy (due a removed or inserted node), the node increases the value of its counter, and adds an entry to the log. The size of the log will be equal to the number of the counter of its own version vector entry. This version vector also keeps information to identify the parent and children of each node. In ParTree there are three main reasons to keep a version vector on each node:

1. it allows nodes to detect and resolve conflicting updates;

2. it gives nodes the ability to automatically rectify the hierarchy, without a central coordination entity, after a node failure;

3. it allows nodes to known which messages the new communication partners are missing, and need to be re-propagated. When the hierarchy changes, nodes related with the one that caused the hierarchy change might need to re-propagate some messages.

These advantages will be better explained in the next subsection, where operations are explained in detail.

To clarify how this version vector works lets consider the example in Figure 4.2 (this example uses the hierarchy of Figure 4.1). In this example all nodes of the hierarchy have not executed any operations, and nodes W, Y, and X contain the key a (the version vectors

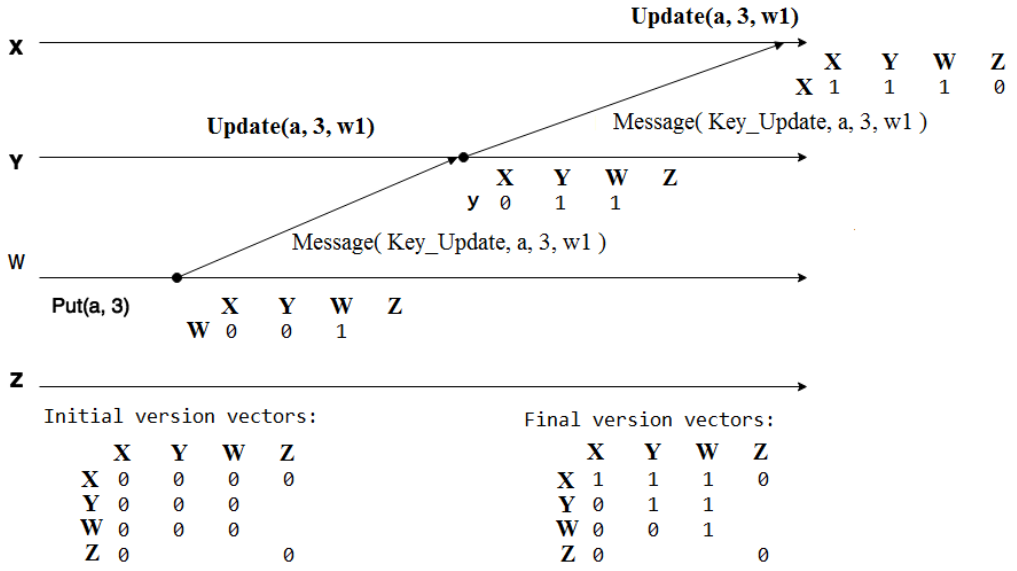

Figure 4.2: Key update propagation example.

of the example only show the counter of executed operations, the parent and children are omitted). The example starts when the leaf node (node W) updates the key a. After the update has been fully propagated, i.e, after the root applied the update, the version vector of the root would have the counter of executed operations = 1 on every entry referring a node that applied the update, i.e, all nodes in the path between the root X and the node that originally created the update, W. If the counter was only keeping track of operations executed locally, the version vectors of nodes W, Y, and X would have the value 1 on w's entry, and 0 on other entries.

As explained, this protocol is focused on clients. This means that the hierarchy can become very large, and since clients have less resources that servers, the storage capacity is an issue that must be considered. By keeping a counter of global operations, each node can only keep an entry for every node with n, or less, levels up and down in the hierarchy, where $n-1$ would be the number of levels that could fail simultaneously, and still allowing the node to keep working properly, and correct the hierarchy itself. The example of Figure 4.3 shows the version vectors of each node, when the kept entries are restricted by the number of hierarchy levels. Here $n = 2$, which means that the root (node X) will not keep an entry regarding the leaf (node Z). This example portraits a situation where the leaf (node Z) creates an update that must be propagated to all nodes. After receiving the update, but before propagating it to the root, node Y fails. After correcting the hierarchy, nodes X and W will be connected, and during the process of correcting the hierarchy, X will eventually send its version vector to W. When W receives X's version vector, if the version vectors were storing the local operations executed, it would compare the entries X, Y, and W from X's version vector to the correspondent entries of its own version vector. Since the values of all those entries from both version vectors were 0, W would not be

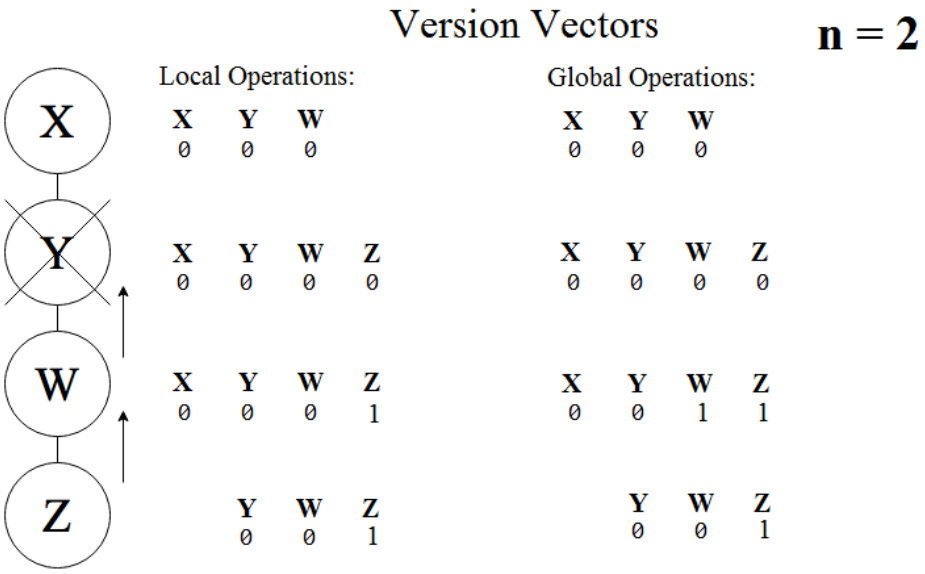

Figure 4.3: Example of version vectors restricted by the levels of the hierarchy.

able to find out that X did not receive the update. On the other hand, if nodes version vectors were storing the counter of global operations, after receiving X's version vector, W would just need to compare its entry on X's version vector (*receivedVV[W]*), with its entry on its own version vector (*VV[W]*). As shown in Figure 4.3, this comparison would compare 0 with 1, and W would realize that X was missing an update. The downside of this approach would be if n upper node levels simultaneously fail, the node would have to be placed again in the hierarchy, instead of connecting directly with the new parent, and independently rectify the hierarchy. Nodes not related, i.e, nodes that do not have keys in common, do not need to keep entries in the version vector, because they will never need to connect with each other.

4.4 Operations

In order to execute and propagate updates, create keys, and allow hierarchy changes, nodes have the ability to execute different operations, either by themselves, or as result of a received message. Most of the executed operations will end up sending a message to another node. Every time a message is sent, the node also sends its version vector, so when the receiver processes the message, it updates its own version vector with the received one. When updating a version vector, a node does not need to update all entries, if the message came from a node higher in the hierarchy, only entries of nodes higher in the hierarchy are updated, if it came from a node lower in the hierarchy, only entries of nodes lower in the hierarchy are updated. For this reason, one way to reduce the information sent is to only send the version vector entries that will be processed by the receiving node. This section will explain those operations and introduce their pseudocode. Section 4.4.1 discusses operations for updating/creating keys, while Section 4.4.3 discusses operations

for changes in the hierarchy. After that, Section 4.4.4 explains the error handling mechanism. Operations are split in two subsections, one for operations updating/creating keys, and other for changes in the hierarchy. After that, a subsection explaining the mechanism to handle errors is explained.

4.4.1 Operations for creating and updating keys

As explained earlier, nodes can create or update keys. The process of creating/updating keys allows nodes to also update keys not contained in them, instead of only allowing updates on keys they contain. Due the properties of the hierarchy, forcing nodes to contain all keys of their children and siblings to contain disjoint datastores, a node cannot independently create a key. The creation of a key only happens in two situations, either by receiving a message with type *new_key*, or if the node is the root. Since the root contains the full datastore, it can independently create a key without violating the properties of the hierarchy. Nodes only know about the existence of keys they contain, so when a node executes an update on a key it does not contain, the key may, or may not, exist in the datastore. For that reason, this update can cause two different effects: *a*) the propagation of the update will eventually reach a node that contains the key, and it will apply the update; *b*) the update reaches the root, and the key will be created.

Both operations are executed with the function $Put(keyId, value)$ and its pseudocode can be seen in Algorithm 1. If the node contains the key, an update is done, if it does not contain the key and it is the root, it means that the key does not exist, and is created, otherwise an update to an unknown key, that may, or may not exist, is created.

Algorithm 1 Put

```
Require: keyId, value
1: datastore {node datastore. Contains information of its keys}
2: if datastore.hasKey(keyId) then
3:     Update(keyId, value)
4: else if isRoot() then
5:     CreateKey(keyId, value, NullVersion, {})
6: else
7:     UpdateUnknownKey(keyId, value, {})
```

The function $Update(keyId, value)$ (see Algorithm 2) creates a new version of the key, updates the value, adds the operation to its log, and then propagates the update with the new version and value to its parent and any child that is interested in the key (a node knows which keys each child is interested in). The maximum number of messages required to fully propagate an update is equal to the height of the hierarchy, for example, if the leaf with the lower hierarchy level executes an update, it must be propagated to the root.

When a node receives a message *key_update*, Algorithm 3 is executed. The first step of this function is to verify if the update should be applied, i.e, if the update does not raise a conflict with the existing version of the key, or if it does, but the received version wins

Algorithm 2 Update

Require: keyId, value
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: *nodeInfo* ← *vv*[*nodeId*]
6: *nodeInfo.executedOperations* ← *nodeInfo.executedOperations* + 1
7: *newVersion* ← *CreateVersion*(*nodeId*, *nodeInfo.executedOperations*)
8: *datastore*[*keyId*] ← *UpdateKey*(*keyId*, *value*, *newVersion*)
9: *op* ← *NewOp*("key_update", *keyId*, *newVersion*, *value*, *vv*)
10: *log* ← *Append*(*log*, *op*)
11: *SendMessage*(*nodeInfo.parent*, *op*)
12: *SendMessage*(*datastore*[*keyId*].*childInterested*, *op*)

the conflict (this process will be explained later). If the update is applied, it keeps being propagated in the same direction, until it reaches the root or a leaf. If the message came from a child, it is propagated to its parent, if it came from its parent, it is propagated to a child, if any is interested in the key.

Algorithm 3 ReceivedKeyUpdate

Require: keyId, value, version (from a message)
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: *nodeInfo* ← *vv*[*nodeId*]
6: **if** *IsToApplyUpdate*(*datastore*[*keyId*], *version*) **then**
7: *nodeInfo.executedOperations* ← *nodeInfo.executedOperations* + 1
8: *datastore*[*keyId*] ← *UpdateKey*(*keyId*, *value*, *version*)
9: *op* ← *NewOp*("key_update", *keyId*, *version*, *value*, *vv*)
10: *log* ← *Append*(*log*, *op*)
11: **if** Message came from parent **then**
12: *SendMessage*(*datastore*[*keyId*].*childInterested*, *op*)
13: **else**
14: *SendMessage*(*nodeInfo.parent*, *op*)

As explained earlier, when a node executes the function *UpdateUnknownKey* (Algorithm 4), it means that the node does not contain the key being updated, and it is not the root. Since the node must contain all keys of its children, it knows that none of its descendant nodes can contain the key. If the key exists, someone higher in the hierarchy has to have it, so a message *new_key_or_update* is propagated to its parent. If this update ends up in the root and the root does not have the key, all nodes in the path between the root, and the node that originally created the update, will create it, so a reference to those nodes must be propagated with the update.

The message *new_key_or_update* is always received from a child. When a node receives it, a function similar with the *Put* function is executed (Algorithm 5). The only difference is the set of nodes interested in the key trying to be updated. This set will

Algorithm 4 UpdateUnknownKey

Require: key, value, setNodesInterested
1: *nodeId*
2: *vv* {node's version vector}
3: *log* {node's log}
4: *nodeInfo* ← *vv*[*nodeId*]
5: *nodeInfo.executedOperations* ← *nodeInfo.executedOperations* + 1
6: *setNodesInterested* ← *setNodesInterested* ∪ {*nodeId*}
7: *op* ← *NewOp*("*new_key_or_update*", *keyId*, *value*, *setNodesInterested*, *vv*)
8: *log* ← *Append*(*log*, *op*)
9: *SendMessage*(*nodeInfo.parent*, *op*)

only be used if the root ends up creating the key. In sum, the propagation of a message *new_key_or_update* will eventually trigger one of two actions, either a key update (Algorithm 2), or a key will be created (Algorithm 6).

Algorithm 5 ReceivedUnknownKeyUpdate

Require: keyId, value, setNodesInterested (from a message)
1: *datastore* {node datastore. Contains information of its keys}
2: **if** *datastore.hasKey*(*keyId*) **then**
3: *Update*(*keyId*, *value*)
4: **else if** *isRoot*() **then**
5: *CreateKey*(*keyId*, *value*, *NullVersion*, *setNodesInterested*)
6: **else**
7: *UpdateUnknownKey*(*keyId*, *value*, *setNodesInterested*)

Finally, if the root receives a message *new_key_or_update*, or decides to execute the *Put* function on a key it does not contain, the function *CreateKey* will be executed (Algorithm 6). When the root executes this function, it will also create the first version of the key. After creating it, and adding the operation to its log, a message *new_key* will be propagated to any interested child (only one child can be in *setNodesInterested*). Nodes interested are the ones between the root and the node that originally created the update on the unknown key, and therefore the ones that propagated the message *new_key_or_update* up the hierarchy. The message *new_key* is always received from the parent, and will force a node to execute the function *CreateKey*. The only difference from a node executing *CreateKey* due to a received message, and the root executing *CreateKey*, is that the message contains the version of the key that must be created, while the root must create the version itself. The maximum number of messages required to create a key is twice the height of the hierarchy, for example, if the leaf with the lower hierarchy level executes an update on a key that does not exist, it must be propagated to the root, that will create the key, and then propagate the message *create_key* back to the leaf.

4.4.2 Read Operation

Similar with the update operation, nodes can also read values of keys that they do not contain. The operation that allows a node to read the value of a key it does not contain

Algorithm 6 CreateKey

Require: keyId, value, version, setNodesInterested (might come from a message)
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: $nodeInfo \leftarrow vv[nodeId]$
6: $nodeInfo.executedOperations \leftarrow nodeInfo.executedOperations + 1$
7: **if** *isRoot*() **then**
8: $\quad version \leftarrow CreateVersion(nodeId, nodeInfo.executedOperations)$
9: $newKey \leftarrow NewKey(keyId, version, value)$
10: $datastore \leftarrow datastore \cup \{newKey\}$
11: $setNodesInterested \leftarrow setNodesInterested \setminus \{nodeId\}$
12: $op \leftarrow NewOp(\text{"}new\_key\text{"}, keyId, value, version, setNodesInterested, vv)$
13: $log \leftarrow Append(log, op)$
14: **for** $n \in setNodesInterested \cap nodeInfo.childrenId$ **do**
15: $\quad SendMessage(n, op)$ {only one child can be in setNodesInterested}

is similar with the operation of updating a key it does not contain (Algorithm 4). The node will send a message *read_value* to its parent and will eventually receive a message *key_value* with the key value or an error message informing that the key does not exist. This will only happen if the message *read_value* reaches the root, and the root does not contain the key, which means that no node contains the key. If the key does exist, the first node that receives the message *read_value* will start to propagate the message *key_value* back to the node that was interested in the value of the key. Therefore, the best case of this operation in messages cost is 0 if the node contains the key in its datastore, or 2 if the node does not contain the key. The worst case is when a node leaf of the hierarchy tries to read the value of a key that does not exist or a key that only the root contains. In this case, the cost will be $2h$. When a node is waiting for the value of a key it can keep executing normally, it does not block itself waiting for the message with the key value.

Since the mechanism used by this operation is similar with the mechanism used when a node updates a key it does not contain, and the operation has no impact on the properties that will be verified (causal consistency, preservation of data, and eventual consistency) the specification of the protocol with TLA+ abstracted the operation and did not implemented it. The implementation of the operation would increase the complexity of the specification. It would generate many worthless execution traces, would require longer periods of time to verify its correctness, and would not change the end result of the protocol verification.

4.4.3 Operations for changing the hierarchy

When a new node tries to enter the hierarchy, it communicates with the root. If the new node has a valid datastore, i.e, it can be placed somewhere in the hierarchy without breaking the properties *a*) nodes must contain all keys of their children; *b*) and siblings

must contain disjoint datastores, then the node will eventually enter the hierarchy, otherwise, it will eventually be rejected. After communicating with the root, the node will be moved down the hierarchy until it reaches its place. The root will execute Algorithm 7. This function will decide what will happen to the new node. One of three things will happen: *a*) its datastore is not valid (breaks the hierarchy properties) and it is rejected; *b*) it will become child of the current node; *c*) it should communicate with a child of the current node to enter the hierarchy. If the new node datastore is contained on any child's datastore, it will either become descendant of that child, or will eventually be rejected by the child or by one of the child's descendants. A message is sent to that child, that after receiving it, will execute this same function (Algorithm 7). If the new node has keys in common with several children, the node will either become child of the current node, and parent of those children, or the operation will fail due to incompatible datastore. Incompatible datastore will happen if the set of children with keys in common with the new node' datastore is different from the set of children with datastores fully contained in its datastore. In this situation, the node cannot become parent of those children nor sibling, so it cannot enter the hierarchy. Otherwise, the function *AddNewNodeAsChild* will be executed, to add the new node in the hierarchy.

Algorithm 7 AddNode

Require: newNodeId, newNodeKeys (might come from a message)
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: $nodeInfo \leftarrow vv[nodeId]$
6: **if** $\exists\ c \in nodeInfo.childrenId : newNodeKeys \subseteq GetChildKeys(c)$ **then**
7: $\quad nodeInfo.executedOperations \leftarrow nodeInfo.executedOperations + 1$
8: $\quad op \leftarrow NewOp(\text{"new\_node"}, newNodeKeys, vv)$
9: $\quad log \leftarrow Append(log, op)$
10: $\quad SendMessage(c, op)$
11: **else if** $\{c \in nodeInfo.childrenId : GetChildKeys(c) \cap newNodeKeys\} = \{c \in nodeInfo.childrenId : GetChildKeys(c) \subseteq newNodeKeys\}$ **then**
12: $\quad AddNewNodeAsChild(newNodeId, newNodeKeys)$
13: **else**
14: $\quad Cancel$

When adding a node in the hierarchy, the Algorithm 8 is executed. This process requires actions of both nodes, the one being added, and the current node, that will be the parent.

Current Node The first step is to check which children have keys in common with the new node. Those children will become children of the new node, so the information must be changed accordingly. Information about their keys is removed, and the node must remove them from its version vector entry (each version vector entry contains a variable representing its children). After removing those children, the new child is added by executing three actions: *a*) creation of a new version vector entry regarding the new

node; *b*) addition of the new node identifier to the children variable of the current node version vector entry; *c*) and information about the new node's keys is stored; Finally, the node adds this operation to its log, and propagates a message *add_node_to_hierarchy* to its parent, to inform it that a new node was added in the hierarchy. Nodes that receive that message will add a new entry on their version vector, and keep propagating it up the hierarchy, until it reaches the root. Notice that nodes will only know that the node entered the hierarchy after processing the message *add_node_to_hierarchy*. Nodes that sent a message *add_node* to a child, do not know if the node will eventually enter the hierarchy, or due to incompatible datastore, will be canceled. If the new node is added as a leaf, the root will be the last node to be aware (nodes not related won't be informed).

New Node The new node will need to copy three pieces of information from the current node (its parent). First it will copy the version vector entries of nodes related to it, all but the siblings and their descendants. Second, the information needed from the datastore, i.e, versions and values of keys it is interested in. Third, the information about the keys of the children it inherited from its parent. After that, a message *new_parent* is sent to all children, informing them to change their parent and that a new node entered the hierarchy. These children might have executed some concurrent operations while the new node was being added. Since they did not yet know about their new parent, those operations were propagated to the old parent, the current node. In order to avoid conflicting versions of being simultaneously propagated through the hierarchy, the new node will be blocked, waiting for the children's acknowledgments. Only after that it can start to execute operations. The new node could also have executed offline operations, those operations will be applied right after the last child acknowledgment is received. After finding where the new node should be placed, and if the new node has children, it will cost 2 messages until the node is fully operational, one from the node to its children, and another back from its children.

Considering an example where the current node is node Y, the new node is W, and the child of the new node is Z. Also, it is assumed the hierarchy has more than these 3 nodes, and that Y is not the root. If new node W did not block waiting for the child Z acknowledge, and Z had, concurrently to the addition of W, executed and propagated an update, both Z and Y would have applied the update before W. If W then executed an conflicting update, since it is higher in the hierarchy than Z, it would win the conflict. The problem would be that Y would already be propagating a superseded update up the hierarchy (the one that lost the conflict) wasting unnecessary bandwidth.

When a node decides to leave the hierarchy, it executes Algorithm 9. In this algorithm, the node will ask its parent to leave the hierarchy, which in turn will be responsible for removing the node, and maintaining a correct hierarchy. The node can only ask its parent to leave after processing all its messages, and if it is not waiting for any acknowledgement. The purpose of this process is to guarantee that its parent and its children synchronize

Algorithm 8 AddNewNodeAsChild

Require: newNodeId, newNodeKeys
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: $nodeInfo \leftarrow vv[nodeId]$
6: $nodeInfo.executedOperations \leftarrow nodeInfo.executedOperations + 1$
7: $childrenNewNode \leftarrow GetChildrenKeysInCommon(newNodeKeys)$
8: $nodeInfo.childrenId \leftarrow nodeInfo.childrenId \setminus childrenNewNode \cup \{newNodeId\}$
9: $newNodeVVEntry \leftarrow NewEntry(newNodeId, nodeId, childrenNewNode, 0)$ {id of the node, its parent, its children, and known executed operations}
10: $vv \leftarrow vv \cup newNodeVVEntry$
11: **for** $kId \in newNodeKeys$ **do**
12: $datastore[kId].childInterested \leftarrow newNodeId$
13: New node will update its datastore and version vector by copying the required information from the current node (its parent)
14: $op \leftarrow NewOp("add\_node\_to\_hierarchy", newNodeId, vv)$
15: $log \leftarrow Append(log, op)$
16: $SendMessage(nodeInfo.parent, op)$
17: **for** $c \in childrenNewNode$ **do**
18: $SendMessage(c, NewOp("new\_parent", newNodeId))$

with each other as fast as possible, so a message *remove_child* is sent to its parent. This message will contain the keys its children are interested in, so that its parent can inherit those children, and store their information. After sending that message, the node will block itself, waiting for the parent's acknowledgement to become offline. Any messages received after that will be ignored. The only exception is if a message *new_parent* arrives. In that case, the node must process all messages until that, and ask permission to leave the hierarchy to the new parent. The messages that arrive before that message (message *new_parent*) must also be processed, to guarantee that causality is respected.

Algorithm 9 Remove

Require: {A node executes this operation when it wants to leave the hierarchy}
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: $nodeInfo \leftarrow vv[nodeId]$
6: $childrenKeys \leftarrow \{\}$
7: **for** $c \in nodeInfo.childrenId$ **do**
8: $childrenKeys \leftarrow childrenKeys \cup \{\{c, datastore.getKeysOf(c)\}\}$
9: $op \leftarrow NewOp("remove\_child", nodeId, childrenKeys, vv)$
10: $SendMessage(nodeInfo.parent, op)$

When a node receives a message *remove_child*, it executes Algorithm 10. This function (Algorithm 10) will start by removing the child's information (version vector entry and keys of interest), and then it will store information about the children it will inherit from the removed child. The received message contains the keys of each new children, so

the node can easily store that information. Since the node already had version vector entries for the new children, no new entries need to be created, it only needs to add those children to its current set of children. As said before, the child that sent the message *remove_child* stopped processing messages, so messages that this node concurrently sent to that child, were not processed and propagated to that child's children (that will now become children of the current node). This means that the new children might be missing some updates, so by comparing it own version vector with the version vector that came with the message (explained in section 4.4), the node is able to realize which messages need to be retrieved from the log, and re-propagated to each child. Along with those messages, a message *new_parent* will be sent to each new child. Furthermore, a message *remove_node* is propagated up the hierarchy, until it reaches the root. Nodes that receive this message remove the entry regarding the removed node from the version vector and keep propagating the message up the hierarchy. Finally, an acknowledgement is sent to the node that wants to leave the hierarchy. All messages (*new_parent*, *remove_child*, and *remove_node*) are sent at the same time.

The number of messages required to correctly remove a node from the hierarchy is 2, one to inform the parent, and other to send an acknowledgement. One way to reduce the cost to 1 would be if the node that wants to leave the hierarchy sent simultaneously a message to its children, informing about their new parent, and to the parent, informing about the new children, and then automatically become offline without waiting for the acknowledgments. Despite reducing the cost , this mechanism would raise some problems. If the parent of the node that wants to be removed changed at the same time as the node removed itself, the message *remove_child* would be wrongly sent to the old parent. Furthermore, the children of the node leaving the hierarchy would end up changing their parent to an incorrect one, the old parent of the removed node, instead of the new. This situation would increase the complexity of the remove protocol, and more messages would be needed to correct the hierarchy.

This subsection described two functions that cause nodes to send messages *new_parent* to their new children, Algorithm 10, due to the removal of those children old parent from the hierarchy, and Algorithm 8 due to addition of a new node in the hierarchy. Depending on whether a node was added or removed from the hierarchy, the receiver of the message *new_parent* will either add, or remove a new entry on its version vector, and will propagate a different message to all of its children (Algorithm 11 describes that situation). Depending on the message, the children will either add or remove an entry from their version vector, and will keep propagating the message down the hierarchy, until it reaches its leafs. Like other types of messages, the message *new_parent* includes the version vector of the source, i.e, the new parent (explained in section 4.4). Using that version vector, the current node will be able to calculate which messages the new parent is missing, and after retrieving them from the log, it re-propagates them along with a

Algorithm 10 RemoveChild

Require: childId, newChildren, newChildrenKeys, childVV
1: *nodeId*
2: *vv* {node's version vector}
3: *datastore* {node datastore. Contains information of its keys}
4: *log* {node's log}
5: $nodeInfo \leftarrow vv[nodeId]$
6: $nodeInfo.executedOperations \leftarrow nodeInfo.executedOperations + 1$
7: $nodeInfo.childrenId \leftarrow nodeInfo.childrenId \setminus childId \cup newChildren$
8: $childrenKeys \leftarrow \{\}$
9: **for** $k \in newChildrenKeys$ **do**
10: $datastore[k] \leftarrow UpdateChildInterested(datastore[k], newChildren, newChildrenKeys)$
11: **for** $c \in childrenNewNode$ **do**
12: $SendMessage(c, NewOp(\text{"new\_parent"}, nodeId))$
13: $SendMessages(c, GetChildMissingMessages(log, vv, childVV)$
14: $vv \leftarrow vv \setminus \{vv[childId]\}$
15: $op \leftarrow NewOp(\text{"remove\_node"}, childId)$
16: $log \leftarrow Append(log, op)$
17: $SendMessage(nodeInfo.parent, op)$

message *ack_parent*. Those missing messages are the ones propagated to the old parent, while the new parent was already online.

Algorithm 11 NewParent

Require: newParentId, parentVV
1: *nodeId*
2: *vv* {node's version vector}
3: *log* {node's log}
4: $nodeInfo \leftarrow vv[nodeId]$
5: $nodeInfo.executedOperations \leftarrow nodeInfo.executedOperations + 1$
6: $oldParent \leftarrow nodeInfo.parent$
7: $nodeInfo.parent \leftarrow newParentId$
8: **if** old parent was removed **then**
9: $vv \leftarrow vv \setminus \{vv[oldParent]\}$
10: $op \leftarrow NewOp(\text{"remove\_node"}, oldParent)$
11: **else**
12: $vv \leftarrow vv \cup parentVV[newParentId]$
13: $op \leftarrow NewOp(\text{"add\_node\_to\_hierarchy"}, newParentId, vv)$
14: **for** $c \in nodeInfo.childrenId$ **do**
15: $SendMessage(c, op)$
16: $log \leftarrow Append(log, op)$
17: $parentMissingMsgs \leftarrow Append(GetParentMissingMessages(log, vv, parentVV), \text{"ack\_parent"})$
18: $SendMessage(newParentId, parentMissingMsgs)$

4.4.4 Error Handling

As explained in Section 4.1, although messages never fail, every node but the root can fail. A node can fail with messages to process and propagate in its queue, and although it will not receive messages, its neighbors can still send it messages after it failed. Obviously, messages in both situations are lost, and in order to maintain causality and convergence

of data, nodes must eventually re-propagate them. For now, let's ignore how nodes detect failures, and focus on how they handle failures and maintain a correct hierarchy. As in the removal of a node, the responsibility to adjust the hierarchy will be from the parent of the node that failed, i.e, if the parent of the node detects the failure, it will handle it, if a child of the node detects the failure, it will inform the parent of the node (that will become its parent). Algorithm 12 shows a simplified version of this process.

Algorithm 12 Node Failed Detected

Require: failNodeId
1: *nodeId*
2: *vv* {node's version vector}
3: **if** *failNodeId* = *vv*[*nodeId*].*parent* **then**
4: *SendMessage*(*vv*[*failNodeId*].*parent*,"*node_failed*")
5: **else if** *failNodeId* ∈ *vv*[*nodeId*].*childrenId* **then**
6: *HandleFailure*()

The question now is why must the child waste a message informing the parent? Why can't the child handle the failure? If a child handled the failure, a situation where a node had recently been added in the hierarchy, but that information had not yet reached the child when the failure occurred (message *add_node_to_hierarchy*), the child would not be able to correctly handle the failure, and would end up communicating with the wrong node. The hierarchy in Figure 4.4 represents a situation where a new node N was added. At that moment, only its neighbors (nodes 2 and 3) know about its existence, nodes 1 and 4 did not yet received the message *add_node_to_hierarchy*. This figure shows the evolution of the hierarchy to two different and independent situations, *a*) node 3 fails; *b*) nodes 2 and 3, the only ones that know about the existence of N, fail. When a node detects a neighbor failure, it can verify which other related nodes failed, i.e, in situation *b)*, after realizing that node 3 failed, node 4 would be able to verify that node 2 also failed. In both situations, if node 4 detected the failure of its parent, and handled the error, it would send its information to the wrong nodes, nodes 2 and 1 respectively. Instead, a simple message *node_failed* with the identifier of the node that failed is sent to those nodes. Nodes that detect failures and start correcting the hierarchy, block until the hierarchy is corrected, only processing messages to that end. Those messages start with the function *HandleFailure* of the Algorithm 12, that will trigger functions similar to Algorithm 10 and Algorithm 11. Their goal is to guarantee that nodes that failed are removed, the hierarchy is corrected, the missing messages are re-propagated, and the parent of the node that failed stores information about its new children . The function *HandleFailure* will be later explained with more detail.

Situation A

- If node 4 detects the failure, it will send a message *node_failed* to node 2. Node 2 will realize that the failed node was not its child, and will send the message *node_failed* to node N, the parent of node that failed.

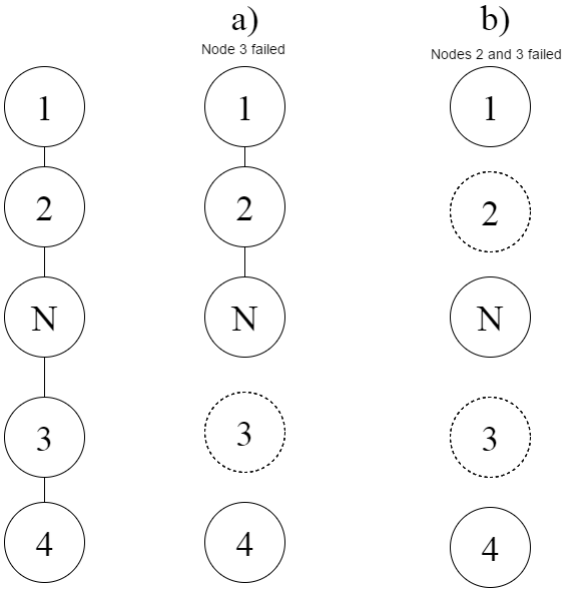

Figure 4.4: Example of nodes failures.

- If node N detects the failure, it will start the process of handling the failure. It will exchange messages with node 4, that will realize that it is its new parent, and should be added.

Situation B

- If node 4 detects the failure of node 3, it also detects the failure of node 2, and will send a message *node_failed* to node 1.

- If node N detects the failure of node 3, it will start the process of handling the failure. It will exchange messages with node 4, that will realize that it is its new parent, and should be added.

- If node N detects the failure of node 2, it will send a message *node_failed* to node 1.

- If node 1 detects the failure of nodes 2 and 3, although it does not know about node N, it is aware that it tried to enter the hierarchy. Node 1 also knows that if node N entered the hierarchy, it was related with nodes 1 and 2 (Algorithm 7, line 8 and 9). Therefore node 1 will communicate with N to verify if it is online, and if so, checks its position. Node 1 then verifies that N was neighbor of the nodes that failed, and should become its child instead of 4. Therefore, node 1 will exchange messages with node N to inform it about the nodes that failed and to correct the hierarchy. As soon as node N receives information about the failure of node 3, it starts exchanging messages with node 4.

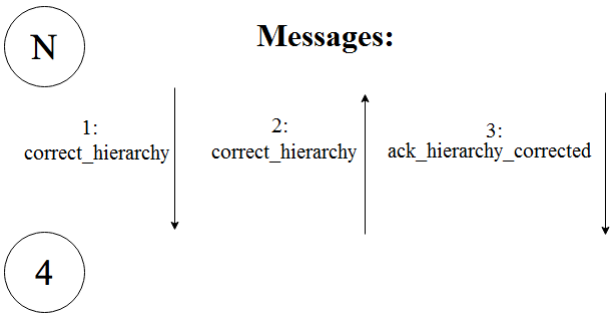

Figure 4.5: Example of failure being handled.

After a parent realizes one of its child has failed, the process to correct the hierarchy takes 3 messages and starts with the function *HandleFailure* (Algorithm 12). Figure 4.5 uses the example *a)* from Figure 4.4, where node N detects that its child, node 3, failed. On this situation, besides correcting the hierarchy, node N needs to know which keys its new child (node 4) is interested in, and needs to know which messages node 4 is missing due node's 3 failure. On the other hand, node 4 only needs to know which messages its new parent, node N, is missing due node's 3 failure. When a node executes the function *HandleFailure* it will:

1. Detect which nodes should become its children due to its child failure, and detect which other nodes, descendant of the child that failed, also failed;

2. Remove failed nodes from its version vector, and remove information about which keys the child that failed was interested in;

3. Find out which nodes that tried to enter the hierarchy (and the node still does not know if they did), might be descendants of the child that failed, and are online. Check which of those nodes should become its children, instead of the ones from step 1. This step is what allows node 1 in Figure 4.4, example *b)* to find out about node N.

4. Sends a message *correct_hierarchy* to each new child, as *message 1* send to node 4 in Figure 4.5. This message contains the version vector of the current node, plus a set with the nodes that failed. Since node N does not know about the version vector of node 4, it does not know how many messages node 4 is missing and need to be re-propagated.

When a node receives a message *correct_hierarchy* (*message 1* in Figure 4.5) from a node higher in the hierarchy, it will:

1. Remove the nodes that failed from its version vector;

2. If it does not know the source of the message, adds it as its parent (Figure 4.5 where node 4 does not know node N);

3. Use the version vector that came with the message *correct_hierarchy*, to calculate which messages the source is missing, and along with a message *correct_hierarchy*, re-propagates them to the source. This is the *message 2* from Figure 4.5, and will contain its version vector, and its keys, so that the source (its new parent) can store that information.

When a node receives *message 2* from Figure 4.5 it will:

1. Store the information about its keys (that came with the message). This message came from a child;

2. Use the version vector that came with the message to calculate which messages the source is missing, and along with a message *ack_hierarchy_correct*, re-propagates them to the source/child. This is the *message 3* from Figure 4.5;

3. Since the hierarchy is corrected and all information received/sent, the node will unblock itself;

When a node receives the last message, *message 3* from Figure 4.5, it means that process to correct the hierarchy has ended, and it unblocks itself.

Chapter 5

Protocol Specification

This section describes the specification of the developed protocol, and presents some of the relevant parts of the TLA+ specification.

5.1 Constants and Variables

The TLA+ specification uses constants and variables to define the model of the system (see Figure 5.1). The protocol specification has two constants, *NodeId* and *KeyId* to describe the set of all node and key ids, respectively. The overall system is described by four variables, *configuration* represents the local state of each node, *msgs* represents the message queue of a node. It contains messages already received but not yet processed by a node, *offlineNodes* represents the set of offline nodes that can be added to the hierarchy, and *failedNodes* represents the set of nodes that failed and still have not been handled.

As said in Chapter 4, messages are not lost, and in order to maintain causality, they must be processed in the same order that they were sent. To keep the order of the messages, a queue is used. When node x sends a message m to node y, it appends m at the end of y's message queue, i.e, $append(msgs[y], m)$.

Variable *configuration* is defined as a function that maps each *NodeId* to *State*, a structure representing the local state of each node. Figure 5.1 shows the TLA+ specification used to represent the state of each node. The local *State* is a tuple where:

nodeId - an element of the set *NodeId* to identify the node. Unique for each node;

datastore - represents the datastore of each node. It maps each element of the set *KeyId* to either *Key* or *NullKey*, depending if the node contains the key in its datastore or not (by "contains", it means that the function of its datastore will return *Key*

instead of *NullKey*). Although a node could infer all the existing keys with this representation of the *datastore*, this knowledge is not used by the nodes in any function, if an element of the *datastore* is mapped to *NullKey* the node ignores it, the behavior is the same as if that key identifier was not in the domain of the function. This representation is used because TLA+ does not support partial functions;

vv- represents the version vector of a node. A version vector only needs to contain entries for nodes related to the current node, i.e, nodes with keys in common. It maps each *nodeId* (each element of the set *NodeId*) to either *NodeInformation*, if the the node represented by *nodeId* is related to the current node, or *NullNodeInformation* if it isn't. The reason to map a *nodeId* to *NullNodeInformation* is the same as the function of the *datastore*, TLA+ does not support partial functions. In the hierarchy of Figure 4.1, node W is related to nodes Y and X, so it would contain version vector entries for them, while node Z would only have an entry for node X;

waiting_acks - a set of elements to halt, or force a node to execute some operations. If a node contains any element of the type *"ack_parent"*, *"correct_hierarchy"*, or *"ack_remove"*, it halts until a specific message arrives. A node is forced to read a message if it contains an element of the type *"receiving_package"*, *"repeat_now"*, or *"offline_operations"*. These elements are removed from the set when a specific message is received and processed by the node. Each type will be explained in Section 5.6;

status - it contains two elements, *online*, a string to represent the current status of the node ("online", "pending", or "offline"), and *numOpOnline*, a natural number representing the number of operations that the node had executed when it went offline. This number is used to calculate how many operations were executed offline, and to calculate which operations might need to be re-propagated, so that operations before the *numOpOnline* operation won't be propagated again;

log - it contains the sequence of operations executed by the node. The size of the log will be equal to the number of the counter (*executedOperations*) of its own version vector entry. In Figure 5.1, the log sequence is simplified due the number of different operations that can be part of the sequence. There are 6 types of operations that can appear on the log, in Figure 5.1, the specification only shows 2.

Each element *Key* is represented by four elements: *a*) *keyId* to identify the key; *b*) the current *value* of the key represented by an integer; *c*) *versionId* represented by a *nodeId* and an integer *versionNumber* to identify which node created the current update; *d*) and *childInterested*, a variable represented by a *NodeIdOrNull* (*nodeId* or *NullNodeId* to specify that no child is interested in the key) used to identify which child is interested in this key. If an update is done to the key, this variable identifies to whom propagate the update.

Since children have disjoint datastores, only one child can be interested in the key, so this variable is represented with a single value.

Tuple *NodeInformation* (each entry of the version vector) is represented by four elements: *a*) *nodeId* to identify which node the version vector entry corresponds to; *b*) *parent*, a *NodeIdOrNull* that identifies the parent of that node in the hierarchy; *c*) *childrenId*, a subset of the set *NodeId* to represent the children of the node in the hierarchy. *d*) and *executedOperations*, a natural number identifying the number of executed operations that the current node knows that the correspondent node executed. This value represents global operations applied.

5.2 Initialization of the system

The protocol is initialized with the *Init* predicate presented in Figure 5.2, which generates all possible initial states by assigning values to all variables used in the specification. As already explained, the protocol uses four different variables, *configuration*, *offlineNodes*, *msgs*, and *failedNodes*. Variable *msgs* is initialized by mapping each *nodeId* to an empty queue (no messages), while the set *offlineNodes* is initialized by selecting a non-empty subset nodes, ensuring there is at least, one online node. Obviously, the variable *failedNodes* is initialized as an empty set. These 3 variables are initialized on the first three clauses of the *Init* predicate. The *configuration* keeps the state of all nodes of the system, so all nodes will be initialized. As previously explained, the root of the hierarchy will have the full datastore, however, the protocol allows nodes to create new keys, so in terms of the specification, a subset of *KeyId* will not be considered in the initialization. These keys are represented with the variable *newKeys* of the *Init* predicate, and can later be used to add new keys in the datastore.

To initialize the hierarchy, i.e, the *configuration* of online nodes, three functions are used for each of those nodes: *a*) one to select a *nodeId* of the set *onlineNodeIds* or *NullNodeId* to be the parent; *b*) another to select a subset of *onlineNodeIds* to be the children; *c*) and the last, to select a subset of *initialKeys* to be the keys in the datastore of the node. Without any restrictions, the assignment of values to those three variables can generate many combinations. As a matter of efficiency, before executing the main function of the initialization, *InitState*, some basic validations on the results of those three functions is done. These validations verify if the root is the only online node to have no parent, that the root contains all the created keys, and if a node contains a parent, the parent has it as child. Additionally, all the different initial states will use the same node as root, due to TLC always selecting the same element in the function *CHOOSE*. The use of the same root will prevent TLC from testing unnecessary initial states. Those cases will be explained in Chapter 6. The initialization of most state's variables is straightforward ,

CONSTANTS *NodeId*, *KeyId*
vars $\triangleq \langle configuration, msgs, offlineNodes, failedNodes \rangle$

$NullNodeId \triangleq CHOOSE\ x : x \notin NodeId$
$NodeIdOrNull \triangleq NodeId \cup NullNodeId$

$NullKeyId \triangleq CHOOSE\ x : x \notin KeyId$
$KeyIdOrNull \triangleq KeyId \cup NullKeyId$

$VersionId \triangleq$
[*nodeId* : *NodeIdOrNull*,
versionNumber : *Int*]

$NullKey \triangleq$
[*keyId* : {*NullKeyId*}]

$Key \triangleq$
[*keyId* : *KeyId*,
value : *Int*,
childInterested : *NodeIdOrNull*,
versionId : *VersionId*]

$Ack \triangleq$
[*NodeId* : *NodeId*,
type : *STRING*]

$Status \triangleq$
[*online* : *BOOLEAN*,
numOpOnline : *Nat*]

$NodeInformation \triangleq$
[*nodeId* : *NodeId*,
executedOperations : *Nat*,
parent : *NodeIdOrNull*,
childrenId : *SUBSETNodeId*]

$NullNodeInformation \triangleq$
[*nodeId* : {*NullNodeId*}]

$State \triangleq$
[*nodeId* : *NodeId*,
datastore : [*KeyId* $\rightarrow$ *Key* $\cup$ *NullKey*],
vv : [*NodeId* $\rightarrow$ *NodeInformation* $\cup$ *NullNodeInformation*],
waiting_acks : *SUBSET Ack*,
status : *Status*,
log : *Seq*(*MsgKeyUpdate* $\cup$ ... $\cup$ *LogAutoRemove*)]

Figure 5.1: Node specification

the set of *waiting_acks* starts empty, and the log starts as an empty sequence. The two variables that require some explanation are the version vector (*vv*), and the *datastore*:

datastore - The function *c)* of the Init predicate (Figure 5.2) will return a set of *keyId* for each node. For each *keyId* of that set, an element will be initialized (tuple *Key* of Figure 5.1). The element has 3 variables besides the *keyId*, a *versionId* which will be initialized as a null version, the *value* which will start with 0, and *childInterested* which will be initialized with the identifier of one of the node's children, i.e, a *nodeId* of the set returned by the function *b)* of Figure 5.2 (this function attributed a set of children to each online node). If none of those children have the *keyId* on their set

$$
\begin{array}{l}
Init \triangleq \\
\quad LET \\
\qquad initOfflineNodes \triangleq CHOOSE\ x \in SUBSET\ NodeId : x \neq NodeId \\
\qquad onlineNodeIds \triangleq NodeId \setminus offlineNodes \\
\qquad newKeys \triangleq CHOOSE\ nk \in SUBSET\ KeyId : nk \neq KeyId \wedge Cardinality(nk) \neq 0 \\
\qquad initialKeys \triangleq KeyId \setminus newKeys \\
\qquad rootId \triangleq CHOOSE\ rId \in onlineNodeIds : TRUE \\
\quad IN \\
\qquad \wedge\ offlineNodes = initOfflineNodes \\
\qquad \wedge\ failedNodes = \{\} \\
\qquad \wedge\ msgs = [n \in NodeId \mapsto \langle\rangle\,] \\
\qquad\quad (* \ Function\ a) *) \\
\qquad \wedge\ \exists\, parent \in [onlineNodeIds \rightarrow onlineNodeIds \cup \{NullNodeId\}] : \\
\qquad\quad (* \ Function\ b) *) \\
\qquad\quad \exists\, children \in [onlineNodeIds \rightarrow SUBSET\ onlineNodeIds] : \\
\qquad\quad (* \ Function\ c) *) \\
\qquad\quad \exists\, keys \in [onlineNodeIds \rightarrow SUBSET initialKeys] : \\
\qquad\qquad \wedge keys[rootId] = initialKeys \\
\qquad\qquad \wedge parent[rootId] = NullNodeId \\
\qquad\qquad \wedge \forall\, n \in children[rootId] : parent[n] = rootId \\
\qquad\qquad\ \wedge \forall\, n \in onlineNodeIds \setminus \{rootId\} : \quad \wedge parent[n] \neq NullNodeId \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \wedge n \in children[parent[n]] \\
\qquad\qquad \wedge configuration = InitState(children, keys, parent) \\
\qquad\qquad \wedge ValidHierarchy
\end{array}
$$

Figure 5.2: Init predicate

(result of the function *c)*), *NullNodeId* will be assigned to *childInterested*. Below is a simplified example of the initialization of the *datastore* of each node. As mentioned, keys not relevant to the node are mapped to *NullDatastoreEntry*;

$$NullVersion \triangleq [nodeId \mapsto NullNodeId, versionNumber \mapsto -1]$$

$$
\begin{array}{l}
datastore \triangleq \quad [k \in keys[nodeId] \mapsto \\
\qquad\qquad LET \\
\qquad\qquad\qquad CIS \triangleq \{cId \in children[nodeId] : k \in keys[cId]\} \\
\qquad\qquad\qquad CI \triangleq IF\ CIS = \{\} \\
\qquad\qquad\qquad\qquad THEN\ \ NullNodeId \\
\qquad\qquad\qquad\qquad ELSE\ \ CHOOSE\ cId \in CIS : TRUE \\
\qquad\qquad IN \\
\qquad\qquad\qquad [keyId \mapsto k, \\
\qquad\qquad\qquad\ values \mapsto 0, \\
\qquad\qquad\qquad\ childInterested \mapsto CI, \\
\qquad\qquad\qquad\ versionId \mapsto NullVersion]]
\end{array}
$$

vv- As explained, each node only needs to keep version vector entries of nodes with keys in common. Those nodes are called related nodes. Nodes not related will never need

to communicate, so is unnecessary to keep information about them. Therefore, the first step to initialize the version vector of online nodes is to find out which other nodes have keys in common. All nodes are initialized with 0 executed operations, so the initialization of each version vector entry is straightforward. Below is the initialization of the version vector of online nodes:

$$nodesRelated \triangleq \{n \in onlineNodes : keys[nodeId] \cap keys[n] \neq \{\}\} \cup \{nodeId\}$$

$$\begin{array}{lll} vv \triangleq & [n \in NodeId \mapsto & \\ & \quad IF & n \in nodesRelated \\ & \quad THEN & [nodeId \mapsto n, \\ & & \ executedOperations \mapsto 0, \\ & & \ parent \mapsto parents[n], \\ & & \ childrenId \mapsto children[n] \\ & \quad ELSE & NullVVEntry] \end{array}$$

In order to simulate the different nodes that could enter the hierarchy, the specification allows offline nodes to add/drop keys from their datastore, so offline nodes will always be initialized with a full datastore. Those nodes will also be initialized with no parent and no children, i.e *parent* = *NullNodeId* and *childrenId* = {}. Besides that, the version vector of each offline node will be initialized with all entries but its own, as *NullVVEntry*, and all its keys will have the variable *childInterested* as *NullNodeId*.

After the initialization of all variables, the property *ValidHierarchy* is verified to guarantee that each initial state represents a valid hierarchy (not all hierarchies generated by those conditions are valid). This property will be explained in Section 6.1 along with the other properties that the protocol ensures.

5.3 Next state

The *Next* state action describes the protocol functionalities and will execute all possible execution traces of the protocol. To this end, all functionalities have to be specified.

The *Next* state action is divided in two phases, the first one, where is verified if any node is being forced to execute the function *ProcessMessage*, and the second, where any node can execute any enabled operation.

In Section 5.1, when the set *waiting_acks* was described, it was mentioned that there are situations in the execution of the specification, where a node is forced to read a message. The first phase of *Next* is necessary to handle those situations. When a node needs to re-execute an operation, apply offline operations after receiving the confirmation of all its children, or when a node starts to read a pack of messages, an acknowledge is inserted in the *waiting_acks* of that node, to force it to execute the operation *ProcessMessage* on the *Next* state action (those situations will be later explained).

In the second phase, several actions may happen:

- an online node that is not waiting for any acknowledge can execute a *Put* or a *Remove* function (root cannot execute *Remove*);
- an online node but the root can fail;
- if the root is not waiting for any acknowledge, and if there is any offline node, the root can add a new node in the hierarchy;
- an offline node can add or drop a key from its datastore;
- a node can read a message. If the node is waiting for any acknowledge, it can only read a message if the first message of the queue is the acknowledge that the node is waiting for. If a node contains a message in its queue and is not blocked with an acknowledge (or is blocked and the message satisfies the acknowledge) the protocol will always be able to execute an operation, i.e, it will not reach a deadlock.

Figure 5.3 shows the second phase of the *Next* specification.

$$
\begin{array}{l}
Next \triangleq \\
\quad LET \quad onlineN \triangleq \{n \in NodeId : configuration[n].status.connectionStatus = \text{"online"}\} \\
\qquad\quad rootId \triangleq \ CHOOSE\ n \in onlineN : configuration[n].vv[n].parent = NullNodeId \\
\quad IN \\
\qquad \lor \exists\, n \in onlineN : \ \lor ProcessMessage(n) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor Remove(n) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor Fail(n) \\
\qquad\qquad\qquad\qquad\qquad\quad \lor \exists\, k \in KeyId, newValue \in 1..10 : Put(n,k,newValue) \\
\\
\qquad \lor \exists\, n \in OfflineNodeId, \exists\, k \in KeyId : \\
\qquad\qquad\qquad\qquad \lor RemoveKeyOffline(n,k) \\
\qquad\qquad\qquad\qquad \lor AddKeyOffline(n,k) \\
\qquad\qquad\qquad\qquad \lor \exists\, newValue \in 1..10 : \\
\qquad\qquad\qquad\qquad\qquad\qquad \land configuration[n].datastore[k] \neq NullDatastoreEntry \\
\qquad\qquad\qquad\qquad\qquad\qquad \land Put(n,k,newValue) \\
\\
\qquad \lor \ \land offlineNodes \neq \{\} \\
\qquad\qquad \land LET \\
\qquad\qquad\qquad\qquad nOff \triangleq \ CHOOSE\ n \in offlineNodes : TRUE \\
\qquad\qquad\quad IN \\
\qquad\qquad\qquad\qquad \land configuration[rootId].waiting\_acks = \{\} \\
\qquad\qquad\qquad\qquad \land AddNode(rootId, nOff)
\end{array}
$$

Figure 5.3: Next State

5.4 Messages

Different actions will generate different messages. There are several different messages to propagate datastores updates, as well as changes in the hierarchy. This section won't list all the different types of messages, each message will be explained with the operation that generates it. Figure 5.4 shows the specification used to propagate a key update. All messages contain two common fields, *sourceId* to identify who sent the message, and *msgType* to identify the type of message. As explained in Section 4.4, most of the messages also contain the version vector of the node that sent it, field *sourceVV*.

$$
\begin{aligned}
&MsgKeyUpdate \triangleq \\
&\quad [msgType: \ STRING, \\
&\quad\ sourceId: \ NodeId, \\
&\quad sourceVV: \ [NodeId \rightarrow NodeInformation \cup NullNodeInformation], \\
&\quad\ keyId: \ KeyId, \\
&\quad newValue: \ Int, \\
&\quad\ version: \ VersionId]
\end{aligned}
$$

Figure 5.4: Message used to propagate a key update

When node n processes a message, the function *ProcessMessage* is executed. This function starts by verifying that the node's message queue is not empty, the node is online, and the node is either not waiting for any acknowledge (the set *waiting_acks* is empty), or the next message will satisfy the acknowledge blocking the node. If those conditions are met, the value of the element *msgType* is verified, and the proper function is executed, i.e:

$$
\begin{aligned}
IF \quad & msgType = "key\_update" \\
THEN \quad & ReceivedKeyUpdate(n) \\
ELSE\ IF \quad & msgType = "new\_key\_or\_update" \\
THEN \quad & ReceivedNewKeyOrUpdate(n) \\
& ...
\end{aligned}
$$

Since most of the messages contain a version vector, when a node executes a function due to a received message, it will use the *sourceVV* to update its own version vector. Since nodes only communicate with their neighbors, it is easy to determine if a message came from a child, the entries of *sourceVV* (the version vector of the child) referring to nodes higher in the hierarchy, i.e, nodes above the node processing the message, will be outdated and will not be used to update the version vector. The only way for a child to know about operations executed higher in the hierarchy will be by receiving a message from its parent. For that reason, the function used to update the node's version vector will ignore entries that can be outdated. Figure 5.5 is a simplified version of that function. The input variable *nodesAbove* is calculated using the version vector (specifically the variable *parent*), to reconstruct the hierarchy, and verify which nodes are higher in the hierarchy. The variable *sourceIsAboveInHierarchy* is a boolean, it is true if *sourceId* is in

nodesAbove. When updating the version vector entry, only the variable *executedOperations* is copied due the possibility of old messages, that can have outdated version vectors, being re-propagated.

$$
\begin{array}{l}
UpdateVV(nodeId, vv, sourceId, sourceVV, nodesAbove, sourceIsAboveInHierarchy) \triangleq \\
\quad [nId \in NodeId \mapsto \\
\qquad IF \quad \lor \land sourceIsAboveInHierarchy \\
\qquad\qquad \land nId \in nodesAbove \\
\qquad\qquad \lor \land sourceIsAboveInHierarchy = FALSE \\
\qquad\qquad \land nId \notin nodesAbove \\
\qquad THEN \quad [vv[nId] \; EXCEPT \; !.executedOperations = \\
\qquad\qquad\qquad\qquad Max(vv[nId].executedOperations, \\
\qquad\qquad\qquad\qquad\quad sourceVV[nId].executedOperations)] \\
\qquad ELSE \quad vv[nId]]
\end{array}
$$

Figure 5.5: Update of version vector

There are situations where a node might need to re-propagate a set of messages, for example in Algorithm 11, where a node might verify that its new parent is missing several messages. Since those messages are sent at the same time, the specification of the protocol processes all message at the same time, i.e, if a node processes the first message of a pack of messages, it will process all the other messages before executing any other operation. Obviously, if a node processes a pack of messages and ends up propagating more than one update to one of its neighbors, its neighbor should also receive a pack, instead of several independent messages. The use of these packs do not change the end result, however, besides being a functionality that makes sense in a functional point of view, it also diminishes the number of execution traces that the specification can generate by forcing it to execute the function *ProcessMessage* in the *Next* action until all messages are processed. This functionality was specified with two distinct messages, and an acknowledge to force the *Next* action to execute the function *ProcessMessage* of the node that contains it.

When a node needs to send several messages (sequence *msgsToSend*), it will append the message *msgPS* in the beginning of that sequence, and *msgPE* in end, i.e:

$$
\begin{array}{l}
msgPS \triangleq [msgType \mapsto \text{"package\_start"}, sourceId \mapsto n] \\
msgPE \triangleq [msgType \mapsto \text{"package\_end"}, sourceId \mapsto n] \\
ack\_msgPack \triangleq [nodeId \mapsto message.sourceId, type \mapsto \text{"receiving\_package"}] \\
finalMsgsToSend \triangleq \quad IF \quad Len(msgsToSend) > 1 \\
\qquad\qquad\qquad\qquad THEN \quad Append(\langle\langle msgPS \rangle\rangle \circ msgsToSend, \; msgPE) \\
\qquad\qquad\qquad\qquad ELSE \quad msgsToSend
\end{array}
$$

As mentioned earlier, when a node receives a pack of messages, it might also end up propagating a package for other nodes. Since the function used to process the message

package_start cannot know if a package will end up being propagated to other node, and if it is, to whom, the specification of this function will append a message *package_start* in the end of the message queue of every node. In order to process all messages of the pack in sequential *Next* actions, when a node n processes the first message of the pack (*msgPS*), the specification of the protocol adds the *ack_msgPack* to its acknowledge set, and the *Next* action will keep executing the function *ProcessMessage*. In Section 5.3 it was explained that the presented specification of *Next* (Figure 5.3) was missing the conditions that might force the execution of function *ProcessMessage*. The first of those condition will be satisfied after the node adds the acknowledge *ack_msgPack*:

$$\exists\ n \in onlineNodes : \quad \exists\ ack \in configuration[n].waiting\_acks : \\ \vee \quad ack.type = \text{"}receiving\_package\text{"}$$

In order to stop this condition, the *ack_msgPack* must be removed from the node's acknowledge set. That is the first step of the function specified to process the message *package_end*. That message marks the end of the messages pack, so at this point, it can be verified to whom the node propagated a pack of messages. To those nodes, a message *package_end* must be added in the end of their message queue to close their pack, and to the others, the *package_start* must be removed. So, for each node:

- If the last message of its message queue is a message with type *package_start*, it means that the node processing the package did not send it any message, so that message is removed.

- If the node only contains one message after the message *package_start*, it means that the node processing the package only sent it one message. For only one message the package is not needed, therefore the message *package_start* is removed.

- If the node contains more than one message after the message *package_start*, a message *package_end* is appended to its message queue.

This process of removing the message *package_start* when it is not needed is only done to reduce the number of states of the model checker, empty packs or packs with only one message would not affect the end result.

5.5 Node failures

As already explained, each node but the root can fail, and when it does, its *nodeId* is added in the set *failedNodes*. The specification of the error handling starts by inserting a message *node_failed* in the beginning of the message queue of the node that detects the error. Instead of placing this message in the queue of each neighbor, assuming that nodes are always checking the liveness of their neighbors, nodes will receive the message if:

- they try to send a message to an offline node. At the same time as a message is placed on the queue of the destination (the failed node that will never process the message), the message *node_failed* will be placed in the message queue of the node sending the message;
- they sent a message and the destination failed before processing it. When a node fails, function *Fail* checks all of its messages yet to process, and places a message *node_failed* on the message queue of all nodes that sent those messages.

This specification to detect errors assumes that an implementation would use a mechanism to detect whether a message was received by the destination. This mechanism assumes that messages are not lost when two online nodes communicate, the sender can guarantee that the receiver has received the message.

When a fail is detected, besides adding the message *node_failed*, the node also adds an acknowledge to its *waiting_acks* set. This will prevent the node from executing any operations but the ones needed to correct the hierarchy. The doubt here was whether or not to put the message in the beginning of the message queue, making nodes automatically detect failures when sending a message, and preventing them from keep executing operations. If this was not the case, i.e, if instead the message was placed in the end of the queue, the end result would be the same. The only difference would be the node being able to send several messages to a failed node, ending with several messages *node_failed* on its message queue. In terms of the model checker, this would only generate extra worthless execution traces. Notice that although the node is halted, other nodes of the system can keep executing operations. The ability to guarantee fault tolerance introduced some complexity into the specification of almost all operations. Most of the extra details needed will be explained during the next section. One detail used in several operations is a situation where a node processes a message of a node that was already removed due to a failure. Obviously, those messages cannot be ignored. As explained in section 5.4, nodes update their version vector based on the one received with messages. To update the version vector, nodes need to know if the source of the message is from up or down the hierarchy, which is verified by calculating the set *nodesAbove* with the current version vector. Figure 5.6 is an example of that situation. On this example, both nodes execute a concurrent update, but before receiving/applying the update of node X, node Y fails. As explained, node X will realize this fail, and will handle it before applying the received message. By the time it executes the received update, the information about node Y was already removed. On these situations, the node will need to use the *sourceVV* to verify if the message came from a node higher or lower in the hierarchy. Nodes never send messages to nodes they do not know, so the version vector that came with the message will always contain an entry of the current node. Thus, to verify the location of the source of the message, a node will execute:

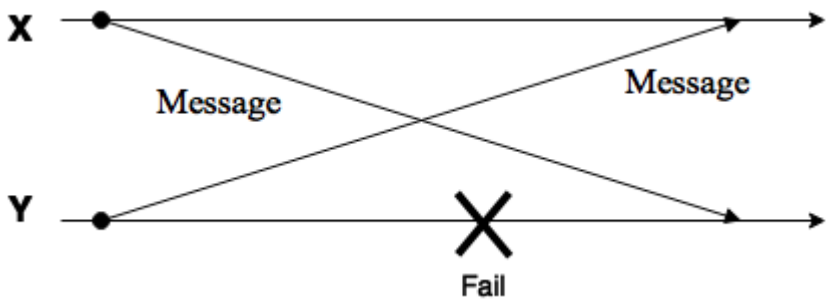

Figure 5.6: Receiving a message from an offline node.

$nodesAbove \triangleq GetNodesAboveInHierarchy(vv, nodeId)$

$sourceIsAboveInHierarchy \triangleq$
$\quad \vee \quad \wedge \quad vv[sourceId] \neq NullVVEntry$
$\quad \quad \wedge \quad sourceId \in nodesAbove$
$\quad \vee \quad \wedge \quad vv[sourceId] = NullVVEntry$
$\quad \quad \wedge \quad nodeId \notin GetNodesAboveInHierarchy(sourceVV, sourceId)$

5.6 Operations specification

This section will explain how the different operations, introduced in Section 4.4, were specified (implemented in TLA+), and the challenges addressed. Most of the operations will be exemplified with a practical example. Those examples will always use the hierarchy of Figure 4.1.

5.6.1 Updating known keys

Recalling Section 4.4.1, it was explained that all operations altering the datastore start with the execution of a *Put* function. When a node executes that function, one of two things will eventually happen, a key will be updated, or a key will be created. Figure 4.2 showed the simplest situation, a node updates a key that it contains, and a message is propagated. The specification of the first operation, the original $Update(n, k, newValue)$ (Algorithm 2), has three main tasks, the creation of the new version, the update of the datastore, and the creation of the message:

$updated\_version \triangleq$
$\quad [nodeId \mapsto n,$
$\quad versionNumber \mapsto vv[n].executedOperations]$

$updated\_datastore \triangleq$
$\quad [datastore\ EXCEPT \quad ![k].value = newValue,$
$\quad\quad\quad\quad\quad\quad\quad\quad ![k].versionId = updated\_version]$

$$
\begin{aligned}
&message \triangleq \\
&\quad [msgType \mapsto \text{"key\_update"}, \\
&\quad sourceId \mapsto n, \\
&\quad sourceVV \mapsto vv, \\
&\quad keyId \mapsto k, \\
&\quad newValue \mapsto newValue, \\
&\quad version \mapsto updated\_version]
\end{aligned}
$$

That message must be sent to its parent (if the node is not the root) and any interested child. It will be appended in the end of their messages queue, i.e:

$$destionations \triangleq vv[n].parent \cup datastore[k].childInterested$$

$$
\begin{array}{lll}
msgs \triangleq [nId \in NodeId \mapsto & IF & nId \in destionations \\
& THEN & Append(msgs[nId], message) \\
& ELSE & msgs[nId]]
\end{array}
$$

This specification of the messages is simplified, as mentioned in Section 5.5, one way to detect node failures is when sending a message, so in the full specification, actions must be taken regarding offline nodes in *destinations*.

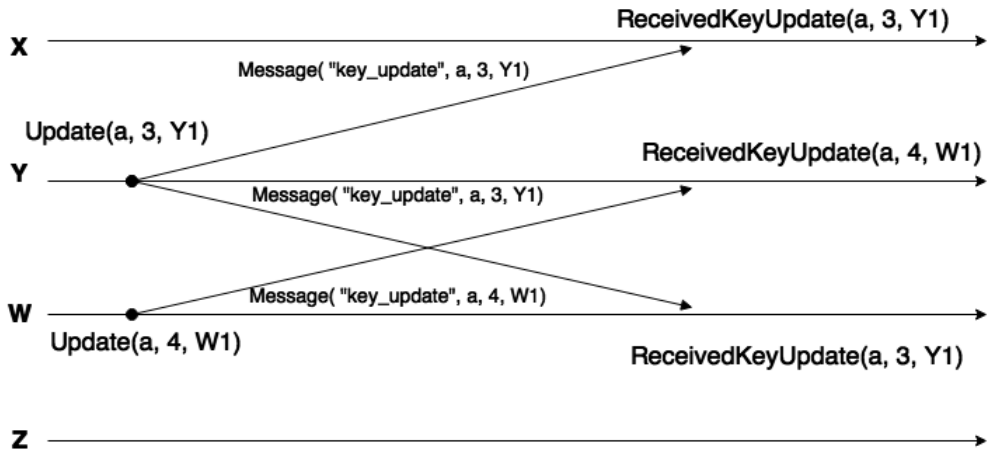

Figure 5.7: Example of conflicting updates.

Due to the ability of nodes to operate concurrently, some key updates will raise a conflict. Figure 5.7 shows an example of one of those situations (uses the hierarchy of Figure 4.1). In the example, nodes Y and W concurrently execute an update, and propagate it to their neighbors. When they receive the message *key_update* from one another and execute the function *ReceivedKeyUpdate*, they both detect the conflict. This conflict must be handled in the same manner, otherwise, the convergence of the data, which the protocol ensures, would fail. When handling a conflict, a node must decide which version of the update wins. If the received update wins the conflict, its version

is applied and keeps being propagated in the same direction. If the received update is not to be applied, the received message is simply ignored. There is no need to send the current version of the key (version that won the conflict) to the source of the losing update, because that message was already sent when the current node updated the key. In the example, node *W* applies the update, and node *Y* ignores the message.

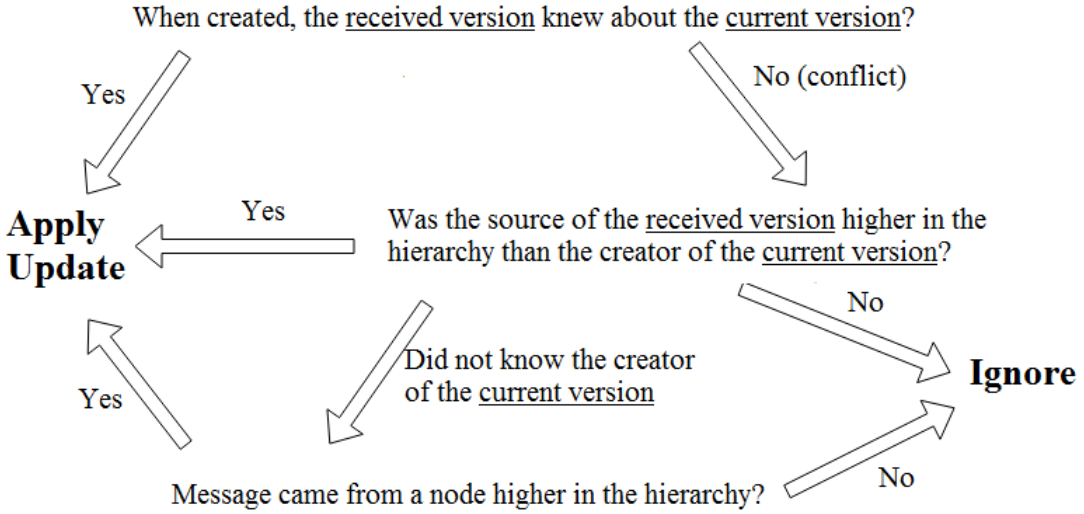

Figure 5.8: Diagram of the function used to decide whether or not to apply the received update.

Therefore, the main difference between the functions *Update* and *ReceivedKeyUpdate* is the latter checks if the received update must be applied or ignored. This check is done by operation *IsToApplyUpdate* that verifies if a conflict is detected, and if it is, which version wins. If no conflict is detected or if the received version wins the conflict, the operation *IsToApplyUpdate* will return *True*. Figure 5.8 explains how this operation decides if the update is to be applied. The first step of the function will use the received version vector to verify if the received version raises a conflict. If it does, it verifies which version was created higher in the hierarchy. If it was the received version, the update must be applied. Figure 5.9 shows the specification of *IsToApplyUpdate*. This operation checks five conditions to decide if the received version should be applied. From these five conditions, only the first must always hold. If the updated key contains a null version ($versionNum = -1$), the update is applied, otherwise, one of the other four conditions must be verified:

First clause - This clause verifies if this update was not previously received.

Re-propagation of updates can lead to situations where a node receives an update more than once, so a node uses the version vector to check if it is the first time it is receiving an update. Due to failures, a node can receive updates from nodes that was already removed (Figure 5.6). For that reason, a node would not be able to verify this condition because the source version vector entry was already removed.

In sum, this condition ensures that the update is only applied if it came from an offline node, or if is the first time the node is receiving it;

Second clause - This, and the third clause are used to detect a conflict. This verifies if the source of the message (that previously applied the updated) knew about the current version, i.e, if the *sourceVV* entry regarding the creator of the current version, shows that it knew the existence of the current version. In a situation where the creator of the current version is offline, this condition can't be verified, since the information about its creator is no longer being kept. The third clause is needed in those situations.

Third clause - This condition is similar to the last one, it verifies if the source knew when the current node applied the current version of the key. If it knew, it also means that it received the message with its propagation, so no conflict detected.

Fourth clause - This, and the fifth clause are used to verify if the received version wins the conflict. These clauses are only verified if a conflict is detected, i.e, second and third clause failed. This clause verifies if the source of the message was higher in the hierarchy than the creator of the current version, when it applied the new version. This information must be checked with the version vector that came with the message (*sourceVV*). If this condition is verified, the update must be applied.

Fifth clause - If the creator of the current version failed (is offline), the source of the message might not have its information, i.e, the entry of the creator of the current version might be null in the *sourceVV*. In these situations, the fourth clause cannot be verified and will return *False* (cannot verify who was higher in the hierarchy). For that reason, the received version will win the conflict if the message came from a node higher in the hierarchy.

Section 4.1 explained that nodes only communicate with their neighbors, and the handler function to resolve a conflict is based on the hierarchy level. Thus, one could wonder why is the forth clause needed. During normal execution, a node could never receive two distinct and concurrent version from its descendants, since those versions should always come from the same neighbor. An exception is when a new node N is added, and inherits some children of its parent P. If an inherited child C executes an update concurrently with the addition of N, and N executes a conflicting update, both updates would be sent to P. The new node N would send its update to its correct parent P. Node C (the inherited child) would incorrectly send its update to node P because it does not yet know its new parent, node N. On this situation node P would receive two conflicting updates from nodes below in the hierarchy, and although is is received from a node lower in the hierarchy, the one from N must always win the conflict. The fifth clause would not handle this situation independently of the messages order, the first to

```
IsToApplyUpdate(nodeId, vv, sourceId, sourceVV, sourceIsAbove, newVersion, currentVer) ≜
  LET
      creatorOfCurVers ≜ currentVer.nodeId
      versionNum ≜ currentVer.versionNumber
      logPosCurrentVersion ≜ GetLogPositionOfVersion(currentVer, nodeId)
  IN
      ∧ newVersion ≠ currentVer
      (* First clause *)
      ∧ ∨ vv[newVersion.nodeId] = NullVVEntry
        ∨ newVersion.versionNumber > vv[newVersion.nodeId].executedOperations
      ∧ ∨ versionNum = −1
      (* Second clause *)
        ∨ ∧ sourceVV[creatorOfCurrentVers] ≠ NullVVEntry
          ∧ sourceVV[creatorOfCurrentVers].executedOperations ≥ versionNum
      (* Third clause *)
        ∨ ∧ sourceVV[nodeId] ≠ NullVVEntry
          ∧ sourceVV[nodeId].executedOperations ≥ logPosCurrentVersion
      (* Fourth clause *)
        ∨ ∧ sourceVV[creatorOfCurrentVers] ≠ NullVVEntry
          ∧ creatorOfCurrentVers ∉ GetNodesAboveInHierarchy(sourceVV, sourceId)
      (* Fifth clause *)
        ∨ ∧ sourceVV[creatorOfCurrentVers] = NullVVEntry
          ∧ sourceIsAbove
```

Figure 5.9: Verification if update should be applied

be received would always in the conflict. Those situations are handled with the fourth clause.

5.6.2 Update unknown keys

When a node updates a key it does not know (Algorithm 4), one of two things will happen, a key will eventually be updated, or a key will be created. In either case, this process starts with the execution of the function *UpdateUnknownKey* (Algorithm 4). The goal of this function is to update a key, and if it does not exist, guarantee that that it is eventually created by all nodes interested in the update. To do this, besides sending the update, some information regarding the nodes interested in creating the key must also be propagated.All nodes that execute this operation will be on the set *nodesInterested* because each node that sends that message to its parent, adds its own identifier. The set *nodesInterested* is initially empty:

$$
\begin{aligned}
message \triangleq\ & [msgType \mapsto \text{"}new\_key\_or\_update\text{"}, \\
& sourceId \mapsto n, \\
& sourceVV \mapsto vv, \\
& keyId \mapsto k, \\
& value \mapsto value, \\
& nodesInterested \mapsto nodesInterested \cup \{n\}]
\end{aligned}
$$

If a node receives this message and contains the key k, it will execute the *Update* function explained in the last section (Section 5.6.1). It will use the received value to update the key, create a new version, and propagate the update to the interested nodes, the parent (if the node is not the root) and any interested child. In this case, the set *nodesInterested* is not used. Figure 5.10 shows an example of this situation, node W executes an update on a key it does not contain, and the update keeps being propagated until it reaches a node that contains the key (in this example, the node that received the message of a key it contains was the root, but it did not have to be). After applying the update, it is propagated to any interested node, in this case, only the child Z since no parent exists.

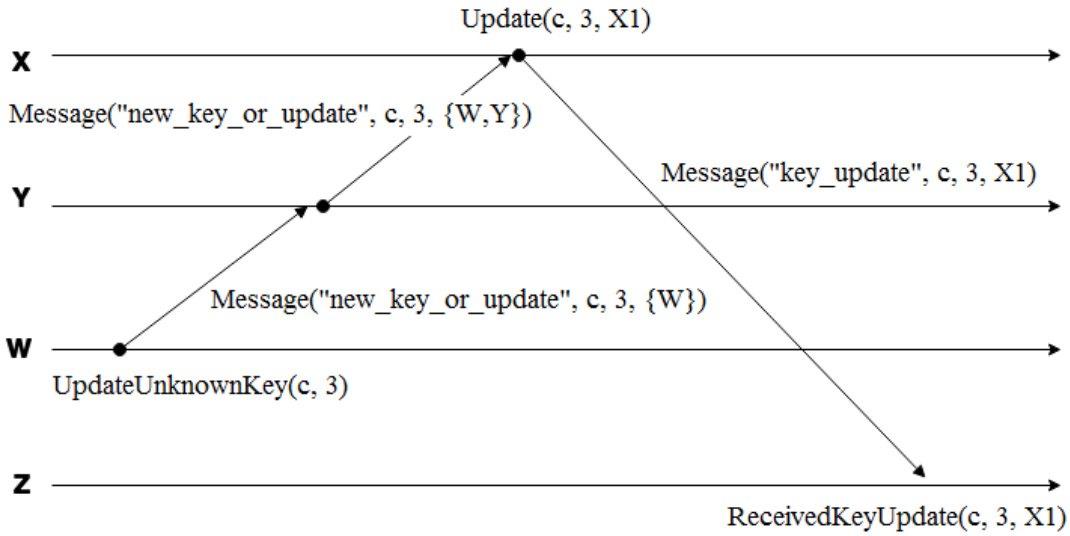

Figure 5.10: Update unknown existing key

On the other hand, Figure 5.11 shows an example where W tries to update a nonexistent key, and the root ends up receiving a message *new_key_or_update* for a key it does not contain. Because of that, the function *CreateKey* is executed. As explained in Algorithm 6, the root will create the *version* of the key, while the rest of the nodes will receive that version in a message. If no hierarchy changes happen, a node will be able to easily find a child in the received set *nodesInterested*, create the key, assign the identifier of that child to the variable *childInterested*, and finally propagate a message to that child:

$childInterested \triangleq vv[n].childrenId \cap message.nodesInterested$

$$
\begin{aligned}
new\_key \quad \triangleq \quad & [keyId \mapsto message.keyId \\
& value \mapsto message.value, \\
& childInterested \mapsto childInterested \\
& versionId \mapsto version]
\end{aligned}
$$

$$
\begin{aligned}
message \quad \triangleq \quad & [msgType \mapsto \text{"}new\_key\text{"}, \\
& sourceId \mapsto n, \\
& sourceVV \mapsto vv, \\
& keyId \mapsto message.keyId, \\
& value \mapsto message.value, \\
& version \mapsto version, \\
& nodesInterested \mapsto message.nodesInterested \setminus \{n\}]
\end{aligned}
$$

When designing and specifying these features of the protocol (update of an unknown key and creation of a key), three main decisions were made:

1. If the source of the message *new_key_or_update* is not a child, the message is ignored (it is known that this message is only sent to the parent).

2. The version that will eventually be created when the the function *Update* (Algorithm 2) or *CreateKey* (Algorithm 6) is executed will identify the node that executes one of those functions, instead of the node that started to propagate the message *new_key_or_update* (the node that originally executed the update).

3. Propagation of the set *nodesInterested*, instead of only propagating the identifier of the node that originally created the update. Since the version vector kept by nodes allows them to reconstruct the hierarchy, in some situations, the set *nodesInterested* propagated, instead of the single *nodeId*?

All three decisions were take due to the possibility of hierarchy changes during this process. The first situation can only happen if a node is added as parent of a node concurrently sending a message *new_key_or_update* to its old parent. Using the example of Figure 5.11, a situation where a node N is added between X and Y (child of X and parent of Y), at the same time as Y sends the message to X. The consequence would be its old parent receiving a message from a node that no longer was its child (X receiving from Y), and after changing its parent, re-propagating the message to the new parent (Y sending to N). If X did not ignore the message (the decision taken), the same message would end up being propagated and applied twice. It would not alter the end result, but the specification would generate unnecessary execution traces. Still to avoid unnecessary execution traces, if a node receives a message *new_key_or_update* of a key it contains (Figure 5.10), and the value on the message is the same as the current value of the key,

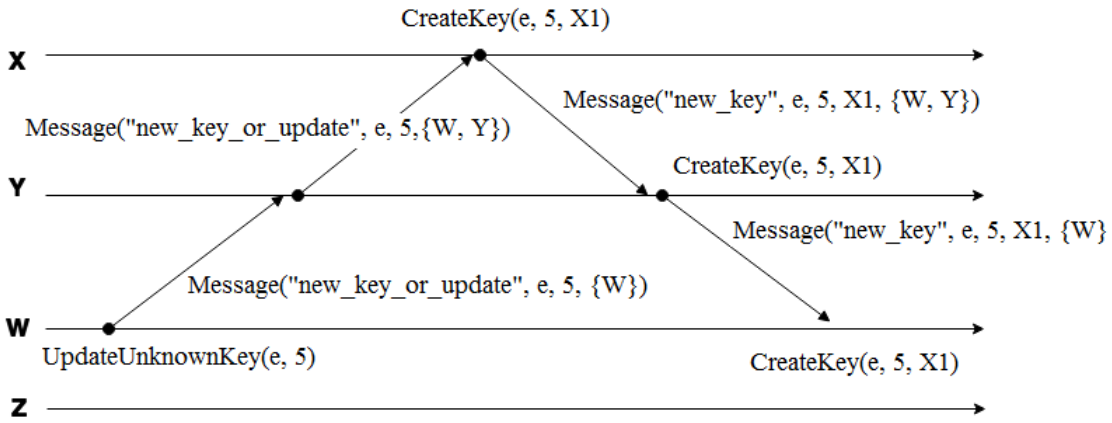

Figure 5.11: Update an nonexistent key

the message is also ignored. There is no need to create a new version while keeping the same value.

The second situation is needed to maintain causality. Using the example of Figure 5.11, lets consider a situation where, after propagating the message *new_key_or_update* with a version $W1$, but before X created the key e, W updated the key a with version $W2$. This would make Y receive and apply the update with version $W2$, before receiving the message *new_key* with the version $W1$, which would break causality. Notice that Y would receive the message with the version $W1$ before the update with the message $W2$, however, it would not be able to apply it since it did not know the key, it would only apply $W1$ when it created the key. In sum, if the second decision was not taken, in order to maintain causality, node W would have to stop executing operations, until it knew which operation its *new_key_or_update* message triggered.

When a node receives a message *new_key*, even if it's identifier is not in the set *nodesInterested*, it will create it, and keep propagating down the hierarchy. A node might send that message to a child not in the set, if it verifies that there are descendants of that child in *nodesInterested*. This situation can happen if a node is added concurrently during the process of key creation. Using the example of Figure 5.11, a situation where a node N was added between Y and W (child of Y and parent of W), at the same time as Y received a message *new_key* from X. After processing the message, Y would realize that although *nodesInterested* is not empty, none of its children were in the set (in the example, the only child would be N, and the set *nodesInterested* would only contain W). In this situation, the node would verify if any child contained descendants of that set (only one child can). Y would verify that N had a descendant in the set, so it would propagate the message to it. Since nodes can do this verification by reconstructing the hierarchy with their version vector, as long as the node contains a version vector entry of the node that originally

created the update, the set *nodesInterested* is not needed. So, the first reason to keep set *nodesInterested* is for hierarchies where nodes only keep version vector entries for nodes in a certain distance. Without this set, nodes could not verify which child was related to the node that originally create the update. Using the example of Figure 4.3, where nodes only keep version vector entries for nodes with maximum, two levels of difference, if Z updated a nonexistence key, after X received a message *new_key_or_update* and created the key, without set *nodesInterested*, it would not be able to verify to whom propagate the update. The second reason is in a situation where the node that originally create the update leaves the hierarchy before the message *new_key* has been fully propagated. Without this set, the other nodes interested in the key would not receive that message.

5.6.3 Adding nodes

The process of adding a node in the hierarchy starts with the root executing the *AddNode* function (Algorithm 7). This function will select a *nodeId* from the variable *offlineNodes*, and will retrieve the respective node's datastore, to verify if it is able to enter the hierarchy. The first step of the function is to checks which child of the current node, contains all keys of the new node. If any child contains, a possible parent for that node is found, and a message *new_node* is sent to it. Any node that receives that message will also execute *AddNode*, using the information of the message. If no possible parent was found, the function will select children of the current node whose keys are subsets of the keys of the new node. If the new node is added, it will inherit those children. As explained, if a node has an incompatible datastore with the nodes already in the hierarchy, it will not be added. To verify this situation, the last step of the function will verify which children have, at least, one key in common with the keys of the new node. If those children are different from the ones that will become its children, the node's datastore is invalid because it cannot have keys in common with siblings. As shown in Figure 5.3, this function starts with the *Next* state selecting a node from the set *offlineNodes*, and executing the function *AddNode* with the root.

If *parentFound* is true, the function *AddNode* will send a message to a child, and the new node will become *pending*. If due to invalid datastore, *error* is true, it will cancel the new node by adding it again in the set *offlineNodes*, and returning its status to *offline*. If *error* is false, the current node will execute a function to add the new node as its child. The necessity of the *pending* state, besides the *online* and *offline*, will be explained in the Section 5.6.5

$$newNodeParent \triangleq \{c \in vv[n].childrenId : newNodeKeys \subseteq GetKeysChildIsInterested(n, c)\}$$
$$parentFound \triangleq newNodeParent \neq \{\}$$

$newNodeChildren \triangleq \{c \in vv[n].childrenId : GetKeysChildIsInterested(n, c) \subseteq newNodeKeys\}$
$childrenWithKeysInCommon \triangleq$
$\quad \{c \in vv[n].childrenId : GetKeysChildIsInterested(n, c) \cap newNodeKeys / = \{\}\}$

$$
\begin{aligned}
error \triangleq \quad & \wedge \quad \vee \ parentFound = FALSE \\
& \qquad \vee \ newNodeChildren \neq \{\} \\
& \quad \wedge \quad newNodeChildren \neq childrenWithKeysInCommon
\end{aligned}
$$

$$
\begin{aligned}
message \quad \triangleq \quad & [msgType \mapsto \text{"new\_node"}, \\
& \ sourceId \mapsto n, \\
& \ sourceVV \mapsto vv, \\
& \ nodeId \mapsto newNodeId, \\
& \ keyIds \mapsto newNodeKeys]
\end{aligned}
$$

$configuration' =$
$\quad [configuration \ EXCEPT \ ![newNodeId].status.connectionStatus = \text{"pending"}]$

As explained, if the current node adds the new node as child, it will execute the Algorithm 8. This function executes the following actions:

1. Creates and adds a new entry to its version vector;
2. Stores information about which keys the new node contains;
3. Copies the keys which the new node is interested from the current node's datastore (variable *childInterested* of each key);
4. Copies the version vector entries of interest from the current node's version vector. Since the dastore is copied, the new node will be aware of the same operations that its parent is (the current node);
5. Propagates a message to its parent, informing it that a new node was added;
6. The new node sends a message to each child (each child in *newNodeChildren*) informing them to change their parent. After receiving the acknowledge of each child, the new node will apply its offline operations, and can start executing new operations;
7. Alters the status of the new node to *online*.

As mentioned earlier, a node can execute offline operations, but those operations will only be applied and propagated after the node is fully connected, i.e, after receiving the acknowledge of its children. When a node executes an operation, it increases its executed operations, i.e, if n executes a key update, it will do

$$vv[n].executedOperations = vv[n].executedOperations + 1$$

and will append the operation (the message *key_update*) to its log. As explained in Section 5.1, when a node goes offline, it keeps the number of its *executedOperations* in the variable *numOpOnline*, i.e:

$$configuration[n].status.numOpOnline = configuration[n].vv[n].executedOperations$$

By doing this, when a node becomes online, it is able to calculate how many operations were executed offline:

$$numOfflineOp \triangleq vv[n].executedOperations - configuration[n].status.numOpOnline$$

$$offlineOp \triangleq SubSeq(\ configuration[n].log,\ executedOperations - numOfflineOp + 1,\ executedOperations)$$

In order to apply those offline operations, they are removed from the node's log, and added to the node's messages. The value of *executedOperations* must be equal to the size of the log, so its value must be changed to the value of *numOpOnline*. After processing all those messages, the value of *executedOperations* will return to its original value. In order to force the *Next* state to execute the function *ProcessMessage(newNodeId)*, until all those messages are processed (all offline operations applied sequentially), an acknowledge *offline_operations* is added to the node's *waiting_acks* set. To stop forcing the *Next* state to execute the operation, a simple message with type *offline_operations* will be added after the messages with the offline operation, which will only be used to remove the acknowledge *offline_operations*. If the new node has children, it will wait for the acknowledges of those children to apply these offline operations (read those messages). For each child, the node will also add an acknowledge *ack_parent*, that will be removed with a similar message (will be explained later).

Section 5.4 presented one of the conditions to force the *Next* state to execute the function *ProcessMessage* of a specific node. Now, a new condition must be added. This new condition will be verified immediately after the last last child acknowledge is processed, and will stop after all the offline operations are applied:

$$\begin{aligned}\exists\ n \in onlineNodes :\ \exists\ ack \in configuration[n].waiting\_acks :\\ \lor\ ack.type = \text{"receiving\_package"}\\ \lor\ \land\ ack.type = \text{"offline\_operations"}\\ \land\ configuration[n].waiting\_acks \setminus \{ack\} = \{\}\end{aligned}$$

As mentioned, while the node is waiting for its children acknowledgments, it will be halted, not being allowed to execute any operation, nor process messages, other than the ones with those acknowledges.This is specified by adding some conditions in the function

ProcessMessage. This function is enabled if node n satisfies these conditions:

$$
\begin{array}{l}
\lor\ configuration[n].waiting\_acks = \{\} \\
\lor\ \exists\ ack \in configuration[n].waiting\_acks : \\
\qquad\qquad \lor\ \ ack.type = "receiving\_package" \\
\qquad\qquad \lor\ \ \land\ ack.type = "offline\_operations" \\
\qquad\qquad\qquad\ \ \land\ configuration[n].waiting\_acks \setminus \{ack\} = \{\} \\
\qquad\qquad \lor\ \ \land\ ack.type = "ack\_parent" \\
\qquad\qquad\qquad\ \ \land\ ContainsMsgAckParent(msgs[n])
\end{array}
$$

If when a node became offline, instead of keeping the log and storing the value of *executedOperations* in the variable *numOpOnline*, its log sequence was reset, and consequentially, the value of *executedOperations* became 0, the variable *numOpOnline* would not be needed. That way, if a node entered the hierarchy, its offline operations would be all the operations in its log, and *numOpOnline* would be unnecessary. There are two reason to not reset the log. The first is to guarantee that the causality property can be always verified. If a node N received a message *key_update* with a version $X3$ ($[nodeId \mapsto X, versionNumber \mapsto 3]$), went offline (reset the log), re-entered the hierarchy, and received another message *key_update* with version $X2$, the protocol would violate causality but would not be able to detect it because the update with version $X2$ would be the only operation in the log. The second reason is to guarantee that causality is not wrongfully detected. Lets consider an hierarchy with only two nodes, X and Y, where both nodes contain the same keys. If Y created two updates with versions $Y1$ and $Y2$ respectively, and then went offline (log reset), its next update would generate a version $Y1$ again. When X applied this last update, the protocol would wrongfully violate causality.

The forth step of *AddNewNodeAsChild* is to copy the relevant version vector entries to the new node version vector. In order to find out which are the relevant entries, a function *GetNodesRelatedTo* will be executed. This function will return a set with several *nodeId*, and only the entries of those nodes will be copied. Figure 5.12 shows the specification of the rest of the main steps of this function.

During the execution of the function *AddNewNodeAsChild*, the current node will add the operation with type *add_node_to_hierarchy* to its own log (*message_to_parent* of the specification in Figure 5.12), while the new node does not add any operation to its own log, it sends a message to each child but does not execute any operation (as explained, offline operations will eventually be added). Finally, when a node receives a message *add_node_to_hierarchy*, it gets the entry of the new node from the *sourceVV*, and adds it to its own version vector ($message.sourceVV[message.nodeId]$). After that, it adds the operation to its log and keeps propagating the message. If the message came from a child, it is propagated to its parent, if it came from the parent, it is propagated to all children. All nodes that will eventually receive the message are the ones related to the new node (as they contain keys in common). The first node to receive this message will

$$
\begin{array}{l}
newNode\_VVEntry \triangleq \\
\quad [nodeId \mapsto newNodeId, \\
\quad executedOperations \mapsto configuration[newNodeId].status.numOpOnline, \\
\quad parent \mapsto n, \\
\quad childrenId \mapsto newNodeChildren] \\
newNode\_waiting\_acks \triangleq \{[nodeId \mapsto newNodeId, type \mapsto \text{"offline\_operations"}]\} \\
\qquad\qquad \cup \{[nodeId \mapsto c, type \mapsto \text{"ack\_parent"}] : c \in newNodeChildren\}
\end{array}
$$

$$
\begin{array}{lll}
newNode\_datastore \triangleq & & \\
\quad [k \in KeyId \mapsto & IF & k \in newNodeKeys \\
& THEN & datastore[k] \\
& ELSE & NullDatastoreEntry]
\end{array}
$$

$$
\begin{array}{lll}
updated\_currentNode\_datastore \triangleq & & \\
\quad [k \in KeyId \mapsto & IF & k \in newNodeKeys \\
& THEN & [datastore[k] \ EXCEPT \ !.childInterested = newNodeId] \\
& ELSE & datastore[k]]
\end{array}
$$

$$
\begin{array}{lll}
message\_to\_parent & \triangleq & [msgType \mapsto \text{"add\_node\_to\_hierarchy"}, \\
& & \ sourceId \mapsto n, \\
& & \ sourceVV \mapsto configuration[n].vv, \\
& & \ nodeId \mapsto newNodeId]
\end{array}
$$

$$
\begin{array}{lll}
message\_to\_newNode\_children & \triangleq & [msgType \mapsto \text{"new\_parent"}, \\
& & \ parent \mapsto newNodeId, \\
& & \ sourceVV \mapsto configuration[newNodeId].vv, \\
& & \ nodeToRemove \mapsto NullNodeId]
\end{array}
$$

Figure 5.12: Specification of function *AddNewNodeAsChild*

be the parent of the current node, step 5. When the message *add_node_to_hierarchy* is added to the log, the node will replace any message *new_node* referring to that new added node, by a message with type *ignore_old_hierarchy_change*. Since the node is already in the hierarchy, this replacement avoids the re-propagation of the *new_node* message (functions that re-propagate messages will be explained later).

Figure 5.13 shows an example of a new node with id N being added in the hierarchy. Node N has the keys a, b in its datastore (as node Y), so it will eventually become a child of the node Y. The process will start with the root (node X) trying to add N in the hierarchy. It will realize that one of its children contains all keys of the new node, so it sends a message *new_node* to that child (node Y). When node Y receives and processes the message, it realizes that the new node must become its child, and the datastore of one of its children is contained in the new node datastore (datastore of node W). That means that after adding the new node, that child (node W) must become child of the new node. To do that, node Y will execute the operation *AddNewNodeAsChild* and will inform node W that must change its parent by sending it a message with type *new_parent*. If the node W realized that its new parent (node N) is missing messages, those messages would be re-propagated with the message *ack_parent*. This situation would happen if node W propagated messages to node Y when the new node N was already in the hierarchy.

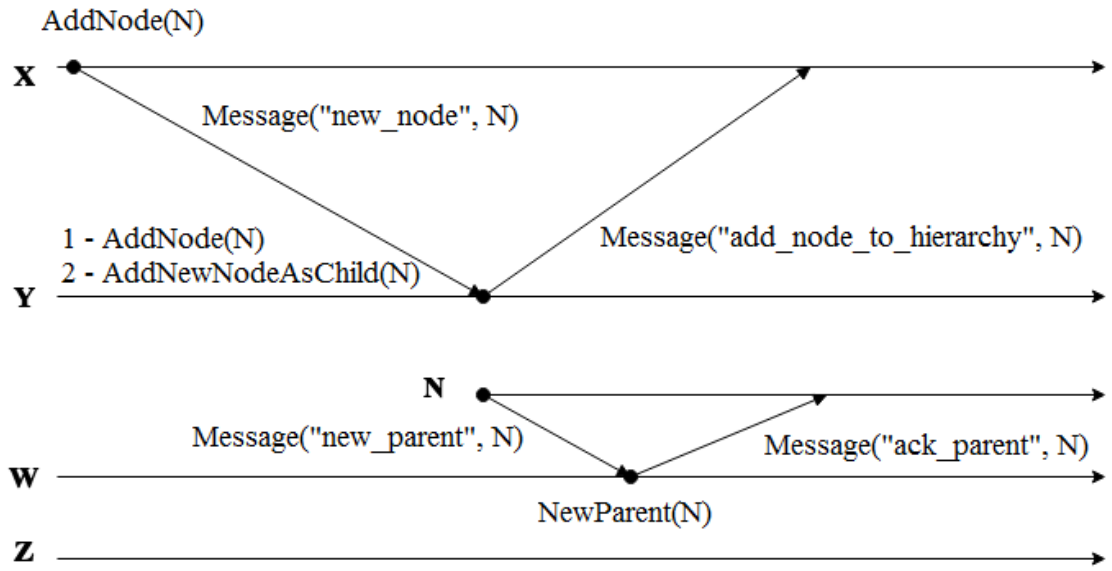

Figure 5.13: Insertion of a new node in the hierarchy

If node *N* had executed offline operations, those operations would be propagated after receiving the message *ack_parent*.

5.6.4 Removing nodes

Until now, five operations have been explained, *key_update*, *new_key_or_update*, *new_key*, *new_node*, and *add_node_to_hierarchy*. These operations, which correspond to the information sent in the propagated messages, have also been stored in the nodes logs. In order to verify causality, only the *key_update* operations would be needed to be kept in the logs, however there is another important reason to keep other messages in the logs, it allows nodes to re-propagate messages that its neighbors might be missing due to hierarchy changes. Therefore, the operations stored in the logs are the ones that might need to be re-propagated.When a node realizes that its neighbor is missing messages, those operations/messages are copied from the node's log to its neighbor's messages. This section will explain the last two operations that can be stored in the logs, *remove_child* and *remove_node*.

When a node chooses to leave the hierarchy, it must guarantee that its parent inherit its children, and the hierarchy stays correct, with the minimum amount of messages possible. To do this, the node will send a message to its parent, with the keys of its children, so that the parent can store that information, and send them future updates:

$childrenKeys \triangleq [nId \in vv[n].childrenId \mapsto GetKeysChildIsInterested(n, nId)]$

$$
\begin{aligned}
message \triangleq\ & [msgType \mapsto \text{"}remove\_child\text{"}, \\
& newChildren \mapsto childrenKeys, \\
& sourceVV \mapsto vv, \\
& nodeToRemove \mapsto n]
\end{aligned}
$$

The function *GetKeysChildIsInterested(n, nId)* returns a set with the keys of node n that contain the variable *childInterested* = *nId*. The operation must be stored in the node's log in case of a concurrent hierarchy change alters the node's parent. In that situation, the node will need to re-propagate the message to the new parent. When a node executes this operation, it stops executing operations/reading messages, it will wait for a message from its parent to become offline. This is done by adding an acknowledge *ack_remove* to its *waiting_acks* set, which will not allow the model checker to execute the function *ProcessMessage* unless the first message of the queue has type *ack_remove*. The only exception that allows a node with an acknowledge *ack_remove* to read a message other than on with type *ack_remove* is if it receives a message *new_parent*. If the node receives this message, it means that it sent the message *remove_child* to a node that no longer is its parent, and will ignore it. For that reason, the node must change its parent and re-propagate the message. A new condition was added to the specification of the *ProcessMessage* to allow node n to process a message *new_parent*:

$$
\begin{aligned}
& \lor\ configuration[n].waiting\_acks = \{\} \\
& \lor\ \exists\ ack \in configuration[n].waiting\_acks : \\
& \qquad \lor\ ack.type = \text{"}receiving\_package\text{"} \\
& \qquad \lor\ \land\ ack.type = \text{"}offline\_operations\text{"} \\
& \qquad\quad \land\ configuration[n].waiting\_acks \setminus \{ack\} = \{\} \\
& \qquad \lor\ \land\ ack.type = \text{"}ack\_parent\text{"} \\
& \qquad\quad \land\ ContainsMsgAckParent(msgs[n]) \\
& \qquad \lor\ \land\ ack.type = \text{"}ack\_remove\text{"} \\
& \qquad\quad \land\ \lor\ Head(msgs[n]).msgType = \text{"}ack\_remove\text{"} \\
& \qquad\qquad \lor\ ContainsMsgNewParent(msgs[n])
\end{aligned}
$$

When a node receives a message *remove_child* from a child, besides removing the information of that child, it will need to update its datastore (variable *childInterested*) to store the information about its new children, and will need to check which messages each new child is missing. Those missing messages are the ones it sent to the child that wants to leave the hierarchy, at the same time as that child sent it the message *remove_child*. Since the child is halt waiting for the *ack_remove*, it will not apply and propagate those messages to its children. The node will calculate which messages to re-propagate by using the *sourceVV* to verify how many messages its child knew about, when it sent the message *remove_child*:

$$
\begin{aligned}
&node \triangleq configuration[n] \\
&numMsgsNewChildrenKnow \triangleq message.sourceVV[n].executedOperations \\
&numExecutedOp \triangleq node.vv[n].executedOperations \\
&msgsMissing \triangleq SubSeq(node.log,\ numMsgsNewChildrenKnow + 1,\ numExecutedOp) \\
&messageToNewChildren \triangleq [msgType \mapsto \text{"new\_parent"}, \\
&\qquad\qquad parent \mapsto n, \\
&\qquad\qquad sourceVV \mapsto vv, \\
&\qquad\qquad nodeToRemove \mapsto message.nodeToRemove] \\
&msgsNewChildren \triangleq SelectSeqToChildren(\ msgsMissing, \\
&\qquad\qquad\qquad\qquad\qquad\qquad message.newChildren, \\
&\qquad\qquad\qquad\qquad\qquad\qquad MessageToNewChildren)
\end{aligned}
$$

The function *SelectSeqToChildren* will iterate through the missing messages, and decide which new child should receive each message. Not all children should receive the same messages, so depending on the type of message, this function will decide which child must receive it:

key_update - The node knows the keys of each new child, so depending on which key this operation updated, it will decide which child, if any, should receive the message;

new_key_or_update - This message is only propagated up the hierarchy, so it will not be sent to any new child;

add_node_to_hierarchy - This message is only propagated if two conditions are met. The first condition is to verify if the added node was not already removed. If it was, a message *remove_node* will also be in the queue of *msgsMissing*, and therefore, is not necessary to re-propagate this message. The second condition is to verify if the added node is higher in the hierarchy than the current node (*nodesAbove*, explained in Section 5.4, will be used). If the added node is a descendant of the current node, it means that the operation either came from the child that wants to leave the hierarchy, which means it also sent the message to its children (the new children of the current node), or the message came from another child, i.e, a sibling of the child leaving the hierarchy, which means the added node is not related to any of the new children. In either case the message does not need to be re-propagated to any child. If both conditions are satisfied, the message should be re-propagated to all new children, since they are all related to the added node;

remove_node - This message follows the same explanation of the *add_node_to_hierarchy*, it will only be re-propagated if the removed node was higher in the hierarchy, which means that the child leaving the hierarchy did not removed. This is verified by checking if *sourceVV*, the version vector of the child leaving the hierarchy, contains an entry for the removed node. If it does, its children (the new children of the

current node) will also have $sourceVV[msg.nodeId] \neq NullVVEntry$. An example of this *msg* is specified below as *msgToParent*;

remove_child - This message is only propagated to the parent, so it will not be sent to any new child;

new_key - If the current node sent this message to the child that wants to leave the hierarchy, it means the child was interested in creating the key. Now, the node will need to verify if any the new children is also interested. As explained in Section 5.6.2, this verification is done using the set *nodesInterested* of this message. It will check if any of the children, or its descendants, is in *nodesInterested*. If the message ends up being propagated to any child, the information of the datastore must be changed accordingly, the variable *childInterested* of this key must store that identifier of that child. For example, if the message *new_key*, which creates the key k, is re-propagated to a child c, the current node will execute $datastore[k].childInterested = c$;

new_node - If the current node x sent this message to the child (node c) that wants to leave the hierarchy, it means that node c contained all keys of the new node, so if the new node n was to enter the hierarchy, it would become a descendant of node c. Now without node c, node x must verify the situation of the new node n, which is still in a *pending* state, waiting to enter the hierarchy. The current node x will compare the new node's datastore (node n) with the datastores of the children it inherited from node c (children of c become children of x) to verify it the new node n can still enter the hierarchy. The specification of this process, instead of copying the code of *AddNode* to decide if the message should be re-propagated, adds the message in the beginning of the current node's message queue to force it to re-execute the function *AddNode* in the *Next* state. In order to force the protocol to process the message in the next *Next* state, an acknowledge $[nodeId \mapsto x, type \mapsto "repeat\_now"]$ is added to the node's *waiting_acks* set (current node x), and a condition is added to force the execution of the function *ProcesMessage*:

$$\begin{aligned}
\exists\ n \in onlineNodes :\quad \exists\ ack \in configuration[n].waiting_acks :\\
&\lor\ ack.type = "receiving_package"\\
&\lor\ \land\ ack.type = "offline_operations"\\
&\quad\ \land\ configuration[n].waiting_acks \setminus \{ack\} = \{\}\\
&\lor\ ack.type = "repeat_now"
\end{aligned}$$

A condition must also be added in the function *ProcessMessage* to enable a node to process a message when it contains the new acknowledge:

$\lor$ $configuration[n].waiting\_acks = \{\}$
$\lor$ $\exists$ $ack \in configuration[n].waiting\_acks$:
$\lor$ $ack.type = "receiving\_package"$
$\lor$ $\land$ $ack.type = "offline\_operations"$
$\land$ $configuration[n].waiting\_acks \setminus \{ack\} = \{\}$
$\lor$ $\land$ $ack.type = "ack\_parent"$
$\land$ $ContainsMsgAckParent(msgs[n])$
$\lor$ $ack.type = "repeat\_now"$

Obviously, if the acknowledge is not removed, the protocol will keep executing the function *ProcessMessage*, so in order to remove it, after the message *new_node*, a message $[msgType \mapsto "repeat\_now", sourceId \mapsto x]$ is added. The function that processes this message will remove the acknowledge.

As explained in Section 5.4, if a node needs to send several messages to a neighbor, it uses a package. Therefore, after selecting the messages to send to each new child, the ones set to receive more than one message will receive a package. The message *new_parent* will be sent to all new children and will be the last message of the pack.

Until now, the propagated messages appended the new message(s) at the end of the destination's message queue. Since the old child (node that wants to leave the hierarchy) is waiting for the message with the acknowledge, the message *msgsToOldChild* will be added in the beginning of its queue (it has higher priority). Finally, during the execution of the function *RemoveChild*, the current node will send a message with type *remove_node* to its parent informing it to remove the node (*msgToParent*), and will add the operation to its own log. When a node receives a message *remove_node*, it removes the correspondent entry of its own version vector, adds the operation to its log, and keeps propagating the message. If the message came from a child, it is propagated to its parent, if it came from the parent, it is propagated to all children. All nodes that will eventually receive the message are the ones related to the removed node (have keys in common).

$msgPS \triangleq [msgType \mapsto "package\_start", sourceId \mapsto n]$
$msgPE \triangleq [msgType \mapsto "package\_end", sourceId \mapsto n]$
$msgsToOldChild \triangleq [msgType \mapsto "ack\_remove", sourceId \mapsto n]$
$msgsToChildren \triangleq [c \in DOMAIN\ message.newChildren$
$IF \quad Len(msgsNewChildren[c]) > 1$
$THEN \quad Append(\langle\langle msgPS \rangle\rangle \circ msgsNewChildren[c], msgPE)$
$ELSE \quad msgsNewChildren[c]]$
$msgToParent \triangleq [msgType \mapsto "remove\_node",$
$nodeId \mapsto message.nodeToRemove,$
$sourceVV \mapsto vv,$
$sourceId \mapsto n]$

Now that both functions that generate the message *new_parent* were explained, the function *NewParent* that processes it, will be explained (Algorithm 11). This message was either generated by the function *RemoveChild* or *AddNewNodeAsChild*. The only difference on the message is the variable *nodeToRemove* that in the latter function will be null. The first step of function *NewParent* is to verify what caused its parent to change. If *nodeToRemove* is null, it means that its new parent just entered the hierarchy, if it is not null, it means that its old parent left the hierarchy. Depending on that, the node will either add a new entry to the version vector and propagate a message *add_node_to_hierarchy* to its children, or will remove an entry from the version vector and propagate a message *remove_node*.

$$
\begin{aligned}
msgToChildren \triangleq \quad [msgType \mapsto\ & IF\ message.nodeToRemove = NullNodeId,\\
& THEN\ "add\_node\_to\_hierarchy"\\
& ELSE\ "remove\_node"\\
nodeId \mapsto\ & IF\ message.nodeToRemove = NullNodeId\\
& THEN\ message.parent\\
& ELSE\ message.nodeToRemove\\
sourceVV \mapsto\ & vv,\\
sourceId \mapsto\ & n]
\end{aligned}
$$

Like the function *RemoveChild*, function *NewParent* will use the *sourceVV* to verify if its new parent is missing messages. This process is similar to the function *SelectSeqToChildren* used by *RemoveChild*. After retrieving the missing messages from the log, a filter is applied on those messages to verify which need to be re-propagated. The following function *SendToParent* will be executed on every message. All messages with type *new_key_or_update*, *remove_child* and *remove_node* will be re-propagated. Messages *new_key* and *new_node* are only propagated down the hierarchy, therefore will not be re-propagated to the parent. For each message *key_update* the node will use its parent version vector (*sourceVV*) to verify if the parent already applied the update. If the verification fails or cannot be done due to the parent not storing information about the creator of the update, the message will be re-propagated. Finally, messages *add_node_to_hierarchy* will be re-propagated if the parent does not contain information regarding the added node, and if the added node was not already removed. If it was, a message *remove_node* will also be in the sequence of *msgsMissing*, and therefore, is not necessary to re-propagate it. This is the function used by the current node *nId*, to verify each message the parent might be missing:

$$
\begin{aligned}
&SendToParent(message, sourceVV, nId) \triangleq\\
&\quad \lor\ message.msgType = "new\_key\_or\_update"\\
&\quad \lor\ message.msgType = "remove\_child"\\
&\quad \lor\ message.msgType = "remove\_node"
\end{aligned}
$$

$$
\begin{array}{llll}
\vee & \wedge & message.msgType = \text{"key\_update"} & \\
 & \wedge & \vee & sourceVV[message.version.nodeId] = NullVVEntry \\
 & & \vee & sourceVV[message.version.nodeId].executedOperations < \\
 & & & \qquad message.version.versionNumber \\
\vee & \wedge & message.msgType = \text{"add\_node\_to\_hierarchy"} & \\
 & \wedge & sourceVV[message.nodeId] = NullVVEntry & \\
 & \wedge & configuration[nId].vv[message.nodeId] \neq NullVVEntry &
\end{array}
$$

Along with those messages, a message with type *ack_parent* will be sent in a package of messages. This message will unlock the parent, which is halted, waiting for the acknowledgments of the new children to start executing operations. Since the parent is halted, this pack of messages will be placed in the beginning of its message queue.

Figure 5.14 shows an example of a node leaving the hierarchy. In this example, node *Y* decides to leave the hierarchy, so as explained, it sends a message to its parent asking permission to leave. When the parent (node *X*) receives the message *remove_child*, if it realized that it sent messages concurrently to node *Y* (and because of that, those messages were not processed), it would re-propagate those messages with the message *new_parent*. As explained, if the node *W* realized that its new parent, node *X*, was missing messages, those messages would be sent with the message *ack_parent*. Finally, when node *Y* receives the acknowledge from its parent, it becomes offline.

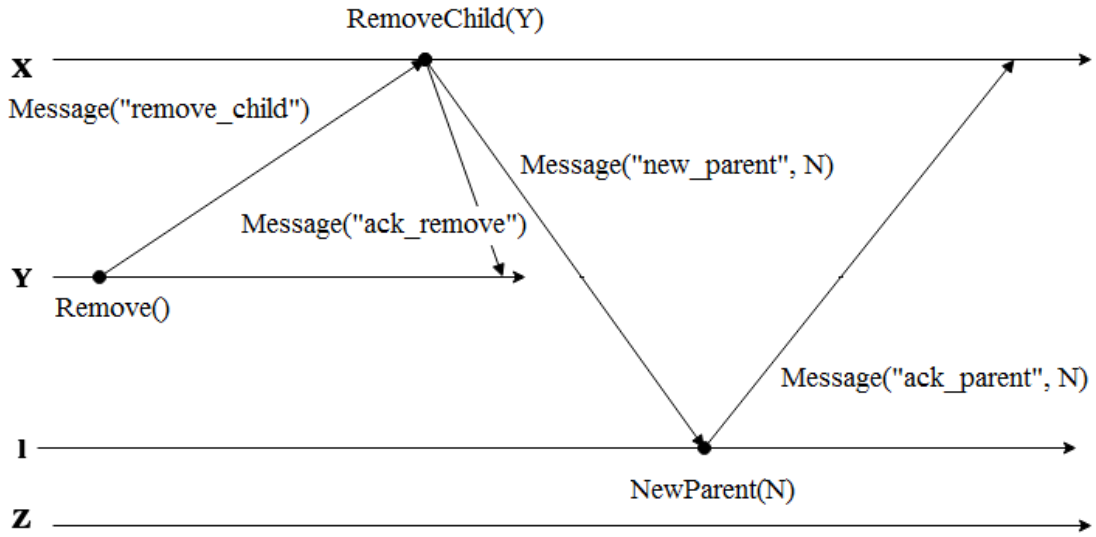

Figure 5.14: Removal of a node from the hierarchy

Until now, the variable *offlineNodes* was barely mentioned. This variable contains the *nodeId* of each offline node, however, the *nodeId* is not automatically added when a node becomes offline, i.e, the *nodeId* of a node is not added after it processes the message

ack_remove. A *nodeId* will only be added after all messages referring the node are processed (*new_node*, *add_node_to_hierarchy*, *remove_node*, and *new_parent*), and after all the other nodes remove it from theirs version vectors. This is done to avoid conflicts like allowing the root to execute the function *AddNode* for a node n that left the hierarchy, but still exists in root's version vector. This could happen if node n tried to re-enter the hierarchy before the root received the message *remove_node*. Since nodes retrieve and re-propagate missing messages from their logs, in order to avoid re-propagating old messages like *remove_node* regarding a node that left and rejoined the hierarchy, when the root executes the function *AddNode*, the specification removes all hierarchy changes of that node, from all nodes logs. In practice, it does not remove the entries from the log, it replaces them by messages with type *ignore_old_hierarchy_change*. This verification is done every time a message *ack_remove*, *remove_node*, and *new_parent* is processed. These are the messages that change the status of a node to *offline* and remove version vector entries.

$$
\begin{array}{l}
GetOfflineNodes \triangleq \\
\quad LET \\
\qquad nodesToCheck \triangleq DOMAIN\ configuration \setminus offlineNodes \\
\qquad newOff \triangleq \{n \in nodesToCheck : \\
\qquad\qquad \wedge\ configuration[n].status.connectionStatus = \text{"offline"} \\
\qquad\qquad \wedge\ \forall nId \in nodesToCheck \setminus \{n\} : \\
\qquad\qquad\qquad \wedge\ configuration[nId].vv[n] = NullVVEntry \\
\qquad\qquad\qquad \wedge\ \vee\ configuration[nId].status.connectionStatus = \text{"offline"} \\
\qquad\qquad\qquad\quad \vee\ NoMessagesAffectingNode(newMsgs[nId],\ n)\ \} \\
\quad IN \\
\qquad offlineNodes \cup newOff
\end{array}
$$

5.6.5 Failure handling

Section 5.5 explained that when a node n detects the failure of its neighbor *nId*, the specification adds an acknowledge $[nodeId \mapsto nId, type \mapsto \text{"correct\_hierarchy"}]$ in its *waiting_acks* set, and a message $[msgType \mapsto \text{"node\_failed"}, nodeId \mapsto nId, sourceId \mapsto n]$ in the beginning of its message queue. This will prevent the node from executing any operation other than reading the message with type *node_failed*, *correct_hierarchy*, or *ack_hierachy_corrected*, the messages used to correct the hierarchy. Figure 5.15 explains how *NodeFailed*, the function that reads a message *node_failed*, works. The first step is to verify the position of the failed node in the hierarchy. This function starts by verifying the relation between the current node and the one that failed. If the failed node f was the parent of current node n, node n will execute function *ParentFailDetected* to inform the parent of f of the failure of its child f. To do this, the function will execute several actions:

1. The current node n will use its version vector to verify who is its new parent,

i.e, the parent of f will inherit n as child. It will start by verifying if the parent of the failed node f (its old parent) is online. This verification simulates the use of a one bit message in a practical implementation, to verify the liveness of that node. In the TLA+ specification, it is only verified if the node is online, i.e $configuration[newParent].status.online = "online"$. If the new parent is offline, the node will keep going up in the hierarchy, until it finds an online node to become its new parent;

2. A message *node_failed* is sent to the new parent found in step 1, informing it that its child (node f) has failed;

3. Each node that step 1 found out to have failed is removed from the current node's (node n) version vector;

4. A pack of messages *remove_node* is sent to all children of the current node, informing them about the nodes that were removed from the hierarchy;

5. Those messages/actions are added in the current node's (node n) log;

6. The node will keep the acknowledge *node_failed* and will wait for messages *correct_hierarchy*, and *ack_hierachy_corrected* from its new parent to complete the process of correcting the hierarchy.

If the failed node f was below in the hierarchy than the current node n (f was descendant of n), similar with step 1 of *ParentFailDetected*, node n will verify which other nodes in the path failed. If no children of the current node n failed, and f is a descendant of one n's children, the function *InformProbableParent* will be executed. If a node was recently added in the hierarchy and ended up becoming the parent of the node that failed, the node that executed *ParentFailDetected* might not know it (situation *a)* of Figure 4.4, where node 4 detects the failure). The function *InformProbableParent* will just remove node f from the current node's (node n) version vector, propagate the message *remove_node* to its parent (parent of n), add the operation to n's log, and send the message *node_failed* to the parent of the node that failed. On the other hand, if n detects that one of its children failed, it will execute the function *HandleFailure*.

The function *HandleFailure* is executed by the current node n, and has five main steps:

1. Detect which nodes should become children of n due to its old child failure (which children must be inherited). Node n will start by retrieving the children of the child that failed f from its version vector. After that, it will verify if any of those children also failed by checking if they are online (just like step 1 of *ParentFailDetected*). For the ones that also failed, their children are retrieved. This process ends when node n has the information of the online children it must inherit due to failures, and a set *nodesFailed* with the *nodeId* of each descendant node that failed;

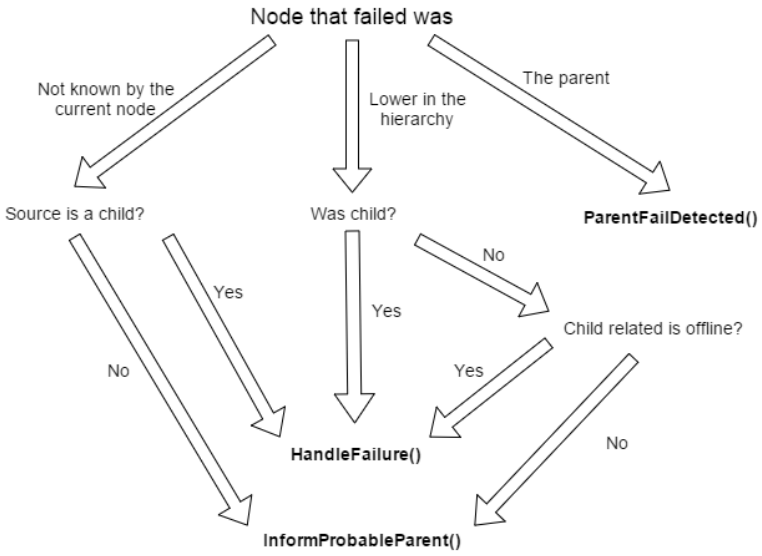

Figure 5.15: Function executed by the current node n after detecting a failure

2. Remove failed nodes from its version vector, and remove information about which keys the child that failed was interested in;

$$
\begin{array}{lll}
DS \triangleq [k \in KeyId \mapsto & IF & k \in GetKeysChildIsInterested(n, childFailed) \\
& THEN & [datastore[k] \ EXCEPT \ !.childInterested = NullNodeId] \\
& ELSE & datastore[k]]
\end{array}
$$

3. Find out which nodes that tried to enter the hierarchy, might be descendants of the child that failed, and are online. After that, the current node n checks which of those nodes should become its children, instead of the ones from step 1. This step is what allows node 1 in Figure 4.4, example *b)* to find out about node N. To do this, node n will verify its log, and retrieve any *new_node* messages regarding nodes with keys in common with the child that failed (message *new_node* contains the keys of the new node). These messages represent nodes that tried to enter the hierarchy, but the current node still does not know if they succeeded. If they had succeeded, the message *new_node* would have been replaced by one with type *ignore_old_hierarchy_change*. For those nodes, the current node verifies their liveness (if they are online), and checks their version vectors to determine if they are above in the hierarchy than any nodes of step 1;

4. Send a message *correct_hierarchy* to each new child. In the example of Figure 4.5, the message is sent to node 4. This message is the *message 1* from Figure 4.5, and will contain the version vector of the current node, plus the nodes that failed. They key ids do not need to be sent because the children know that the current node contains all their keys.

$$
\begin{aligned}
msgToNewChildren \triangleq\ & [msgType \mapsto \text{"}correct\_hierarchy\text{"}, \\
& sourceId \mapsto n, \\
& sourceVV \mapsto vv, \\
& nodesFailed \mapsto nodesFailed, \\
& keysId \mapsto \{\}]
\end{aligned}
$$

After sending the message *msgToNewChildren* to the new children, the node will halt until it receives a message *correct_hierarchy* from its new children.

5. Send to its parent a message *remove_node* for each node in *nodesFailed*. Those messages will also be added in the log.

After the execution of *HandleFailure*, nodes can receive a message *correct_hierarchy* from a node higher in the hierarchy, or a node they do not know. If a node does not know the source of the message, it must added it because the source will become its new parent (Figure 4.4.b, where node 4 receives the message from N). Function *ReceivedMsgCorrectHierarchy* that processes message *correct_hierarchy* starts by verifying which nodes that failed were lower in the hierarchy. Those failures must be handled by a different process, so the node adds a message *node_failed* for each of those nodes in the beginning of its message queue. This situation happens in the situation *b)* of Figure 4.4. The message received by N from 1 will have both nodes 2 and 3 in the set *nodesFailed*. The failure of node 3 must be handled separately. Using the same functions of *NewParent* (Section 5.6.3), the node will use the *sourceVV* to verify which messages the parent is missing, and along with a message *correct_hierarchy*, propagate them to the new parent. Since the parent is blocked waiting for this message, the message/pack of messages will be added in the beginning of its message queue. To the children, a message *remove_node* will be sent for each node that failed, and if the current node did not know the source of the received message, a message *add_node_to_hierarchy* will also be sent (those messages will be added in the log). The current node will also add an acknowledge *correct_hierarchy*, and will halt, waiting for a message *ack_hierachy_corrected* from the new parent.

When a node receives a message *correct_hierarchy* from a new child, it starts by storing the information about the keys the child is interested, and using the same functions of *RemoveChild*, it will use the received version vector (*sourceVV*) to verify which messages the child is missing. As mentioned, messages *new_node* might need to be re-executed (Section 5.6.4) if those new nodes still are on a "pending" state. Along with those missing messages, a message *ack_hierachy_corrected* will be propagated to the child, to remove the acknowledge blocking the child. The processing of this message ends the correction of the hierarchy, so the current node can remove the acknowledge *correct_hierarchy*, and start working normally.

$$
\begin{array}{lll}
message \triangleq & IF & sourceIsHigherInHierarchy \\
& THEN & [msgType \mapsto "correct\_hierarchy", \\
& & \ sourceId \mapsto n, \\
& & \ sourceVV \mapsto vv, \\
& & \ nodesFailed \mapsto nodesFailed, \\
& & \ GetNodeKeys(n)] \\
& ELSE & [msgType \mapsto "ack\_hierachy\_corrected", \\
& & \ sourceId \mapsto n, \\
& & \ sourceVV \mapsto vv]
\end{array}
$$

Figure 5.16: Message sent by the function *ReceivedMsgCorrectHierarchy*

As explained, functions *ReceivedMsgCorrectHierarchy*, *HandleFailure*, *InformProbableParent*, and *ParentFailDetected* remove the failed nodes from the node's version vector. In addition, nodes executing those functions also remove messages referring one of the failed nodes from their message queue (only messages changing the hierarchy). If several sibling nodes detect the failure of the parent, their new parent will receive a message *node_failed* several times. Since only one have to be processed, the others end up being removed from the message queue. All messages satisfying this condition are removed:

$$
\begin{array}{l}
IsHierarchyUpdateOfNode(message, fails) \triangleq \\
\quad \lor \ \land \ \lor \ message.msgType = "remove\_node" \\
\qquad \lor \ message.msgType = "remove\_node" \\
\qquad \lor \ message.msgType = "add\_node\_to\_hierarchy" \\
\qquad \lor \ message.msgType = "fails" \\
\quad \land \ message.nodeId \in fails \\
\quad \lor \ \land \ message.msgType = "new\_parent" \\
\quad \land \ message.parent \in fails
\end{array}
$$

Section 5.5 mentioned that when a node fails, its *nodeId* is added in the set *failedNodes*. A *nodeId* is removed from the set, after all its neighbors handle the failure, i.e, remove it from the *parent* and *childrenId* variable. The following function is used to verify which *nodeId* should be in the set, and is executed on every function that remove version vector entries, and alter the hierarchy:

$$
\begin{array}{l}
FailuresNotHandled(onlineNodes) \triangleq \\
\quad \{ nId \in failedNodes : \\
\qquad \exists n \in onlineNodes : \ \lor \ configuration[n].vv[n].parent = nId \\
\qquad\qquad\qquad\qquad\ \lor \ nId \in Configuration[n].vv[n].childrenId \}
\end{array}
$$

The full protocol specification can be found in the Appendix A. It is divided in two files:

1. The main file is the *MODULE NewProtocol*, it contains all the operations explained in this thesis, including the initialization, the *Next* state action, and the correctness properties.

2. The auxiliary file, *MODULE LocalOperations*, contains some auxiliary functions.

5.7 Operations costs

	Key Update	**Create Key**	**Update Remote Key**	**Add Node**
Best Case	0	0	1	$2c$
Worst Case	$h-1$	$2(h-1)$	$2(h-1)$	$Max(0,h-2)+2c$
	Remove Node	**Detect Fail**	**Failure Handling**	**Read Key**
Best Case	2 + c	0	3c	0
Worst Case	2 + c	c	3c	$2(h-1)$

Table 5.1: Messages required for each operation

This section ends the explanation of the protocol by presenting the operations's cost is terms of number of messages exchanged. Table 5.1 shows the best and worst case for each operation. The variable h used on some fields of the table stands for the height of the hierarchy, the variable c stands for the number of children of a specific node.

Starting with the operation *Key Update*, the best case is when the root updates a key that no other node is interested in, therefore, the update does not need to be propagated. The worst case is when a leaf updates a key because the update needs to be propagated to the root.

The best case of the operation *Create Key* is when the root independently creates a key, which means that none of its children will create it (no messages need to be propagated). The worst case is when a leaf updates a key that does not exist in the system. As explained in Section 5.6.2, a message "*new_key_or_update*" will be propagated to the root, and then a message "*new_key*" will be propagated back to the leaf.

The operation *Update Remote Key* is when a node updates a key it does not have, but exists in the system, i.e, at least the root has the key. The best case is when a child of the root updates a key that only the root contains. The worst case is when a leaf updates a key it does not have, the message "*new_key_or_update*" ends up being propagated to the root that after applying the update, needs to propagate it to another leaf that also has the key.

Operation *Add Node* is used to add a new node in the hierarchy. When a node enters the hierarchy it sends a message "*new_parent*" to each child it inherits, and waits for the

acknowledgment of each child to start executing properly. For that reason, in the best and worst case, there is a fixed cost of $2c$, where c stands for the number of children that the new node inherits (in the best case $c = 0$). The best case will be when the new node becomes child of the root because no message "*new_node*" will need to be propagated. In the worst case, the message "*new_node*" will be propagated down the hierarchy and the new node ends up inheriting several leafs as children.

Operation *Remove Node* as equal cost in the best or worst case. The node that wants to leave the hierarchy will send one message to its parent asking permission to leave, and then will wait for the parent acknowledgement. Besides those 2 messages, the parent will send a message "*new_parent*" to each child of the node that will leave the hierarchy (c stands for the number of those children).

As explained in Section 5.6.5, if a node detects the failure of a child, it handles it. This situation is the best case of the phase *Detect Fail*. On the other hand, if a node detects the failure of its parent (node p), it will send a message to the parent of p. The worst case is when all children of p detect the failure of its parent, which means that all those children will send a message to the parent of p. Here, c stands for the number of children of the node that failed (node p).

The phase *Failure Handling* starts when a node realizes that its child failed. As explained in Section 5.6.5, in order to handle the failure 3 messages are exchanged between the parent of the node that failed, and each child of the node that failed. Here, c also stands for the number of children of the node that failed.

Chapter 6

Protocol Verification

6.1 Properties

In order to validate the correctness of the protocol, we specified the correctness properties ensured by the protocol:

TypeInvariant - This property guarantees the type correctness of the specification, so every state reached by any possible execution has the correct type;

$$TypeInvariant \triangleq configuration \in [NodeId \rightarrow State]$$

Msgs_Log_Invariant - This property guarantees that all message queues only contain valid messages. In addition, this property also validates that the size of the log is equivalent to the number of executed operations of a node. The elements of the log are validated in the previous property (*TypeInvariant*).

$$\begin{aligned} &Msgs\_Log\_Invariant \triangleq \\ &\quad \land\ msgs \in [NodeId \rightarrow Seq(MsgKeyUpdate \cup ... \cup MsgCorrectHierarchy)] \\ &\quad \land\ \forall n \in NodeId : \\ &\qquad\qquad configuration[n].vv[n].executedOperations = Len(configuration[n].log) \end{aligned}$$

Causality - This property expresses that the causality between related operations is respected. Related operations are the ones executed locally by a node. A node breaks causality if it has executed an operation with a version previous to another operation executed before, e.g, if it applied an update with version *X10* after an update with version *X11*. This property is verified by checking the applied operations in the log of each node. The only operations that can break causality are the key updates/key creations, and since each *Next State* can only execute one of those operations at a time. For that reason, and as a matter of efficiency of the model checker, this property will only verify if the last operation of each log is breaking causality, i.e,

if the operation is a key update/key creation, it will verify if the applied version is previous to any other existent in the log;

$$
\begin{aligned}
&Causality \triangleq \\
&\quad \forall n \in NodeId : \\
&\qquad LET\ log \triangleq configuration[n].log \\
&\qquad\quad lastOp \triangleq Len(log) \\
&\qquad IN\ \ ContainsUpdateBreakingCausality(\ \ log[lastOp] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad SubSeq(log,\ 1,\ lastOp - 1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad) = FALSE
\end{aligned}
$$

$$
\begin{aligned}
&ContainsUpdateBreakingCausality(op,\ log) \triangleq \\
&\quad \wedge\ \ op.msgType = \text{"key\_update"} \\
&\quad \wedge\ \ \exists\ i \in 1..Len(seq) : IsKeyUpdateWithVersionAfter(log[i],\ op.version)
\end{aligned}
$$

$$
\begin{aligned}
&IsKeyUpdateWithVersionAfter(op,\ version) \triangleq \\
&\quad \wedge\ \ op.msgType = \text{"key\_update"} \\
&\quad \wedge\ \ op.version.nodeId = version.nodeId \\
&\quad \wedge\ \ op.version.versionNumber > version.versionNumber
\end{aligned}
$$

Convergence - This property expresses that all nodes that store a key will eventually have the same value and version. If this property is verified, it also means that no data was lost. Data could be lost if some updates were not propagated/re-propagated when needed. One way to verify this is to check the convergence of the values of the keys on the different nodes. If the values converge, then all messages arrived their destination and no data was lost. This property is verified by checking if all nodes that contain a key have the same value and version, as long as no message *key_update* of that key exist in the message queues. When the hierarchy is changing, i.e, when a message *new_parent*, or *ack_remove* is in queue to be read, this property cannot be ensured. When a node has a message *ack_remove* on its queue, it means that it is still online but it is not receiving messages anymore, which means that the propagated updates will not be applied by it. When a node has a message *new_parent* on its queue, it means that it might need to re-propagate messages to its new parent to guarantee that convergence is guaranteed. As previously explained, when nodes fail some messages are lost. In order to correct the hierarchy and re-propagate those missing messages, messages with type *node_failed* and *correct_hierarchy* are used. While those messages exist, and the set *failedNodes* is not empty, this property does not hold, because there are still nodes that might need to handle the failure and re-propagate messages. After that re-propagation the property can be verified. Those types of messages are verified with the function *IsTypeAffectingConvergence*, used in the specification of the *Convergence* property. In sum, if no messages affecting convergence exist in the system, keys are not being updated and must have converged:

$$
\begin{array}{l}
Convergence \triangleq \\
\quad LET \\
\qquad onlNs \triangleq \{n \in AllNodeId : configuration[n].status.online = TRUE\} \\
\qquad rootId \triangleq CHOOSE\ x \in onlNs : configuration[x].vv[x].parent = NullNodeId \\
\qquad rootDS \triangleq configuration[rootId].datastore \\
\quad IN \\
\qquad \lor\ failedNodes \neq \{\} \\
\qquad \lor\ \exists n \in onlNs : ContainsMsgAffectingConvergence(msgs[n]) \\
\qquad \lor\ \forall k \in KeyId : \quad \lor\ \exists n \in onlNs : ContainsMsgToUpdateKey(msgs[n], k) \\
\qquad\qquad\qquad\qquad\quad \lor\ \forall n \in onlNs\ : \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ configuration[n].datastore[k] = NullDatastoreEntry \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ configuration[n].datastore[k].versionId = \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad rootDS[k].versionId
\end{array}
$$

$$
\begin{array}{l}
ContainsMsgAffectingConvergence(nodeMsgs) \triangleq \\
\quad \exists\ i \in 1..Len(nodeMsgs) : \quad \lor\ nodeMsgs[i].msgType = \text{"new\_parent"} \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ nodeMsgs[i].msgType = \text{"ack\_remove"} \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ nodeMsgs[i].msgType = \text{"node\_failed"} \\
\qquad\qquad\qquad\qquad\qquad\quad \lor\ nodeMsgs[i].msgType = \text{"correct\_hierarchy"}
\end{array}
$$

$$
\begin{array}{l}
ContainsMsgToUpdateKey(nodeMsgs,\ k) \triangleq \\
\quad \exists\ i \in 1..Len(nodeMsgs) : \quad \land\ nodeMsgs[i].msgType = \text{"key\_update"} \\
\qquad\qquad\qquad\qquad\qquad\quad \land\ nodeMsgs[i].keyId = k
\end{array}
$$

ValidHierarchy - This property expresses that the hierarchy of online nodes must remain valid despite adding or removing nodes. An hierarchy is valid if four conditions are met:

1. only one root exists, i.e, there is only one online node with *parent* = *NullNodeId*;
2. if the node has a parent, the parent will have it in the *childrenId* set;
3. the node knows every key in the datastore of its children, i.e, the keys in a child's datastore are the same as the keys in the parent's datastore that have the variable *childInterested* = *child*;
4. and siblings have disjoint datastores. This is verified in condition 3 because *childInterested* only represents one child.

Due to the possibility of hierarchy changes, this property only holds when no messages with type *new_key*, *new_parent*, or *ack_remove* exists in the message queues. Otherwise, the hierarchy is still evolving and has not stabilized. When a message *new_key* exists, condition 3 does not hold, because the parent will have a key with the variable *childInterested* = *child* and the child has not created the key yet. If messages *new_parent* or *ack_remove* exist, the condition 2 does not hold, because the

parents of the nodes that contain those message already removed them from their *childrenId* set. Obviously, if any node fails, this condition does not hold until its neighbors (parent and children) handle the failure with the messages *node_failed*, *correct_hierarchy*, or *ack_hierachy_corrected*. In sum, if *failedNodes* is empty, and if no message *new_key*, *new_parent*, *ack_remove*, *node_failed* or *correct_hierarchy* exist, the condition is checked using the following function *IsProperHierarchy*.

$$
\begin{array}{l}
IsProperHierarchy(rootId) \triangleq \\
\quad \land\ IsSingleRoot(rootId) \\
\quad \land\ \forall\ nId \in NodeId : \\
\qquad LET \\
\qquad\quad parent \triangleq configuration[nId].vv[nId].parent \\
\qquad\quad childrenId \triangleq configuration[nId].vv[nId].childrenId \\
\qquad IN \\
\qquad\quad \land\ nId \neq parent \\
\qquad\quad \land\ nId \notin childrenId \\
\qquad\quad \land\ parent \notin childrenId \\
\qquad\quad \land\ \lor\ parent = NullNodeId \\
\qquad\qquad \lor\ \land parent \neq NullNodeId \\
\qquad\qquad\quad \land\ IsParentOfChild(parent, nId) \\
\qquad\quad \land\ \forall\ c \in\ childrenId : \\
\qquad\qquad\quad \land\ IsChildOfFather(nId, c) \\
\qquad\qquad\quad \land\ GetKeysChildIsInterested(nId, c) = GetNodeKeys(c)
\end{array}
$$

6.2 Model checker initialization

In order to initialize the model checker and verify the correctness of the specification, all constants must be initialized. As explained in Section 5.1, the specification uses two constants, *NodeId*, a set with all the node identifiers, and *KeyId* with all key identifiers. The initialization of these two constants must have more than one element, *NodeId* needs at least one online node and one offline node, and *KeyId* needs at least on key to be initialized, and one key to be created during the execution of the protocol (function *CreateKey*).

$initOfflineNodes \triangleq CHOOSE\ x \in SUBSET\ NodeId : x \neq NodeId \land Cardinality(x) = 1$

$newKeys \triangleq CHOOSE\ nk \in SUBSET\ KeyId : nk \neq KeyId \land Cardinality(nk) = 1$

For each different initialization, some conditions were added to the *Init* predicate to ignore initial states that generate equal execution traces, therefore not presenting different results.

In order to verify the specification, several initializations and executions were tested. These tests were executed in a server with 4 AMD Opteron 6272 2.1 GHz, and 64 GB of RAM. Those initializations vary in three aspects, the number of nodes used, the number

of keys used, and *MaxOperations*, the number of operations that each node can locally execute beside processing messages, i.e, if the number of *executedOperations* of a node is bigger than *MaxOperations*, the only operation that the node can execute is *ProcessMessage*. The executions vary in the operations that nodes can locally execute. If all operations are enabled, nodes can update a key they contain, update an unknown key, fail, leave the hierarchy (operation *Remove*), and while offline, besides updating their keys, can drop keys they contain. The root of the hierarchy can also add new nodes.

The results presented in each initialization To obtain the

2 nodes, 2 keys
$NodeId \triangleq \{x, y\}$
$KeyId \triangleq \{a, b\}$
The only optimization to be done during this initialization is to force the use of the same root on all initial states with the specification of $rootId \triangleq CHOOSE\ rId \in onlineNodeIds : TRUE$. With this condition, only one initial state will be generated.
This initialization was tested with 3 different executions, and $MaxOperations = 4$:

1. All operations are allowed except fail and leave the hierarchy.
   2.177.402 distinct states found, 0 states left on queue. The depth of the complete state graph search is 18.

2. All operations allowed except fail.
   2.563.455 distinct states found, 0 states left on queue. The depth of the complete state graph search is 17.

3. All operations.
   3.583.905 distinct states found, 0 states left on queue. The depth of the complete state graph search is 20.

3 nodes, 2 keys
$NodeId \triangleq \{x, y, z\}$
$KeyId \triangleq \{a, b\}$
The optimizations done during this initialization were to force the use of the same root on all initial states (like the previous initialization), and the use of the same offline node. With these conditions, only one initial state will be generated.
This initialization was tested with 3 different executions, and $MaxOperations = 3$:

1. All operations allowed except fail and leave the hierarchy. This execution had a duration of approximately 6 hours.
   225.193.182 distinct states found, 0 states left on queue. The depth of the complete state graph search is 31.

2. All operations allowed except fail. This execution had a duration of approximately 7 hours.

275.642.278 distinct states found, 0 states left on queue. The depth of the complete state graph search is 31.

3. All operations.
   Unfortunately, after 6 hours this execution ended with the error "No space left on device".
   278.340.231 distinct states found, 33.985.904 states left on queue.

3 nodes, 3 keys

$NodeId \triangleq \{x, y, w\}$

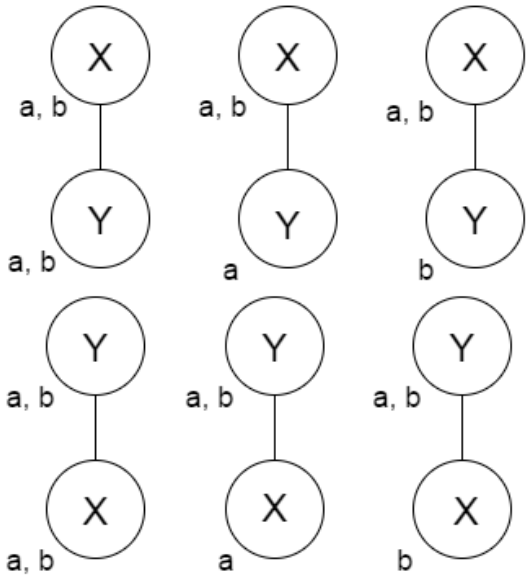

Figure 6.1: All initial states with two keys and two nodes

$KeyId \triangleq \{a, b, c\}$

If w was selected to be the offline node, c the new key not yet created, and no conditions were used to restrict the initial states (not even the use of the same root on all initial states), six different initial states would be generated. Those states are presented in Figure 6.1. It is easy to understand that the hierarchies with root Y are equal, and will generate equal executions traces than hierarchies with root X, so by forcing the root to be node X, the bottom hierarchies are ignored. The 3 hierarchies left, have two hierarchies where the leaf node only contains one key (a or b). The executions traces generated be each of those two hierarchies would only differ in the value of the key variable, besides that, the execution traces would be equal. If any property is violated in an execution trace of the hierarchy that uses a leaf with key a, it will also be violated in the hierarchy with key b, therefore, it is not necessary to verify both hierarchies since they will both validate the same execution traces. Thus, a condition was added to force the leaf to use a specific key, which only generates two hierarchies to be verified (the first and second of Figure 6.1):

$k \triangleq CHOOSE\ k \in initialKeys : TRUE$
$\forall n \in NodeId : k \in keys[n]$

Due to the amount of states that this initialization can generate, it was only used to verify the property *ValidHierarchy*. Using *MaxOperations* = 4, the execution used to verify this property allowed nodes to execute operations that can break the hierarchy, i.e, *AddNode*, *Remove*, and *Fail*, and allowed offline nodes to drop keys from their datastore. The results were:
Finished computing initial states: 2 distinct states generated.
12.468 distinct states found. The depth of the complete state graph search is 27.

Another execution with all operations allowed except fail, leave the hierarchy, and offline operations (*MaxOperations* = 3) ended after 4 hours with the error "No space left on device".
177.681.782 distinct states found, 125.030.785 states left on queue.

4 nodes, 3 keys
$NodeId \triangleq \{x, y, z, w\}$

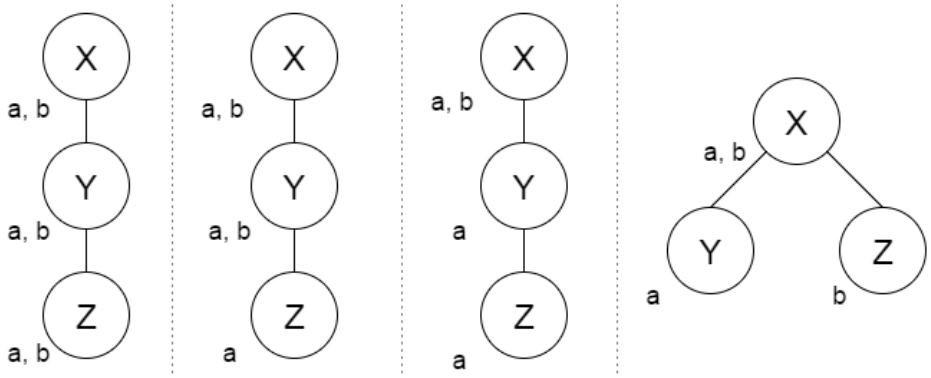

Figure 6.2: Initial states of interest with two keys and three nodes

$KeyId \triangleq \{a, b, c\}$
If w was selected to be the offline node, c the new key not yet created, and x forced to be the root, 12 different initial states would be generated. From those 12, only 4, represented in Figure 6.2, would generate different execution traces. This is done by forcing a specific key, and a specific node to be a child of the root. Unless the root has two children, all nodes will be forced to contain the specific key. This is done by adding the following condition after the initialization of the variable root (Figure 5.2):

$$
\begin{array}{l}
\wedge \ \mathit{LET} \\
\qquad \mathit{otherNodes} \triangleq \mathit{onlineNodeIds} \setminus \mathit{rootId} \\
\qquad \mathit{secNode} \triangleq \mathit{CHOOSE}\ \mathit{nId} \in \mathit{otherNodes} : \mathit{TRUE} \\
\qquad k \triangleq \mathit{CHOOSE}\ k \in \mathit{initialKeys} : \mathit{TRUE} \\
\quad \mathit{IN} \\
\qquad \wedge\ \mathit{secNode} \in \mathit{configuration}[\mathit{rootId}].\mathit{vv}[\mathit{rootId}].\mathit{childrenId} \\
\qquad \wedge\ k \in \mathit{GetNodeKeys}(\mathit{secNode}) \\
\qquad \wedge\ \mathit{IF} \quad \mathit{Cardinality}(\mathit{configuration}[\mathit{rootId}].\mathit{vv}[\mathit{rootId}].\mathit{childrenId}) = 1 \\
\qquad\ \ \mathit{THEN}\ \forall\ \mathit{nId} \in \mathit{otherNodes} \setminus \mathit{secNode} : k \in \mathit{GetNodeKeys}(\mathit{nId}) \\
\qquad\ \ \mathit{ELSE} \quad \mathit{TRUE}
\end{array}
$$

Like the previous initialization, due to the amount of states that this initialization can generate, it was only used to verify the property *ValidHierarchy*. Using *MaxOperations* = 4, the execution used to verify this property allowed nodes to execute operations that can break the hierarchy, i.e, *AddNode*, *Remove*, and *Fail*, and allowed offline nodes to drop keys from their datastore. The results were:
Finished computing initial states: 4 distinct states generated.
5.695.514 distinct states found. The depth of the complete state graph search is 60.

Another execution with all operations allowed except fail, leave the hierarchy, and offline operations (*MaxOperations* = 3) ended after 5 hours with the error "No space left on device".
340.748.716 distinct states found, 291.868.183 states left on queue.

TLA+ is most useful for finding bugs in algorithms written at a relatively high level of abstraction. With the presented results is easy to verify that beyond a few threads, the model checker will run out of memory or will take to long to verify a specification, however is usually enough to find concurrency errors. Due to this difficulty to verify the specification with many concurrent nodes/process, TLA+ is not suitable to get any information on how scalable an algorithm is [9]. As mentioned in Section 3.7, Cimbiosys was also specified with TLA+. Its verifications (can be found in the appendix of [28]) were similar with the ones presented in this sections, they used a maximum of 3 nodes and 2 items (equivalent to keys in ParTree). Regarding Pastry, which was also specified with TLA, the only information found about its verification was that they stopped the execution of the model checker after more than month without completing the verification.

Chapter 7

Conclusions

This thesis presents the design and specification of ParTree, a new protocol for partial replication, which could be used by clients that have limited resources. This protocol has shown the advantage of using a tree hierarchy when causal consistency is required. Furthermore it shows how causal consistency can be guaranteed in a protocol that allows partial replication, a dynamic hierarchy, an update anywhere model, and provides fault tolerance. During chapters 4 and 5, the protocol was presented by explaining how the different operations work with illustrative examples, as well as how those operations were specified in TLA+. The choice of specifying the protocol using TLA+ specification language and tools before its implementation, enabled the iteration and evolution of the protocol while ensuring the correctness properties. The development of ParTree had 3 main steps. The first was allowing nodes to execute updates as well as enter the hierarchy. The second step was allowing nodes to leave the hierarchy. This step introduced a lot of complexity due to the possibility of nodes concurrently entering and leaving the hierarchy, as well as executing updates. Several details were considered, and actions taken to guarantee that updates were not lost, and all nodes knew their correct position in the hierarchy. The third and final step was the introduction of fault tolerance. This last step was specially challenging due to the possibility of nodes failing while other nodes are concurrently entering and leaving the hierarchy.

7.1 Scalability

As mentioned in Section 2.2, scalability defines the ability of a system to grow, which is extremely important in distributed systems. For that reason, several decisions were taken to ensure that ParTree is scalable. Centralized systems, like those which use primary replicas (Section 2.1.2) and those that require agreement between large numbers

of nodes, usually have scalability problems. On the other hand, decentralized systems like peer-to-peer systems where all nodes have identical capabilities and responsibilities can be more scalable. Other property that improves the performance of a system is the replication of data to clients due to the reduction of communication needed (network bandwidth) when clients access data. Thus, the main characteristics of the system naturally improve its scalability, a peer-to-peer system that replicates data to clients. Besides those characteristics, several properties of the protocol ensure its scalability. When an update is executed, protocols that require coordination with nodes that do not replicate the updated objects, limit scalability. The developed protocol tackles this problem by organizing nodes in a hierarchy, and by maintaining minimal metadata. Other important detail about ParTree scalability is the fact that besides allowing the hierarchy to grow, nodes do not need to store information about all other nodes in their version vectors. As explained in Section 4.3, ParTree can limit the amount of metadata stored by each node, which could greatly increase as new nodes are added in the hierarchy. Finally, the fact that nodes do not require any central coordination mechanism to handle conflicting updates and node failures also contribute to the scalability of ParTree.

7.2 Related work

This section evaluates ParTree by comparing it with other partial replication protocols explained in Chapter 3.

Despite being very different, some properties of Pastry can be compared with ParTree. Starting with the practical use of both protocols, it can be verified that Pastry is not suitable for replication of data to clients because it forces nodes to replicate specific objects, instead of allowing them to select their objects of interest. As explained, the number of messages required to add a node in Pastry is $4 + 4l + 4$, which will probably be higher than the costs explained in Section 5.7. Since clients are likely to introduce high dynamism in the amount of available nodes, i.e, clients enter and leave the hierarchy with some frequency, that high cost is undesirable. Despite that, the fault tolerance of Pastry uses a mechanism similar to ParTree, it only keeps information of n closest nodes to reduce the amount of information stored by each node, therefore, improving scalability.

Cimbiosys [25, 28] is the most similar protocol with ParTree due to its use of a tree hierarchy and partial replication. Despite the use of a tree hierarchy, and opposed to ParTree who offers causal+ consistency, Cymbiosys only offers eventual consistency, each node may synchronize with any node they encounter, and therefore updates may be received by a replica in a different order than they were produced. As explained earlier, ParTree offers causal+ consistency by forcing nodes to only communicate with their connections (parent and children). As described in Section 3.7, the objects that each node in Cimbiosys stores is based on the objects values, while in ParTree it is based on the objects ids. The impact of this difference is that when an object is updated (in Cimbiosys), several nodes may be

forced to drop it from their datastores. Other distinctive aspect of these two protocols is the communication to propagate updates. ParTree uses a push style synchronization protocol that only sends one message that contains the update, and the version vector of the node after applying that update. On the other hand, nodes in Cimbiosys only synchronize with each other from time to time, and use a two-step, pull style synchronization protocol. This synchronization protocol uses two messages, the first is sent by a node requesting missing updates to another, and the second are the updates that node is missing (one more message than ParTree). This difference can increase the time needed until an update is fully propagated (ParTree is faster), however Cimbiosys does not need to store any information about which data other nodes are interested in. Since both protocols use an update anywhere model, both are vulnerable to conflicting versions. In Cimbiosys, after resolving the conflict, a new version superseding both conflicting versions is created, and all nodes interested must be informed. In ParTree, after resolving a conflict, only nodes that had applied the superseded version (version that lost the conflict) must be informed, which reduces the number of messages that must be propagated on these situations. To finalize the comparison of these two protocols, the metadata that each node store must be addressed. While ParTree keeps some extra information about the hierarchy to allow nodes to automatically handle conflicts and correct the hierarchy after a node failure, Cimbiosys keeps metadata about superseded versions to detect conflicting versions.

As described in Section 3.8, Perspecive has very different system designs than ParTree. Perspective is more suitable for small environments and is less scalable. It requires nodes to keep metadata regarding the *views* of all other nodes, while nodes in ParTree only keep information about the interest set of its children. In ParTree, when a node executes an update it will send a maximum of 2 messages and the update will eventually reach all nodes interested. On the other hand, in Perspecive, a node that executes an update will be responsible to propagate it to all interested nodes, and that operation will require 3 messages for each interested node. The advantage of Perspective is that it does not require the existence of a node with the full datastore, as opposed to ParTree in the root of the hierarchy. Finally, regarding the data stored by each node in Perspecive, as in Cimbiosys, nodes also store objects based on the values, instead of object ids.

As ParTree, Cimbiosys, and Perspecive, Coda [30] is another system that replicates data to clients by allowing them to cache data. Similar with ParTree, Coda also allows nodes to execute disconnected/offline operations, however it does not allow clients to communicate in a peer-to-peer model, they need to communicate with the server to share information.

PRACTI replication system [4] (described in Section 3.9) provides partial replication by allowing clients to select a subset of objects to replicate, supports arbitrary consistency levels, and topology independence. Starting with the information stored by each node,

both protocol require nodes to keep version vectors, however, while in PRACTI each node needs to keep a version vector entry for all other nodes, ParTree only require nodes to keep information about a subset of nodes. Regarding the subset of objects to replicate, PRACTI also have the disadvantage of needing to keep metadata about objects that they are not interested. When propagating updates, PRACTI uses a two-step updates propagation. It sends an ordered stream of invalidations, to send messages identifying what has changed, and unordered body messages, to send messages with the actual changes to the contents. Depending if a node is interested in the updated objects or not (even if the node is not interested in the updated object, it still needs to receive some information), the information sent will be different, so when two nodes communicate, the decision about which information is sent is made during the initial connection, where the receiving node specifies which information it is interested in. So in sum, the number of messages sent to propagate an update will be bigger than in ParTree, that only requires one message (for each node, two messages maximum) to propagate the update. In addition, during nodes synchronization, in ParTree no information regarding which objects the receiver is interested need to be exchanged because nodes store some metadata regarding data that its neighbors are interested in. Finally, regarding the consistency levels, while ParTree offers causal+ consistency, PRACTI can offer arbitrary consistency levels, however, if the implementation of PRACTI chooses to offer causal consistency, reads of objects might be blocked until a node receives some missing information, while in ParTree nodes can respond immediately.

7.3 Future work

As future work, several improvements can be done in the protocol, as well as in the specification. The specification can be improved by abstracting some properties of the protocol, which could lead to a significant decrease of the states generated, and would allow the execution of models with more keys and nodes. On the other hand, the protocol could be improved in some specific features like the implementation of a mechanism to reduce the log stored by each node (remove unnecessary entries), removing the need to keep a full datastore replica, allow siblings to have keys in common, and allow nodes to store keys that their parents do not store. This latter improvement can be done if nodes are allowed to forward update messages of keys they do not contain, without needing to apply them. Other future phase of this protocol will naturally be its practical implementation, which will allow the verification of its scalability.

Bibliography

[1] P. S. Almeida, A. Shoker, and C. Baquero. "Efficient State-based CRDTs by Delta-Mutation". In: *CoRR* abs/1410.2803 (2014). URL: http://arxiv.org/abs/1410.2803.

[2] S. Almeida, J. Leitao, and L. Rodrigues. "ChainReaction: a causal+ consistent datastore based on chain replication". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM. 2013, pp. 85–98. DOI: 10.1145/2465351.2465361.

[3] B. Batson and L. Lamport. "High-level specifications: Lessons from industry". In: *Formal methods for components and objects*. Springer. 2003, pp. 242–261.

[4] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. "PRACTI Replication". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006, pp. 5–5. URL: http://dl.acm.org/citation.cfm?id=1267680.1267685.

[5] W. J. Bolosky, J. R. Douceur, and J. Howell. "The Farsite Project: A Retrospective". In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 17–26. ISSN: 0163-5980. DOI: 10.1145/1243418.1243422. URL: http://doi.acm.org/10.1145/1243418.1243422.

[6] E. A. Brewer. "Towards robust distributed systems". In: *PODC*. 2000, p. 7. DOI: 10.1145/343477.343502.

[7] I. Briquemont. "Optimising Client-side Geo-replication with Partially Replicated Data Structures". MA thesis. UCL, 2014.

[8] G. Cabri, A. Corradi, and F. Zambonelli. "Experience of adaptive replication in distributed file systems". In: *EUROMICRO Conference*. IEEE Computer Society. 1996, pp. 0459–0459. DOI: 10.1109/EURMIC.1996.546470.

[9] *Check a concurrent algoithm using a large number of concurrent threads*. URL: https://groups.google.com/forum/#!msg/tlaplus/dxL-D_Knz24/gP79VBnURXwJ (visited on 09/19/2015).

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. "PNUTS: Yahoo!'s hosted data serving platform". In: *Proceedings of the VLDB Endowment* 1.2 (2008), pp. 1277–1288. DOI: 10.14778/1454159.1454167.

[11] T. Crain and M. Shapiro. "Designing a causally consistent protocol for geo-distributed partial replication". In: *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*. ACM. 2015, p. 6.

[12] *Eventual consistency*. URL: http://en.wikipedia.org/wiki/Eventual_consistency (visited on 01/02/2015).

[13] S. Gilbert and N. Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601.

[14] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens. "Dynamic and adaptive replication for large-scale reliable multi-agent systems". In: *Software engineering for large-scale multi-agent systems*. Springer, 2003, pp. 182–198. DOI: 10.1007/3-540-35828-5_12.

[15] H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. "Data replication strategies in grid environments". In: *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*. IEEE. 2002, pp. 378–383.

[16] L. Lamport. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[17] L. Lamport. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103. ISSN: 0178-2770. DOI: 10.1007/s00446-006-0005-x. URL: http://dx.doi.org/10.1007/s00446-006-0005-x.

[18] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu. *The wildfire challenge problem*. 2001.

[19] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. "Don't settle for eventual: scalable causal consistency for wide-area storage with COPS". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 401–416. DOI: 10.1145/2043556.2043593.

[20] T. Lu, S. Merz, and C. Weidenbach. *Towards verification of the Pastry protocol using TLA+*. Springer, 2011.

[21] S. Merz, T. Lu, and C. Weidenbach. "Towards Verification of the Pastry Protocol using TLA+". In: *31st IFIP International Conference on Formal Techniques for Networked and Distributed Systems*. Ed. by R. Bruni and J. Dingel. Vol. 6722. FMOODS/FORTE 2011. Reykjavik, Iceland, June 2011. URL: https://hal.inria.fr/inria-00593523.

[22] C. Newcombe. "Why Amazon Chose TLA+". In: *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 2014, pp. 25–39. DOI: 10.1007/978-3-662-43652-3_3.

[23] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. *Use of Formal Methods at Amazon Web Services*. Tech. rep. 2013.

[24] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. "Flexible Update Propagation for Weakly Consistent Replication". In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 288–301. ISSN: 0163-5980. DOI: 10.1145/269005.266711.

[25] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. "Cimbiosys: A platform for content-based partial replication". In: *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*. 2009, pp. 261–276.

[26] P. Regnier, G. Lima, and A. Andrade. "A TLA+ formal specification and verification of a new real-time communication protocol". In: *Electronic Notes in Theoretical Computer Science* 240 (2009), pp. 221–238.

[27] R. van Renesse and F. B. Schneider. "Chain Replication for Supporting High Throughput and Availability." In: *OSDI*. Vol. 4. 2004, pp. 91–104.

[28] T. L. Rodeheffer. *CIMSync Protocol Specification*. Tech. rep. MSR-TR-2009-75. 2009. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=81313.

[29] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. "Perspective: Semantic data management for the home". In: *FAST*. Vol. 9. 2009.

[30] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. "Coda: A highly available file system for a distributed workstation environment". In: *Computers, IEEE Transactions on* 39.4 (1990), pp. 447–459.

[31] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, et al. *A comprehensive study of convergent and commutative replicated data types*. Tech. rep. 2011.

[32] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. "Proactive Replication for Data Durability". In: *IPTPS*. IPTPS '06. 2006.

[33] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. "Partial replication in the database state machine". In: *Network Computing and Applications, 2001. NCA 2001. IEEE International Symposium on*. IEEE. 2001, pp. 298–309.

[34] *What is TLA?* URL: http://research.microsoft.com/en-us/um/people/lamport/tla/tla-intro.html (visited on 08/25/2015).

APPENDIX

TLA+ Specification

MODULE *NewProtocol*

EXTENDS *Naturals*, *Integers*, *FiniteSets*, *TLC*, *Sequences*, *LocalOperations*

CONSTANTS
NodeId,
KeyId,
MaxOperations

$NullNodeId \triangleq \text{CHOOSE } x : x \notin NodeId$
$NodeIdOrNull \triangleq NodeId \cup \{NullNodeId\}$

$NullKeyId \triangleq \text{CHOOSE } x : x \notin KeyId$
$KeyIdOrNull \triangleq KeyId \cup \{NullKeyId\}$

$Ack \triangleq$
$[nodeId : NodeId,$
$type : \text{STRING}]$ *ack_parent*, *ack_remove*, *receiving_package*, *offline_operations*, *repeat_now*

$NullVersion \triangleq [nodeId \mapsto NullNodeId, versionNumber \mapsto -1]$
$VersionId \triangleq$
$[nodeId : NodeIdOrNull,$
$versionNumber : Int]$

$NullDatastoreEntry \triangleq [keyId \mapsto NullKeyId]$
$NullKey \triangleq [keyId : \{NullKeyId\}]$
$Key \triangleq$
$[keyId : KeyId,$
$value : Int,$
$childInterested : NodeIdOrNull,$
$versionId : VersionId]$

$NullVVEntry \triangleq [nodeId \mapsto NullNodeId]$
$NullNodeInformation \triangleq [nodeId : \{NullNodeId\}]$
$NodeInformation \triangleq$ Information of a each node in *VV*
$[nodeId : NodeId,$
$executedOperations : Nat,$
$parent : NodeIdOrNull,$
$childrenId : \text{SUBSET } NodeId]$

$Status \triangleq [connectionStatus : \text{STRING},$ online, pending, *offline*
Number of operations executed by the node, before becaming *offline*. Used to calculate *offline* operations
$numOpOnline : Nat]$

Most of the messages has a 'sourceId' and a 'sourceVV'. This is the information of the node that sent the message, not the node that created original message. It changes on every step of the propagation. After receiving a message, a node uses the $sourceVV$ to update its own VV

$MsgKeyUpdate \triangleq$
$[msgType : \text{STRING}, \quad \text{"key\_update"}$
$sourceId \quad : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation],$
$keyId : KeyId, \quad$ Key that was changed
$newValue : Int, \quad$ New value of the key
$version : VersionId]$ New Version of the key

$MsgUpdateUnknownKey \triangleq$
$[msgType : \text{STRING}, \quad \text{"new\_key\_or\_update"}$
$nodesInterested : \text{SUBSET } NodeId,$
$keyId : KeyId,$
$value : Int, \quad$ New value of the key
$sourceId \quad : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation]]$

$MsgNewKey \triangleq \quad \text{"new\_key"}$
$[msgType : \text{STRING},$
$nodesInterested : \text{SUBSET } NodeId, \quad$ All the nodes that must create the key
$keyId : KeyId, \quad$ Id of the key
$value : Int,$
$version : VersionId,$
$sourceId \quad : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation]]$

$MsgNewNode \triangleq$
$[msgType : \text{STRING}, \quad \text{"new\_node"}$
$nodeId : NodeId, \quad$ Id of the new node to be created
$keyIds \quad : \text{SUBSET } KeyId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation],$
$sourceId \quad : NodeId]$

$MsgHierarchyChange \triangleq$
$[msgType : \text{STRING}, \quad \text{"add\_node\_to\_hierarchy"} , \text{"remove\_node"}$
$nodeId : NodeId, \quad$ Id of the node to be added/removed
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation],$
$sourceId \quad : NodeId]$

$MsgNewParent \triangleq$
$[msgType : \text{STRING},$ "new_parent"
$parent : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation],$
If a node must change its parent because its previous parent was removed from the hierarchy, this will have a value, else will be null
$nodeToRemove : NodeIdOrNull]$

$MsgAckNewParent \triangleq$
$[msgType : \text{STRING},$ "ack_parent" , "ack_hierachy_corrected"
$sourceId : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation]]$

"package_start" , "package_end" , "ack_remove"
$MsgPackegesAndRemoveAck \triangleq [msgType : \text{STRING}, sourceId : NodeId]$

$MsgRemoveChild \triangleq$
$[msgType : \text{STRING},$ "remove_child"
$nodeToRemove : NodeId,$ Id of the node that wants to be removed
Children of the node that want to be removed. The node that receives this message must add this children
$newChildren : [NodeId \to \text{SUBSET } KeyId],$
$sourceId : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation]]$

$LogAutoRemove \triangleq [msgType : \text{STRING}]$ "offline" , "ignore_old_hierarchy_change" , "datastore_change"

$MsgNodeFailed \triangleq$
$[msgType : \text{STRING},$ "node_failed"
$nodeId : NodeId,$
$sourceId : NodeId]$

$MsgCorrectHierarchy \triangleq$
$[msgType : \text{STRING},$ "correct_hierarchy"
$nodesFailed : \text{SUBSET } NodeId,$
$keysId : \text{SUBSET } KeyId,$
$sourceId : NodeId,$
$sourceVV : [NodeId \to NodeInformation \cup NullNodeInformation]]$

$State \triangleq$
$[nodeId : NodeId,$
$datastore : [KeyId \rightarrow Key \cup NullKey],$
$vv : [NodeId \rightarrow NodeInformation \cup NullNodeInformation],$
$waiting\_acks : \text{SUBSET } Ack,$
$status : Status,$
$log : Seq(MsgKeyUpdate \cup MsgUpdateUnknownKey \cup MsgNewKey \cup MsgNewNode$
$\cup MsgRemoveChild \cup MsgHierarchyChange$
$\cup LogAutoRemove)]$

Verify Hierarchy

Get all the *keyIds* of the keys in the node *datastore*

$IsProperHierarchy(rootId) \triangleq$
$\land IsSingleRoot(rootId, NullNodeId)$
$\land failedNodes \cap offlineNodes = \{\}$
$\land \forall id \in NodeId :$
LET
$node \triangleq configuration[id]$
$parentId \triangleq node.vv[id].parent$
$childrenId \triangleq node.vv[id].childrenId$
$datastore \triangleq node.datastore$
IN
$\land id \neq parentId$
$\land id \notin childrenId$
$\land parentId \notin childrenId$
$\land \exists k \in KeyId : datastore[k] \neq NullDatastoreEntry$
Verify if the parent of the node assumes it as its child.
$\land parentId = NullNodeId \lor \land parentId \neq NullNodeId$
$\land IsParentOfChild(parentId, id)$
$\land \forall child \in childrenId :$
$\land IsChildOfFather(id, child)$
$\land GetNodeKeys(child,$
$NullDatastoreEntry) = GetKeysChildIsInterested(id,$
$child,$
$KeyId,$
$NullDatastoreEntry)$

$ValidHierarchy \triangleq$
LET
$onlineNodes \triangleq \{n \in NodeId : configuration[n].status.connectionStatus = \text{“online”}\}$

$rootId \triangleq$ CHOOSE $rId \in onlineNodes : configuration[rId].vv[rId].parent = NullNodeId$

IN

$\lor\ failedNodes \neq \{\}$

$\lor\ \exists\, n \in onlineNodes :\ ContainsMsgChangingHier(msgs[n])$

$\lor\ IsProperHierarchy(rootId)$

Init System

$InitKey(keyId,\ childInterested) \triangleq$

$[keyId \mapsto keyId,$
$\ value \mapsto 0,$
$\ childInterested \mapsto childInterested,$
$\ versionId \mapsto NullVersion]$

$InitVV(nodeId,\ allChildren,\ allParent,\ allKeys,\ offNodes) \triangleq$

LET

$nodeIsOffline \triangleq nodeId \in offNodes$

$nodeInformation \triangleq [nodeId \mapsto nodeId,$
$executedOperations \mapsto 0,$
$parent \mapsto$ IF $nodeIsOffline$
THEN $NullNodeId$
ELSE $allParent[nodeId],$
$childrenId \mapsto$ IF $nodeIsOffline$
THEN $\{\}$
ELSE $allChildren[nodeId]]$

$onlineNodes \triangleq NodeId \setminus offNodes$

$nodesRelated \triangleq$ IF $nodeIsOffline$
THEN $\{\}$
ELSE $\{n \in onlineNodes : allKeys[nodeId] \cap allKeys[n] \neq \{\}\}$

IN

$[n \in NodeId \mapsto$ IF $n = nodeId$
THEN $nodeInformation$

ELSE IF $n \in nodesRelated$
THEN $[nodeId \mapsto n,$
$executedOperations \mapsto 0,$
$parent \mapsto allParent[n],$
$childrenId \mapsto allChildren[n]]$

ELSE $NullVVEntry]$

$InitNode(nId,\ \ nodeVV,\ datastore,\ connectionStatus) \triangleq$

$[nodeId \mapsto nId,$

```
    datastore ↦  datastore,
    vv ↦ nodeVV,
    waiting_acks ↦ {},
    status ↦ [connectionStatus ↦ connectionStatus, numOpOnline ↦ 0],
    log ↦ ⟨⟩]

InitState(children, keys, parent, initialKeys, initOfflineNodes) ≜
  [nodeId ∈ NodeId ↦
    LET
      nodeIsOffline ≜ nodeId ∈ initOfflineNodes

      nodeKeys ≜ IF nodeIsOffline
                 THEN initialKeys
                 ELSE keys[nodeId]

      datastore ≜
        [k ∈ KeyId ↦
          IF k ∉ nodeKeys
          THEN NullDatastoreEntry
          ELSE IF nodeIsOffline
          THEN InitKey(k, NullNodeId)
          ELSE LET
                 CIS ≜ {cId  ∈ children[nodeId] : k ∈ keys[cId]}
                 CI ≜ IF CIS = {}
                      THEN NullNodeId
                      ELSE CHOOSE cId ∈ CIS : TRUE
               IN
                 InitKey(k, CI)]

      vv ≜ InitVV(nodeId, children, parent, keys, initOfflineNodes)

    IN
      IF nodeIsOffline
      THEN InitNode(nodeId, vv, datastore, "offline")
      ELSE InitNode(nodeId, vv, datastore, "online")]

------------------------------------------------------------------------

IgnoreMessage(node) ≜
  LET
    new_config ≜ [configuration EXCEPT ![node.nodeId] = node]
    new_msgs ≜ [msgs EXCEPT ![node.nodeId] = Tail(msgs[node.nodeId])]
  IN
    ∧ configuration′ = new_config
    ∧ msgs′ = new_msgs
    ∧ offlineNodes′ = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
```

$\land\ failedNodes' = FailuresNotHandled(new\_config)$

Updates the value of a key. Creates a new version of the key and sends it to its parent and interested child.

$Update(n,\ k,\ newValue,\ sourceId,\ sourceVV)\ \triangleq$

LET

$node \triangleq configuration[n]$
$nodeInformation \triangleq node.vv[n]$
$parent \triangleq nodeInformation.parent$

$datastore \triangleq node.datastore$
$key \triangleq datastore[k]$
$childInterested \triangleq key.childInterested$
$version \triangleq key.versionId$

$ignore \triangleq n \neq sourceId \land version.nodeId = n \land key.value = newValue$

$readingMessage \triangleq$
$\lor\ n \neq sourceId$
$\lor\ \land\ n = sourceId$
$\land\ \exists\, ack \in configuration[n].waiting\_acks : ack.type =$ "offline_operations"

$updated\_nodeInformation \triangleq$
$[nodeInformation$ EXCEPT $!.executedOperations =$ IF $ignore$
THEN $@$
ELSE $@ + 1]$

$update\_vv \triangleq UpdateVV(n,\ node.vv,\ sourceId,\ sourceVV,\ updated\_nodeInformation,$
$NullVVEntry,\ NullNodeId)$

$updated\_version \triangleq$
$[nodeId \mapsto n,$
$versionNumber \mapsto updated\_nodeInformation.executedOperations]$

$updated\_DS \triangleq [datastore$ EXCEPT $![k].value = newValue,$
$![k].versionId = updated\_version]$

$downstream \triangleq [msgType \mapsto$ "key_update",
$sourceId \mapsto n,$
$sourceVV \mapsto update\_vv,$
$keyId \mapsto k,$
$newValue \mapsto newValue,$
$version \mapsto updated\_version]$

$destinations \triangleq (\{childInterested\} \cup \{parent\}) \setminus \{NullNodeId\}$

$offlineDestinations \triangleq$

```
        {x ∈ destinations : configuration[x].status.connectionStatus = "offline"}

    msgsToSend ≜ [x ∈ destinations ↦ [msgs ↦ ⟨downstream⟩, priority ↦ FALSE]]

    new_waiting_acks ≜
        {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}
IN
    ∨ ∧ ignore
      ∧ IgnoreMessage([node EXCEPT !.vv = update_vv])
    ∨ ∧ ignore = FALSE
      ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
      ∧ configuration' = [configuration EXCEPT
                                        ![n].vv = update_vv,
                                        ![n].datastore  = updated_DS,
                                        ![n].waiting_acks = @ ∪ new_waiting_acks,
                                        ![n].log = Append(@, downstream)]
      ∧ msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, readingMessage)
```

Receives an update from another node. 3 situations can happen :
- No conflict detected. Key is updated and the received version keeps being propagated in the same direction.
- Conflict detected and the current node (node that received the update) wins the conflict. Key is not updated.
- Conflict detected and the received (new) version wins the conflict. Key is updated and the received version keeps being propagated in the same direction.

```
ReceivedKeyUpdate(n) ≜
    LET
        update ≜ Head(msgs[n])
        sourceId  ≜ update.sourceId
        sourceVV ≜ update.sourceVV
        keyUpdated ≜ update.keyId
        newValue ≜ update.newValue
        updateVersion ≜ update.version

        node ≜ configuration[n]
        datastore  ≜ node.datastore
        vv ≜ node.vv
        nodeInformation ≜ vv[n]
        key ≜ datastore[keyUpdated]
        version ≜ key.versionId
        parent ≜ nodeInformation.parent
        childInterested ≜ key.childInterested

        sourceIsAboveInHierarchy ≜
            ∨ ∧ vv[sourceId] ≠ NullVVEntry
              ∧ sourceId ∈ GetNodesAboveInHierarchy(vv,
```

```
                                                    vv[n].parent,
                                                    NullNodeId)
    ∨ ∧ vv[sourceId] = NullVVEntry
      ∧ n ∉ GetNodesAboveInHierarchy(sourceVV,
                                     sourceVV[sourceId].parent,
                                     NullNodeId)

applyUpdate ≜
    IsToApplyUpdate(n, vv, sourceIsAboveInHierarchy, sourceId, sourceVV,
                    updateVersion, version, NullVVEntry,
                    NullNodeId, NullDatastoreEntry)
```

The update should be propagated in the same direction. If it came from a child, sould be sent to the parent. If it came from the parent, sould be sent to a child. If the update was not applied, it will not be propagated. Due to hierarchy changes, the node might receive the update from other than one of its neighbours

```
propagation_destination ≜
        IF applyUpdate
         THEN IF sourceIsAboveInHierarchy
                 THEN {childInterested} ∪ IF sourceId = parent
                                          THEN {}
                                          ELSE {parent}
                 ELSE {parent} ∪ IF sourceId = childInterested
                                 THEN {}
                                 ELSE {childInterested}
         ELSE {}

updated_DS ≜ IF applyUpdate
               THEN [datastore EXCEPT ![keyUpdated].value = newValue,
                                      ![keyUpdated].versionId = updateVersion]
               ELSE datastore

updated_nodeInformation ≜
        [nodeInformation EXCEPT !.executedOperations = IF applyUpdate
                                                       THEN @ + 1
                                                       ELSE @]

updated_vv ≜
    UpdateVV(n, vv, sourceId, sourceVV, updated_nodeInformation,
             NullVVEntry, NullNodeId)


downstream ≜ [msgType ↦ update.msgType,
              sourceId  ↦ n,
              sourceVV ↦ updated_vv,
              keyId ↦ keyUpdated,
              newValue ↦ newValue,
              version ↦ updateVersion]
```

```
        temp_destinations ≜ propagation_destination \ {NullNodeId}
        destinations ≜ IF applyUpdate = FALSE
                       THEN {}
                       ELSE IF ∧ sourceIsAboveInHierarchy = FALSE
                               ∧ key.childInterested ≠ NullNodeId
                               ∧ ∨ vv[sourceId] = NullVVEntry
                                 ∨ sourceId ≠ key.childInterested
                            THEN temp_destinations ∪ {key.childInterested}
                            ELSE temp_destinations

        offlineDestinations ≜
            {x ∈ destinations : configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
            [x ∈ destinations ↦ [msgs ↦ ⟨downstream⟩, priority ↦ FALSE]]

        new_waiting_acks ≜
            {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}

    IN
        ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
        ∧ configuration' = [configuration EXCEPT
                              ![n].vv = updated_vv,
                              ![n].datastore = updated_DS,
                              ![n].waiting_acks = @ ∪ new_waiting_acks,
                              ![n].log = IF applyUpdate
                                         THEN Append(@, downstream)
                                         ELSE @]
        ∧ msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

CreateOrUpdateRemoteKey(n, k, value, nodesInterested, updateSourceId, updateVV) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv
        nodeInformation ≜ vv[n]

        updated_nodeInfo ≜ [nodeInformation EXCEPT !.executedOperations = @ + 1]

        updated_vv ≜
            UpdateVV(n, vv, updateSourceId, updateVV, updated_nodeInfo,
                     NullVVEntry, NullNodeId)

        msgToParent ≜ [msgType ↦ "new_key_or_update",
                       nodesInterested ↦ nodesInterested ∪ {n},
                       keyId ↦ k,
                       value ↦ value,
                       sourceId ↦ n,
```

```
                                sourceVV ↦ updated_vv]

        destinations ≜ {nodeInformation.parent} \ {NullNodeId}
        offlineDestinations ≜
            {x ∈ destinations : configuration[x].status.connectionStatus = "offline"}
        msgsToSend ≜
            [x ∈ destinations ↦ [msgs ↦ ⟨msgToParent⟩, priority ↦ FALSE]]

        new_waiting_acks ≜
            {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}

    IN
        ∧ nodeInformation.parent ∉ nodesInterested
        ∧ configuration' = [configuration EXCEPT
                                        ![n].vv  = updated_vv,
                                        ![n].waiting_acks = @ ∪ new_waiting_acks,
                                        ![n].log = Append(@, msgToParent)]
        ∧ msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, n ≠ updateSourceId)
        ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩


CreateKey(n, k, value, version, nodesInterested, updateSourceId, updateVV) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv
        nodeInformation ≜ vv[n]
        parent ≜ nodeInformation.parent
        datastore ≜ node.datastore

        ignore ≜ datastore[k] ≠ NullDatastoreEntry

        updated_nodeInfo ≜
            [nodeInformation EXCEPT !.executedOperations = IF ignore
                                                           THEN @
                                                           ELSE @ + 1]

        updated_vv ≜
            UpdateVV(n, vv, updateSourceId, updateVV, updated_nodeInfo,
                     NullVVEntry, NullNodeId)

        childInterestedInNewKey ≜
            GetChildInterestedInKey(nodesInterested,
                                    nodeInformation.childrenId,
                                    vv,
                                    NullNodeId,
                                    NullVVEntry)
```

$newVersion \triangleq$ IF $parent \neq NullNodeId$
THEN $version$
ELSE $[nodeId \mapsto n,$
$versionNumber \mapsto updated\_nodeInfo.executedOperations]$

$new\_key \triangleq [keyId \mapsto k,$
$value \mapsto value,$
$childInterested \mapsto childInterestedInNewKey,$
$versionId \mapsto newVersion]$

$updated\_DS \triangleq [datastore$ EXCEPT $![k] = new\_key]$

$msgToChild \triangleq [msgType \mapsto$ "new_key",
$nodesInterested \mapsto nodesInterested \setminus \{n\},$
$keyId \mapsto k,$
$value \mapsto value,$
$version \mapsto newVersion,$
$sourceId \mapsto n,$
$sourceVV \mapsto updated\_vv]$

$destinations \triangleq \{childInterestedInNewKey\} \setminus \{NullNodeId\}$
$offlineDestinations \triangleq$
$\{x \in destinations : configuration[x].status.connectionStatus =$ "offline"$\}$

$msgsToSend \triangleq$
$[x \in destinations \mapsto [msgs \mapsto \langle msgToChild \rangle, priority \mapsto$ FALSE$]]$

$new\_waiting\_acks \triangleq$
$\{[type \mapsto$ "correct_hierarchy", $nodeId \mapsto x] : x \in offlineDestinations\}$

IN
$\lor \land ignore$
$\land IgnoreMessage([node$ EXCEPT $!.vv = updated\_vv])$
$\lor \land ignore =$ FALSE
$\land msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, n \neq updateSourceId)$
$\land$ UNCHANGED $\langle offlineNodes, failedNodes \rangle$
$\land configuration' = [configuration$ EXCEPT $![n].vv = updated\_vv,$
$![n].datastore = updated\_DS,$
$![n].waiting\_acks = @ \cup new\_waiting\_acks,$
$![n].log = Append(@, msgToChild)]$

This function will be executed if a node tries to update (or receives a message to update) a key that doesn't have. If the node is the root, it means that no node has the key, so it creates the key. If it is not the root, propagates the message to its parent

$UpdateUnknownKey(n, k, newValue, version, nodesInterested, updateSourceId, updateVV) \triangleq$
LET
$node \triangleq configuration[n]$

```
        parent ≜ node.vv[n].parent
    IN
        IF parent = NullNodeId
         THEN CreateKey(n, k, newValue, version, nodesInterested,
                        updateSourceId, updateVV)
         ELSE CreateOrUpdateRemoteKey(n, k, newValue, nodesInterested,
                                      updateSourceId, updateVV)
```

Node 'n' updates the value of a key 'k'

```
PUT(n, k, newValue) ≜
    IF configuration[n].datastore[k] ≠ NullDatastoreEntry
     THEN Update(n, k, newValue, n, ⟨⟩)
     ELSE UpdateUnknownKey(n, k, newValue, NullVersion, {}, n, ⟨⟩)
```

Node 'n' received a message from one of its children, trying to update a key that didn't have. If the node has the key, an update is done.

```
ReceivedNewKeyOrUpdate(n) ≜
    LET
        node ≜ configuration[n]
        update ≜ Head(msgs[n])
        updatedKey ≜ update.keyId
        sourceId ≜ update.sourceId
    IN
        IF sourceId ∉ node.vv[n].childrenId
         THEN IgnoreMessage(node)
         ELSE IF configuration[n].datastore[updatedKey] ≠ NullDatastoreEntry
         THEN Update(n, updatedKey, update.value, sourceId, update.sourceVV)
         ELSE UnknownKey(n,
                         updatedKey,
                         update.value,
                         NullVersion,
                         update.nodesInterested,
                         sourceId,
                         update.sourceVV)
```

If a node tries to add another node and finds out that the new node has an incompatible *datastore* with the current hierarchy, the operation to add the new node is canceled. The *id* of the new node returns to the list of *offlineNodes*, so it can be added in the future

```
CancelNewNode(n, node, isFirstStep, sourceVV, newNodeId) ≜
    ∧ msgs' = [msgs EXCEPT ![n] = IF isFirstStep THEN @ ELSE Tail(@)]
    ∧ configuration' = [configuration EXCEPT
                        ![n].vv = IF ∨ isFirstStep
                                     ∨ node.vv[n].parent = NullNodeId
```

```
                                    THEN @
                                    ELSE  UpdateVV(n,
                                                   configuration[n].vv,
                                                   configuration[n].vv[n].parent,
                                                   sourceVV,
                                                   configuration[n].vv[n],
                                                   NullVVEntry,
                                                   NullNodeId),
                              ![newNodeId].status.connectionStatus = "offline"]
  ∧ UNCHANGED failedNodes
  ∧ offlineNodes' = offlineNodes ∪ {newNodeId}


SendNodeToPossibleParent(n, sourceVV, destination, newNodeId, newNodeKeys,
                         updated_config, updated_msgs, sourceId) ≜
  LET
    isFirstStep ≜ sourceVV = ⟨⟩
    node ≜ configuration[n]
    vv ≜ node.vv
    nodeInformation ≜ vv[n]

    opRepeatingMsg ≜ ∃ ack ∈ node.waiting_acks : ack.type = "repeat_now"

    updated_nodeInformation ≜
      [nodeInformation EXCEPT
                    !.executedOperations = IF opRepeatingMsg
                                           THEN @
                                           ELSE @ + 1]

    updated_vv  ≜ UpdateVV(n, vv, sourceId, sourceVV, updated_nodeInformation,
                           NullVVEntry, NullNodeId)

    msgAddNode ≜ [msgType ↦ "new_node",
                  nodeId ↦ newNodeId,
                  keyIds ↦ newNodeKeys,
                  sourceVV ↦ updated_vv,
                  sourceId  ↦ n]

    destinations ≜ {destination} \ {NullNodeId}
    offlineDestinations ≜
      {x ∈ destinations : configuration[x].status.connectionStatus = "offline"}

    msgsToSend ≜
      [x ∈ destinations ↦ [msgs ↦ ⟨msgAddNode⟩, priority ↦ FALSE]]

    new_waiting_acks ≜
      {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}
```

```
IN
    ∧ msgs' = SendMessages(updated_msgs, n, msgsToSend, offlineDestinations,
                           isFirstStep = FALSE)
    ∧ configuration' = [updated_config EXCEPT
                          ![n].vv = updated_vv,
                          ![n].waiting_acks = @ ∪ new_waiting_acks,
                          ![newNodeId].status.connectionStatus = "pending",
                          ![n].log = IF opRepeatingMsg
                                     THEN @
                                     ELSE Append(@, msgAddNode)]
    ∧ offlineNodes' = offlineNodes \ {newNodeId}
    ∧ UNCHANGED failedNodes
```

A node tries to add another node and finds out that the new node must be its child. If some children of the current node have keys in common with the new node, those children must change their parent to the new node. A message of *new_parent* is sent to those children and the new node is added in the hierarchy. The new node won't execute any operation until it receives an *ack* of each child, If the new node executed *offline* operations, those operations must be applied and propagated. Those operations will be copied from the *log* to the messages and will all be executed after receiving all the *ack* of the children

```
AddNewNodeAsChild(n, sourceVV, newNodeId, newNodeChildren, newNodeKeys,
                  updated_config, updated_msgs, sourceId) ≜
LET
    isFirstStep ≜ sourceVV = ⟨⟩
    node ≜ updated_config[n]
    nodeDatastore ≜ node.datastore
    vv ≜ node.vv

    offNode ≜ updated_config[newNodeId]

    newCreatedNode_nodeInformation ≜
        [offNode.vv[newNodeId] EXCEPT
                        !  .executedOperations = offNode.status.numOpOnline,
                        !  .parent = n,
                        !  .childrenId = newNodeChildren]

    -- Update information of the node 'n', that will add the new node to the hierarchy

    newNodesInfos ≜ [nn ∈ {newNodeId} ↦ newCreatedNode_nodeInformation]

    tempVV ≜ AddEntriesToVV(vv, newNodesInfos, {}, NullVVEntry)

    updated_nodeInformation ≜ [tempVV[n] EXCEPT !.executedOperations = @ + 1]

    updated_vv ≜
        UpdateVV(n, tempVV, sourceId, sourceVV, updated_nodeInformation,
                 NullVVEntry, NullNodeId)
```

$updated\_datastore \triangleq$
$ChangeKeysChildInterested(nodeDatastore, newNodeKeys, newNodeId)$

—— *Update* information of the new node that will be added to the hirarchy

$nn\_datastore \triangleq [k \in KeyId \mapsto \text{IF } k \in newNodeKeys$
$\text{THEN } nodeDatastore[k]$
$\text{ELSE } NullDatastoreEntry]$

$nodesAbove \triangleq GetNodesAboveInHierarchy(vv, vv[n].parent, NullNodeId)$

$nodesInterest \triangleq$
$GetNodesRelatedTo(\{newNodeId\}, updated\_vv, NullVVEntry, nodesAbove)$

$nn\_vv \triangleq$
$[nId \in NodeId \mapsto \text{IF } nId = n$
$\text{THEN } updated\_nodeInformation$
$\text{ELSE IF } nId = newNodeId \lor nId \in nodesInterest$
$\text{THEN } updated\_vv[nId]$
$\text{ELSE } NullVVEntry]$

BOOLEAN . Node executed *offline* operations? If it did, those operations must be applied and propagated. Those operations will be placed in the beginning of the node's message queue

$executedOffOps \triangleq offNode.vv[newNodeId].executedOperations \neq offNode.status.numOpOnline$

$ack\_off\_operations \triangleq$
$\text{IF } executedOffOps$
$\text{THEN } \{[nodeId \mapsto newNodeId, type \mapsto \text{“offline\_operations”}]\}$
$\text{ELSE } \{\}$

$offlineOperations \triangleq$
$\text{IF } executedOffOps$
$\text{THEN } Append(SubSeq(offNode.log,$
$offNode.status.numOpOnline + 1,$
$Len(offNode.log)),$
$[msgType \mapsto \text{“offline\_operations”}])$
$\text{ELSE } \langle\rangle$

$msgToParent \triangleq [msgType \mapsto \text{“add\_node\_to\_hierarchy”},$
$nodeId \mapsto newNodeId,$
$sourceVV \mapsto updated\_vv,$
$sourceId \mapsto n]$

$msgToNewNodeChildren \triangleq [msgType \mapsto \text{“new\_parent”},$
$parent \mapsto newNodeId,$
$sourceVV \mapsto updated\_vv,$
$nodeToRemove \mapsto NullNodeId]$

Messages Sent by the current node 'n'

$destinations \triangleq \{updated\_nodeInformation.parent\} \setminus \{NullNodeId\}$
$offlineDestinations \triangleq$
  $\{x \in destinations :$
    $configuration[x].status.connectionStatus = \text{“offline”}\}$

$msgsToSend \triangleq$
  $[nId \in destinations \mapsto [msgs \mapsto \langle msgToParent \rangle,$
    $priority \mapsto \text{FALSE}]]$

$temp\_msgs \triangleq$
  $SendMessages(updated\_msgs,\ n,\ msgsToSend,\ offlineDestinations,$
    $isFirstStep = \text{FALSE})$

$new\_waiting\_acks \triangleq$
  $\{[type \mapsto \text{“correct\_hierarchy”},$
    $nodeId \mapsto x] : x \in offlineDestinations\}$

Messages Sent by the new node 'newNodeId'

$nn\_offlineDestinations \triangleq$
  $\{x \in newNodeChildren :$
    $configuration[x].status.connectionStatus = \text{“offline”}\}$

$nn\_msgsToSend \triangleq$
  $[nId \in newNodeChildren \mapsto [msgs \mapsto \langle msgToNewNodeChildren \rangle,$
    $priority \mapsto \text{FALSE}]]$

$nn\_new\_waiting\_acks \triangleq$
  $\{[type \mapsto \text{“correct\_hierarchy”},$
    $nodeId \mapsto x] : x \in nn\_offlineDestinations\}$

$newCreatedNode \triangleq$
  $[offNode \text{ EXCEPT}$
    $!.datastore = nn\_datastore,$
    $!.vv = nn\_vv,$
    $!.waiting\_acks = \{[nodeId \mapsto c,\ type \mapsto \text{“ack\_parent”}] : c \in newNodeChildren\}$
      $\cup\ ack\_off\_operations$
      $\cup\ nn\_new\_waiting\_acks,$
    $!.status = [@ \text{ EXCEPT } !.connectionStatus = \text{“online”}],$
    $!.log = SubSeq(@,\ 1,\ offNode.status.numOpOnline)]$

IN

$\wedge\ offlineNodes' = offlineNodes \setminus \{newNodeId\}$
$\wedge\ configuration' = [updated\_config \text{ EXCEPT}$
  $![newNodeId] = newCreatedNode,$
  $![n].vv = updated\_vv,$
  $![n].datastore = updated\_datastore,$

```
                                        ![n].waiting_acks = @ ∪ new_waiting_acks,
                                        ![n].log = Append(@, msgToParent)]
    ∧ UNCHANGED failedNodes
    ∧ msgs' = SendMessages([temp_msgs EXCEPT ![newNodeId] = offlineOperations],
                           newNodeId,
                           nn_msgsToSend,
                           nn_offlineDestinations,
                           FALSE)

AddNode(n, newNodeId, sourceVV, sourceId) ≜
  LET
    isFirstStep ≜ sourceVV = ⟨⟩
    node ≜ configuration[n]
    vv ≜ node.vv

    newNodeKeys ≜ GetNodeKeys(newNodeId, NullDatastoreEntry)

    children ≜ vv[n].childrenId
     Checks which child of the corrent node, contains all keys of the new node
    newNodeParent ≜
      {nChild ∈ children :
            newNodeKeys ⊆ GetKeysChildIsInterested(n, nChild, KeyId,
                                                   NullDatastoreEntry)}
     Children have disjunt datasets, so newNodeParent will have 1 id or will be empty.
    parentFound ≜ newNodeParent ≠ {}

   Checks children of the current node whose keys are subsets of the keys of the new node. This
   nodes will be the children of the new node
    newNodeChildren ≜
      {nChild ∈ children :
        GetKeysChildIsInterested(n, nChild,
                                 KeyId,
                                 NullDatastoreEntry) ⊆ newNodeKeys}

     Children that have, at least, one key in common with the keys of the new node (newKeys)
    childrenWithKeysInCommon ≜
      {nChild ∈ children :
        GetKeysChildIsInterested(n, nChild,
                                 KeyId,
                                 NullDatastoreEntry) ∩ newNodeKeys ≠ {}}

    error ≜
      ∨ Cardinality(newNodeKeys) > Cardinality(GetNodeKeys(n, NullDatastoreEntry))
      ∨ ∧ ∨ parentFound = FALSE
          ∨ newNodeChildren ≠ {}
        ∧ newNodeChildren ≠ childrenWithKeysInCommon
```

```
    updated_config ≜
        IF error = FALSE ∧ isFirstStep
         THEN RemovePreviousHierarchyUpdatesOfNode({newNodeId})
         ELSE configuration

    updated_msgs ≜
        IF error = FALSE ∧ isFirstStep
         THEN RemoveInvMsgsUpdatesOfNode({newNodeId})
         ELSE msgs

IN
    IF  ∧ isFirstStep = FALSE
        ∧ ∨ vv[newNodeId] ≠ NullVVEntry
          ∨ NodeWasRemoved(node.log, newNodeId)
          ∨ newNodeId ∈ offlineNodes

     THEN IgnoreMessage([node EXCEPT
                              !.vv = UpdateVV(n, vv, sourceId, sourceVV, vv[n],
                                              NullVVEntry, NullNodeId)])

     ELSE IF error
     THEN CancelNewNode(n, node, isFirstStep, sourceVV, newNodeId)

     Send message to 'newNodeParent'. After receiving the message, the 'newNodeParent' will also execute this function
     ELSE IF parentFound
     THEN SendNodeToPossibleParent(n,
                                   sourceVV,
                                   CHOOSE x ∈ newNodeParent : TRUE,
                                   newNodeId,
                                   newNodeKeys,
                                   updated_config,
                                   updated_msgs,
                                   sourceId)

     Means that the new node will be a child of the current node.
     ELSE AddNewNodeAsChild(n,
                            sourceVV,
                            newNodeId,
                            newNodeChildren,
                            newNodeKeys,
                            updated_config,
                            updated_msgs,
                            sourceId)
```

```
Node 'n' received a message to change its parent. It must send an acknowledge to the new parent,
with any messages that the new parent might be missing. This message can be caused by the
addition of a new node or by the removal of the previous parent. Depending on that, a different
message is propagated to its children, and add to the log
NewParent(n) ≜
    LET
        update ≜ Head(msgs[n])
        newParentId ≜ update.parent
        sourceVV ≜ update.sourceVV
        nodeToRemove ≜ update.nodeToRemove

        vv ≜ configuration[n].vv
        nodeLog ≜ configuration[n].log


        newNodesInfos ≜ [nn ∈ {newParentId} ↦ sourceVV[newParentId]]

        add ≜ nodeToRemove = NullNodeId

        add_online ≜ ∧ add

        tempVV ≜ IF add_online
                 THEN AddEntriesToVV(vv, newNodesInfos, {}, NullVVEntry)
                 ELSE IF add ∨ vv[nodeToRemove] = NullVVEntry
                 THEN vv
                 ELSE RemoveNodeFromVV(vv, vv[nodeToRemove], NullVVEntry)

        updated_nodeInfo ≜ IF add_online ∨ add = FALSE
                           THEN [tempVV[n] EXCEPT !.executedOperations = @ + 1]
                           ELSE tempVV[n]

        updated_vv ≜
            UpdateVV(n, tempVV, newParentId, sourceVV, updated_nodeInfo,
                     NullVVEntry, NullNodeId)


        msgToChildren ≜ [msgType ↦ IF add
                                   THEN "add_node_to_hierarchy"
                                   ELSE "remove_node",
                         nodeId ↦ IF add
                                  THEN newParentId
                                  ELSE nodeToRemove,
                         sourceVV ↦ updated_vv,
                         sourceId ↦ n]

Send an acknowledge to its new parent, plus the messages parent is missing
        numMsgsParentKnowns ≜ sourceVV[n].executedOperations
```

$numExecutedOperations \triangleq tempVV[n].executedOperations$

$msgsToParent1 \triangleq$
$\quad SelectSeqToParent(SubSeq(nodeLog,$
$\qquad\qquad\qquad\qquad numMsgsParentKnowns + 1,$
$\qquad\qquad\qquad\qquad numExecutedOperations),$
$\qquad\qquad\qquad sourceVV,$
$\qquad\qquad\qquad n,$
$\qquad\qquad\qquad NullVVEntry,$
$\qquad\qquad\qquad KeyId,$
$\qquad\qquad\qquad NullDatastoreEntry)$

$ackToParent \triangleq$
$\quad [msgType \mapsto$ "ack_parent", $sourceId \mapsto n,\ sourceVV \mapsto updated\_vv]$

$msgsToParent2 \triangleq Append(msgsToParent1,\ ackToParent)$

$msgPS \triangleq [msgType \mapsto$ "package_start", $sourceId \mapsto n]$
$msgPE \triangleq [msgType \mapsto$ "package_end", $sourceId \mapsto n]$

$finalMsgsParent \triangleq$ IF $Len(msgsToParent2) > 1$
$\qquad\qquad\qquad\qquad$ THEN $Append(\langle msgPS \rangle \circ msgsToParent2,\ msgPE)$
$\qquad\qquad\qquad\qquad$ ELSE $msgsToParent2$

$destinations \triangleq \{newParentId\} \cup$ IF $add =$ FALSE $\vee\ add\_online$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ THEN $updated\_nodeInfo.childrenId$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ ELSE $\{\}$

$offlineDestinations \triangleq$
$\quad \{x \in destinations : configuration[x].status.connectionStatus =$ "offline"$\}$

$msgsToSend \triangleq$
$\quad [nId \in destinations \mapsto$
$\qquad\qquad$ IF $nId \in updated\_nodeInfo.childrenId$
$\qquad\qquad$ THEN $[msgs \mapsto \langle msgToChildren \rangle,\ priority \mapsto$ FALSE$]$
$\qquad\qquad$ ELSE IF $nId = newParentId$
$\qquad\qquad$ THEN $[msgs \mapsto finalMsgsParent,\ priority \mapsto$ TRUE$]$
$\qquad\qquad$ ELSE $\langle\rangle]$

$new\_waiting\_acks \triangleq$
$\quad \{[type \mapsto$ "correct_hierarchy", $nodeId \mapsto x] : x \in offlineDestinations\}$

$new\_msgs \triangleq SendMessages(msgs,\ n,\ msgsToSend,\ offlineDestinations,$ TRUE$)$

$ack\_removed\_node \triangleq \{[type \mapsto$ "correct_hierarchy", $nodeId \mapsto nodeToRemove]\}$

$changed\_log \triangleq$
$\quad RemoveHierarchyUpdatesOfNodeFromLog(nodeLog,$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ IF add

```
                                                THEN {newParentId}
                                                ELSE {nodeToRemove})
        new_config ≜
            [configuration EXCEPT
                ![n].vv = updated_vv,
                ![n].waiting_acks = (@ ∪ new_waiting_acks) \ ack_removed_node,
                ![n].log = IF add = FALSE ∨ add_online
                           THEN Append(changed_log, msgToChildren)
                           ELSE @]
    IN
        ∧ msgs' = new_msgs
        ∧ configuration' = new_config
        ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
        ∧ failedNodes' = FailuresNotHandled(new_config)
```

Receives acknowledge of one child. After receiving acknowledge from every children, the node can start executing operations

```
ReceiveChildConfirmation(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv
        update ≜ Head(msgs[n])

        updated_vv ≜ UpdateVV(n, vv, update.sourceId, update.sourceVV,
                              vv[n], NullVVEntry, NullNodeId)
    IN
        ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
        ∧ configuration' = [configuration EXCEPT
                              ![n].waiting_acks = @ \ {[nodeId ↦ update.sourceId,
                                                        type ↦ "ack_parent"]},
                              ![n].vv = updated_vv]
        ∧ msgs' = [msgs EXCEPT ![n] = Tail(@)]
```

Receives a message informing that a new node was added to the hierarchy.

```
ReceiveNewHierarchyNode(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv

        update ≜ Head(msgs[n])
        newNodeId ≜ update.nodeId
        sourceId ≜ update.sourceId
        sourceVV ≜ update.sourceVV
```

If the node was previously added (and possibly removed) or if it is not related with the current node

$$
\begin{aligned}
ignore \triangleq\ & \lor\ vv[newNodeId] \neq NullVVEntry \\
& \lor\ NodeWasRemoved(node.log,\ newNodeId) \\
& \lor\ vv[sourceVV[newNodeId].parent] = NullVVEntry \\
& \lor\ \land\ sourceVV[newNodeId].childrenId \neq \{\} \\
& \quad \land \forall\ c \in sourceVV[newNodeId].childrenId : vv[c] = NullVVEntry
\end{aligned}
$$

$$
\begin{aligned}
&sourceIsAboveInHierarchy \triangleq \\
&\quad \lor\ \land\ vv[sourceId] \neq NullVVEntry \\
&\qquad \land\ sourceId \in GetNodesAboveInHierarchy(vv,\ vv[n].parent,\ NullNodeId) \\
&\quad \lor\ \land\ vv[sourceId] = NullVVEntry \\
&\qquad \land\ n \notin GetNodesAboveInHierarchy(sourceVV, \\
&\qquad\qquad\qquad\qquad\qquad\qquad sourceVV[sourceId].parent, \\
&\qquad\qquad\qquad\qquad\qquad\qquad NullNodeId)
\end{aligned}
$$

$$
newNodesInfos \triangleq [nn \in \{newNodeId\} \mapsto sourceVV[newNodeId]]
$$

$$
\begin{aligned}
&tempVV \triangleq \text{IF } ignore \\
&\qquad\qquad \text{THEN } vv \\
&\qquad\qquad \text{ELSE } AddEntriesToVV(vv,\ newNodesInfos,\ \{\},\ NullVVEntry)
\end{aligned}
$$

$$
\begin{aligned}
&updated\_nodeInf \triangleq \\
&\quad [tempVV[n] \text{ EXCEPT } !.executedOperations = \text{IF } ignore \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } @ \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE } @ + 1]
\end{aligned}
$$

$$
\begin{aligned}
&final\_vv \triangleq \\
&\quad UpdateVV(n,\ tempVV,\ sourceId,\ sourceVV,\ updated\_nodeInf, \\
&\qquad\qquad NullVVEntry,\ NullNodeId)
\end{aligned}
$$

$$
\begin{aligned}
msgToPropagate \triangleq\ & [msgType \mapsto \text{“add\_node\_to\_hierarchy”}, \\
& \ nodeId \mapsto newNodeId, \\
& \ sourceVV \mapsto final\_vv, \\
& \ sourceId \ \mapsto n]
\end{aligned}
$$

$$
\begin{aligned}
propagation\_destination \triangleq\ & \text{IF } sourceIsAboveInHierarchy \\
& \text{THEN } final\_vv[n].childrenId \\
& \text{ELSE } \{final\_vv[n].parent\}
\end{aligned}
$$

$$
destinations \triangleq propagation\_destination \setminus \{NullNodeId\}
$$

$$
\begin{aligned}
&offlineDestinations \triangleq \\
&\quad \{x \in destinations : configuration[x].status.connectionStatus = \text{“offline”}\}
\end{aligned}
$$

$$
\begin{aligned}
&msgsToSend \triangleq \\
&\quad [nId \in destinations \mapsto [msgs \mapsto \langle msgToPropagate \rangle,\ priority \mapsto \text{FALSE}]]
\end{aligned}
$$

$new\_waiting\_acks \triangleq$
$\{[type \mapsto$ "correct_hierarchy", $nodeId \mapsto x] : x \in offlineDestinations\}$

$changed\_log \triangleq RemoveHierarchyUpdatesOfNodeFromLog(node.log, \{newNodeId\})$

$new\_msgs \triangleq$
IF $ignore$
THEN $[msgs$ EXCEPT $![n] = Tail(@)]$
ELSE $SendMessages(msgs, n, msgsToSend, offlineDestinations,$ TRUE$)$

$new\_config \triangleq$
IF $ignore$
THEN $[configuration$ EXCEPT $![n].vv = final\_vv]$
ELSE $[configuration$ EXCEPT
$![n].vv = final\_vv,$
$![n].waiting\_acks = @ \cup new\_waiting\_acks,$
$![n].log = Append(changed\_log, msgToPropagate)]$

IN
$\land offlineNodes' = GetOfflineNodes(new\_config, new\_msgs, NullVVEntry)$
$\land failedNodes' = FailuresNotHandled(new\_config)$
$\land msgs' = new\_msgs$
$\land configuration' = new\_config$

When a set of messages is sent and must be propagated continuously, they are preceded by a message informing the start of the pack of messages, and followed be a message informing the end of the pack of messages. If a node starts to read a message informing the beginning of a message pack, all the next states are executed by that node, until it receives the end of the messages pack.

$ReceiveStartOfMessagesPack(n) \triangleq$
LET
$parent \triangleq configuration[n].vv[n].parent$
$children \triangleq configuration[n].vv[n].childrenId$
$update \triangleq Head(msgs[n])$
IN
$\land msgs' = [nId \in NodeId \mapsto$
IF $nId = n$
THEN $Tail(msgs[nId])$
ELSE IF $\lor nId = parent$
$\lor nId \in children$
$\lor nId = update.sourceId$
THEN $Append(msgs[nId], update)$
ELSE $msgs[nId]]$
$\land$ UNCHANGED $\langle offlineNodes, failedNodes\rangle$
$\land configuration' = [configuration$ EXCEPT
$![n].waiting\_acks = @ \cup \{[nodeId \mapsto update.sourceId,$

```
                                                    type ↦ "receiving_package"]}]

ReceiveFinalOfMessagesPack(n, update) ≜
    ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
    ∧ configuration′ = [configuration EXCEPT
                            ![n].waiting_acks = @ \ {[nodeId ↦ update.sourceId,
                                                      type ↦ "receiving_package"]}]
    ∧ msgs′ = [nId ∈ NodeId ↦
                  IF nId = n
                   THEN Tail(msgs[n])

                   ELSE LET
                            numStartPacks ≜ Len(SelectSeq(msgs[nId], IsPackageStart))
                            numEndPacks ≜ Len(SelectSeq(msgs[nId], IsPackageEnd))

                            packSize ≜ IF numStartPacks > numEndPacks
                                        THEN GetSizeOfMsgsPack(msgs[nId])
                                        ELSE −1
                        IN
                            IF packSize = −1
                             THEN msgs[nId]
                             ELSE IF packSize = 0
                             THEN SubSeq(msgs[nId], 1, Len(msgs[nId]) − 1)
                             ELSE IF packSize = 1
                             THEN SubSeq(msgs[nId], 1, Len(msgs[nId]) − 2)
                                    ∘   SubSeq(msgs[nId],
                                               Len(msgs[nId]),
                                               Len(msgs[nId]))
                             ELSE Append(msgs[nId], update)]
```

Remove Node from Hierarchy

Node tries to remove itself. Sends message to parent informing of all its children and all their datasets, so that the parent can add them as new children and remove it. Waits for parent acknowledge to be removed. Root cannot execute this operation

```
Remove(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv
        parent ≜ vv[n].parent
        children ≜ vv[n].childrenId
        datastore ≜ node.datastore

        updated_nodeInformation ≜ [vv[n] EXCEPT !.executedOperations = @ + 1]

        updated_vv ≜
```

```
        UpdateVV(n, vv, n, ⟨⟩, updated_nodeInformation,
                 NullVVEntry, NullNodeId)

    msgToParent ≜
        [msgType ↦ "remove_child",
         nodeToRemove ↦ n,
         newChildren ↦ [nId ∈ NodeId ↦
                        IF nId ∈ children
                        THEN GetKeysChildIsInterested(n, nId, KeyId, NullDatastoreEntry)
                        ELSE {}],
         sourceId   ↦ n,
         sourceVV ↦ updated_vv]

    offlineDestinations ≜
        {x ∈ {parent} : configuration[x].status.connectionStatus = "offline"}

    msgsToSend ≜
        [x ∈ {parent} ↦ [msgs ↦ ⟨msgToParent⟩, priority ↦ FALSE]]

    new_waiting_acks ≜
        {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}
IN
    ∧ parent ≠ NullNodeId
    ∧ node.status.connectionStatus = "online"
    ∧ Len(msgs[n]) = 0
    ∧ node.waiting_acks = {}
    ∧ configuration' = [configuration EXCEPT
                        ![n].vv = updated_vv,
                        ![n].waiting_acks = @ ∪ {[nodeId ↦ parent,
                                                  type ↦ "ack_remove"]}
                                              ∪ new_waiting_acks,
                        ![n].log = Append(@, msgToParent)]
    ∧ msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, FALSE)
    ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
```

If the node receives a message from a node that it is not its child, the message is ignored. The child sent a message with all its children and their datasets. Node 'n' removes child from its children, adds the children of that child to its own children, and then sends message to the new children, informing that it is the new parent and that the previous parent will be removed. Some messages might be sent to each of the new children. Those messages might need to be re-propagated the node that will be removed didn't process them before asking for removal. To the child that will be removed, an acknowledge message to allow the removal is sent. To the parent of the this node, a message informing that a node will be removed from the hierarchy is sent.

```
RemoveChild(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv
```

```
    update ≜ Head(msgs[n])
    nodeToRemove ≜ update.nodeToRemove
    sourceVV ≜ update.sourceVV
    scId ≜ update.sourceId

    updated_vv ≜ UpdateVV(n, vv, scId, sourceVV, vv[n],
                          NullVVEntry, NullNodeId)
IN
    IF nodeToRemove ∉ vv[n].childrenId
     THEN IgnoreMessage([node EXCEPT !.vv = updated_vv])
     ELSE
        LET
            inheritChildren ≜
                InheritChildren(n, vv, nodeToRemove, scId, sourceVV,
                                NullVVEntry, NullNodeId, NullDatastoreEntry)

            keysOfNewChildren ≜ inheritChildren.keysOfNewChildren
            newChildren ≜ inheritChildren.newChildrenKeys
            end_vv ≜ inheritChildren.vv

            DS ≜ [k ∈ KeyId ↦
                    LET key ≜ node.datastore[k]
                    IN
                        IF ∨ key = NullDatastoreEntry
                           ∨ ∧ key.childInterested ≠ nodeToRemove
                             ∧ k ∉ keysOfNewChildren
                         THEN key
                         ELSE [key EXCEPT
                                 !.childInterested = NewChildInterestedInKey(k,
                                                                             newChildren,
                                                                             NullNodeId)]
                ]

            tempNumMsgsKnown ≜ IF sourceVV[n] = NullVVEntry
                               THEN 0
                               ELSE sourceVV[n].executedOperations

            numMsgsKnown ≜ Max(tempNumMsgsKnown, node.status.numOpOnline)
            numExecutedOperations ≜ vv[n].executedOperations

            msgsMissing ≜ SubSeq(node.log, numMsgsKnown + 1, numExecutedOperations)

            newKeyMsgsMissing ≜ SelectSeq(msgsMissing, IsNewKey)

            newChildrenId ≜ DOMAIN newChildren

            final_datastore ≜ UpdateDSWithNewKey(DS,
                                                 newKeyMsgsMissing,
```

$newChildrenId$,
$sourceVV$,
$NullVVEntry$,
$NullNodeId$)

If this node tried to add a new node, and sent a message "new_node" to the child that will be removed (and the child didn't execute that operation), this node must re-execute that operation to decide if the new node can still enter the hierarchy

$msgsToRepeat1 \triangleq SelectSeqOfNewNodes(msgsMissing, end\_vv, NullVVEntry)$
$msgsToRepeat \triangleq$
IF $Len(msgsToRepeat1) > 0$
THEN $Append(msgsToRepeat1, [msgType \mapsto$ "repeat_now",
$sourceId \mapsto n])$
ELSE $\langle\rangle$

$msgToNewChildren \triangleq [msgType \mapsto$ "new_parent",
$parent \mapsto n,$
$nodeToRemove \mapsto nodeToRemove,$
$sourceVV \mapsto end\_vv]$

$nodesAbove \triangleq GetNodesAboveInHierarchy(vv, vv[n].parent, NullNodeId)$

$msgsChildrenAreMissing \triangleq$
$SelectSeqToChildren(msgsMissing,$
$final\_datastore,$
$end\_vv,$
$sourceVV,$
$[x \in newChildrenId \mapsto \langle\rangle],$
$NullVVEntry,$
$\langle msgToNewChildren\rangle,$
$nodesAbove)$

$msgPS \triangleq [msgType \mapsto$ "package_start", $sourceId \mapsto n]$
$msgPE \triangleq [msgType \mapsto$ "package_end", $sourceId \mapsto n]$

$msgsToChildren \triangleq$
$[c \in newChildrenId \mapsto$
IF $Len(msgsChildrenAreMissing[c]) > 1$
THEN $Append(\langle msgPS\rangle \circ msgsChildrenAreMissing[c], msgPE)$
ELSE $msgsChildrenAreMissing[c]]$

$nodesRemoved \triangleq inheritChildren.nodesRemoved$
$nodesAdded \triangleq inheritChildren.nodesAdded$

$setMsgsToParentR \triangleq$
$\{[msgType \mapsto$ "remove_node",
$nodeId \mapsto nId,$
$sourceVV \mapsto end\_vv,$

```
        sourceId   ↦ n] : nId ∈ {nodeToRemove} ∪ nodesRemoved}

msgsToParentR ≜ SetToSequence(setMsgsToParentR, ⟨⟩)

setMsgsToParentA ≜ {[msgType ↦ "add_node_to_hierarchy",
                     nodeId ↦ nId,
                     sourceVV ↦ end_vv,
                     sourceId   ↦ n] : nId ∈ nodesAdded}

nodesChanged ≜ ({nodeToRemove} ∪ nodesRemoved) ∪ nodesAdded

changed_log ≜
    RemoveHierarchyUpdatesOfNodeFromLog(node.log, nodesChanged)

msgsToParentA ≜ SetToSequence(setMsgsToParentA, ⟨⟩)

tempMsgToParent ≜ msgsToParentA ∘ msgsToParentR

msgsToParent ≜ IF Len(tempMsgToParent) > 1
                 THEN Append(⟨msgPS⟩ ∘ tempMsgToParent, msgPE)
                 ELSE tempMsgToParent

msgsToOldChild ≜ [msgType ↦ "ack_remove", sourceId ↦ n]

updated_nodeInfo ≜
    [end_vv[n] EXCEPT
        !.executedOperations = @ + 1 + Cardinality(nodesAdded)
                                     + Cardinality(nodesRemoved)]

destinations ≜
    ({updated_nodeInfo.parent, n, nodeToRemove}
        ∪ newChildrenId)
            \ {NullNodeId}

offlineDestinations ≜
    {x ∈ destinations :
          configuration[x].status.connectionStatus = "offline"}

msgsToSend ≜
    [nId ∈ destinations ↦
              IF nId = updated_nodeInfo.parent
               THEN [msgs ↦ msgsToParent, priority ↦ FALSE]
               ELSE IF nId = n
               THEN [msgs ↦ msgsToRepeat, priority ↦ TRUE]
               ELSE IF nId ∈ newChildrenId
               THEN [msgs ↦ msgsToChildren[nId], priority ↦ FALSE]
               ELSE IF nId = nodeToRemove
               THEN [msgs ↦ ⟨msgsToOldChild⟩, priority ↦ TRUE]
               ELSE ⟨⟩]
```

```
        new_waiting_acks ≜
            {[type ↦ "correct_hierarchy", nodeId ↦ x] : x ∈ offlineDestinations}

        new_config ≜
            [configuration EXCEPT
                ![n].vv = [end_vv EXCEPT ![n] = updated_nodeInfo],
                ![n].datastore  = final_datastore,
                ![n].waiting_acks = IF Len(msgsToRepeat) = 0
                                    THEN @ ∪ new_waiting_acks
                                    ELSE @ ∪ {[nodeId ↦ n,
                                               type ↦ "repeat_now"]}
                                           ∪ new_waiting_acks,
                ![n].log = changed_log ∘ tempMsgToParent]
    IN
        ∧ msgs' = SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)
        ∧ UNCHANGED offlineNodes
        ∧ failedNodes' = FailuresNotHandled(new_config)
        ∧ configuration' = new_config
```

To make a node re-execute an operation, a message with that operation, followed by a message with "repeat_now" are added in the beggining of the messages sequence. While the node doesn't receive the last message (with type "repeat_now"), the *Next* state will always oblige this node to read messages

```
ReceiveRepeatNow(n) ≜
    ∧ msgs' = [msgs EXCEPT ![n] = Tail(@)]
    ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩
    ∧ configuration' =
        [configuration EXCEPT
            ![n].waiting_acks = @ \ {[nodeId ↦ n,
                                      type ↦ "repeat_now"]}]
```

Received a message informing that a node was removed from the hierarchy. Node 'n' removes entry from its *VV* and keeps propagating the message

```
NodeRemoved(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv

        update ≜ Head(msgs[n])
        nodeToRemove ≜ update.nodeId
        sourceVV ≜ update.sourceVV
        sourceId ≜ update.sourceId
    IN
        IF ∨ vv[nodeToRemove] = NullVVEntry
           ∨ nodeToRemove = n
        THEN
```

```
        LET
            new_config ≜
               [configuration EXCEPT
                     ![n].vv = UpdateVV(n, vv, sourceId, sourceVV,
                                        vv[n], NullVVEntry, NullNodeId)]

            new_msgs ≜
               [msgs EXCEPT
                     ![n] = RemoveHierarchyUpdatesOfNodeFromMsgs(Tail(@),
                                                                 {nodeToRemove})]
            ackToRemove ≜ [nodeId ↦ nodeToRemove, type ↦ "correct_hierarchy"]
        IN
            ∧ configuration' = [new_config EXCEPT
                                   ![n].waiting_acks = @ \ {ackToRemove}]
            ∧ msgs' = new_msgs
            ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
            ∧ failedNodes' = FailuresNotHandled(new_config)

    ELSE
        LET
            executeOp ≜ vv[nodeToRemove] ≠ NullVVEntry

            updated_nodeInf ≜
               [vv[n] EXCEPT !.executedOperations = IF executeOp
                                                    THEN @ + 1
                                                    ELSE @]

            updated_vv ≜
               UpdateVV(n, vv, sourceId, sourceVV, updated_nodeInf,
                        NullVVEntry, NullNodeId)

            final_vv ≜
               IF executeOp
               THEN RemoveNodeFromVV(updated_vv, updated_vv[nodeToRemove], NullVVEntry)
               ELSE updated_vv


            msgToPropagate ≜ [msgType ↦ "remove_node",
                              nodeId ↦ nodeToRemove,
                              sourceVV ↦ final_vv,
                              sourceId  ↦ n]

            sourceIsAboveInHierarchy ≜
               ∨ ∧ vv[sourceId] ≠ NullVVEntry
                 ∧ sourceId ∈ GetNodesAboveInHierarchy(vv, vv[n].parent, NullNodeId)
               ∨ ∧ vv[sourceId] = NullVVEntry
                 ∧ n ∉ GetNodesAboveInHierarchy(sourceVV,
```

```
                                                    sourceVV[sourceId].parent,
                                                    NullNodeId)

    propagation_destination ≜
        IF executeOp
         THEN IF sourceIsAboveInHierarchy
                 THEN final_vv[n].childrenId
                 ELSE {final_vv[n].parent}
         ELSE {}

    destinations ≜ propagation_destination \ {NullNodeId}
    offlineDestinations ≜
        {x ∈ destinations :
              configuration[x].status.connectionStatus = "offline"}

    msgsToSend ≜
        [nId ∈ destinations ↦ [msgs ↦ ⟨msgToPropagate⟩,
                               priority ↦ FALSE]]

    new_waiting_acks ≜
        {[type ↦ "correct_hierarchy",
          nodeId ↦ x] : x ∈ offlineDestinations}

    temp_msgs ≜ SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)
    new_msgs ≜
        [temp_msgs EXCEPT
            ![n] = RemoveHierarchyUpdatesOfNodeFromMsgs(@, {nodeToRemove})]

    ack_removed_node ≜ {[type ↦ "correct_hierarchy", nodeId ↦ nodeToRemove]}

    changed_log ≜ RemoveHierarchyUpdatesOfNodeFromLog(node.log, {nodeToRemove})

    new_config ≜
        [configuration EXCEPT
            ![n].vv = final_vv,
            ![n].waiting_acks = (@ ∪ new_waiting_acks) \ ack_removed_node,
            ![n].log = IF executeOp
                       THEN Append(changed_log, msgToPropagate)
                       ELSE @]
IN
    ∧ configuration' = new_config
    ∧ msgs' = new_msgs
    ∧ failedNodes' = FailuresNotHandled(new_config)
    ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
```

After asking the parent to be removed and receiving the acknowledge, if becomes *offline* The node stores the number of operations executed until this point, to be able to calculate how many *offline* operations were executed.

```
ReceiveRemoveAck(n) ≜
  LET
    node ≜ configuration[n]
    datastore ≜ node.datastore
    children ≜ node.vv[n].childrenId

    rem_ack ≜ CHOOSE ack ∈ node.waiting_acks : ack.type = "ack_remove"

    updated_datastore ≜
      [k ∈ KeyId ↦ IF datastore[k] = NullDatastoreEntry
                   THEN NullDatastoreEntry
                   ELSE [datastore[k] EXCEPT
                           !.childInterested = NullNodeId]]

    msgCorrectHierarchy ≜
      (SelectSeq(msgs[n], IsCorrectHierarchy)
        ∘ SelectSeq(msgs[n], IsNewParent))
          ∘ SelectSeq(msgs[n], IsNodeFailure)

    destinations ≜ GetSourcesOfMessages(msgCorrectHierarchy)
    msgsToSend ≜
      [nId ∈ destinations ↦
          [msgs ↦ ⟨[msgType ↦ "node_failed",
                    nodeId ↦ n,
                    sourceId ↦ n]⟩, priority ↦ TRUE]]

    new_msgs ≜
      [SendMessages(msgs, n, msgsToSend, {}, FALSE) EXCEPT ![n] = ⟨⟩]


    update_vv ≜
      [nId ∈ NodeId ↦ IF nId = n
                     THEN [node.vv[nId] EXCEPT !.parent = NullNodeId,
                                               !.childrenId = {}]
                     ELSE NullVVEntry]

    new_config ≜
      [configuration EXCEPT
          ![n].vv = update_vv,
          ![n].datastore  = updated_datastore,
          ![n].waiting_acks = {},
          ![n].status =
              [connectionStatus ↦ "offline",
               numOpOnline ↦ node.vv[n].executedOperations],
          ![n].log = RemoveRemoveChildMessage(@)]
```

IN

$\land \forall\, c \in children : configuration[c].vv[c].parent \neq n$
$\land\ msgs' = new\_msgs$
$\land\ configuration' = new\_config$
$\land\ offlineNodes' = GetOfflineNodes(new\_config,\ new\_msgs,\ NullVVEntry)$
$\land\ failedNodes' = FailuresNotHandled(new\_config)$

Failure of a Hierarchy Node

$HandleFailure(n,\ ackToRemove,\ childFailed,\ source) \triangleq$

LET

$node \triangleq configuration[n]$
$vv \triangleq node.vv$
$oldChildren \triangleq vv[n].childrenId$

Information about nodes that failed

$keysOfChild \triangleq$
$\quad GetKeysChildIsInterested(n,\ childFailed,\ KeyId,\ NullDatastoreEntry)$

$nodesFailed \triangleq GetNodesFailed(vv,\ \{childFailed\},\ \{\},\ n,\ NullNodeId)$

$temp1\_vv \triangleq RemoveNodesFromVV(vv,\ nodesFailed,\ NullVVEntry)$

$tempNewConnections \triangleq temp1\_vv[n].childrenId \setminus oldChildren$

Added nodes but not yet known

$opAddNodesNotKnown \triangleq$
$\quad SelectSeqOfNodesAddedButNotKnown(node.log,\ n,\ childFailed,$
$\qquad\qquad tempNewConnections,$
$\qquad\qquad NullNodeId,\ NullVVEntry,$
$\qquad\qquad NullDatastoreEntry)$

$nodesNotKnown \triangleq$
$\quad \{opAddNodesNotKnown[i].nodeId : i \in 1\,..\,Len(opAddNodesNotKnown)\}$

Decide which nodes will become children

$nodesToAdd \triangleq$
$\quad GetNewChildren(nodesNotKnown,\ tempNewConnections,\ n,$
$\qquad\qquad NullNodeId,\ NullVVEntry)$

$temp2\_vv \triangleq AddEntriesToVV(temp1\_vv,\ nodesToAdd,\ \{\},\ NullVVEntry)$

$updated\_vv \triangleq$
$\quad [temp2\_vv$ EXCEPT
$\qquad ![n].executedOperations = @ + Cardinality(nodesFailed)$
$\qquad\qquad + Cardinality(\text{DOMAIN}\ nodesToAdd)]$

Remove information of the failed child from the *datastore* (keys it was interested in)

$$
\begin{array}{l}
datastore \triangleq node.datastore \\
DS \triangleq [k \in KeyId \mapsto \\
\quad \text{IF } k \in keysOfChild \\
\quad \text{THEN } [datastore[k] \text{ EXCEPT } !.childInterested = NullNodeId] \\
\quad \text{ELSE } datastore[k]]
\end{array}
$$

Messages to propagate to its parent informing about the failed nodes and the new added nodes

$$
\begin{array}{l}
msgPS \triangleq [msgType \mapsto \text{“package\_start”}, sourceId \mapsto n] \\
msgPE \triangleq [msgType \mapsto \text{“package\_end”}, sourceId \mapsto n]
\end{array}
$$

$$
\begin{array}{l}
setMsgsToRemoveNodes \triangleq \{[msgType \mapsto \text{“remove\_node”}, \\
\qquad\qquad\qquad\qquad nodeId \mapsto nId, \\
\qquad\qquad\qquad\qquad sourceVV \mapsto updated\_vv, \\
\qquad\qquad\qquad\qquad sourceId \mapsto n] : nId \in nodesFailed\}
\end{array}
$$

$$
\begin{array}{l}
setMsgsToAddNodes \triangleq \{[msgType \mapsto \text{“add\_node\_to\_hierarchy”}, \\
\qquad\qquad\qquad\quad nodeId \mapsto nId, \\
\qquad\qquad\qquad\quad sourceVV \mapsto updated\_vv, \\
\qquad\qquad\qquad\quad sourceId \mapsto n] : nId \in \text{DOMAIN } nodesToAdd\}
\end{array}
$$

$$
setMsgsToPropagate \triangleq setMsgsToRemoveNodes \cup setMsgsToAddNodes
$$

$$
msgsToPropagate \triangleq SetToSequence(setMsgsToPropagate, \langle\rangle)
$$

$$
\begin{array}{l}
finalMsgsToPropagate \triangleq \text{IF } Len(msgsToPropagate) > 1 \\
\qquad\qquad\qquad\qquad\quad \text{THEN } Append(\langle msgPS \rangle \circ msgsToPropagate, msgPE) \\
\qquad\qquad\qquad\qquad\quad \text{ELSE } msgsToPropagate
\end{array}
$$

Message to the new connections to correct the hierarchy

$$
\begin{array}{l}
msgToCorrectHierarchy \triangleq \\
\quad [msgType \mapsto \text{“correct\_hierarchy”}, \\
\quad nodesFailed \mapsto nodesFailed, \\
\quad keysId \mapsto GetNodeKeys(n, NullDatastoreEntry), \\
\quad sourceId \mapsto n, \\
\quad sourceVV \mapsto updated\_vv]
\end{array}
$$

$$
\begin{array}{l}
msg\_unlock\_source \triangleq \\
\quad [msgType \mapsto \text{“ack\_hierachy\_corrected”}, \\
\quad sourceId \mapsto n, \\
\quad sourceVV \mapsto [nId \in NodeId \mapsto \\
\qquad\qquad\qquad \text{IF } updated\_vv[nId] = NullVVEntry \\
\qquad\qquad\qquad \text{THEN } updated\_vv[nId] \\
\qquad\qquad\qquad \text{ELSE } [updated\_vv[nId] \text{ EXCEPT} \\
\qquad\qquad\qquad\qquad\qquad !.executedOperations = 0]]]
\end{array}
$$

Send Messages

$new\_connections \triangleq updated\_vv[n].childrenId \setminus oldChildren$

$nodesBelowNewConn \triangleq GetNodesBelow(new\_connections,\ updated\_vv,\ NullVVEntry)$

$unlockSource \triangleq source \in nodesBelowNewConn \land source \notin new\_connections$

$destinations \triangleq$
$(new\_connections \cup (\{updated\_vv[n].parent\} \setminus \{NullNodeId\}))$
$\cup$ IF $unlockSource$
THEN $\{source\}$
ELSE $\{\}$

$offlineDestinations \triangleq$
$\{x \in destinations : configuration[x].status.connectionStatus = \text{“offline”}\}$

$msgsToSend \triangleq$
$[nId \in destinations \mapsto$
IF $nId = updated\_vv[n].parent$
THEN $[msgs \mapsto finalMsgsToPropagate,\ priority \mapsto \text{FALSE}]$
ELSE IF $nId = source \land unlockSource$
THEN $[msgs \mapsto \langle msg\_unlock\_source \rangle,\ priority \mapsto \text{TRUE}]$
ELSE $[msgs \mapsto \langle msgToCorrectHierarchy \rangle,\ priority \mapsto \text{TRUE}]]$

$new\_msgs \triangleq SendMessages(msgs,\ n,\ msgsToSend,\ offlineDestinations,\ \text{TRUE})$

$opAddsFailed \triangleq$
$SelectSeqOfAddsFailed(node.log,\ n,\ keysOfChild,$
$NullNodeId,\ NullVVEntry)$

$addsFailed \triangleq \{opAddsFailed[i].nodeId : i \in 1\ ..\ Len(opAddsFailed)\}$

$rhufm \triangleq RemoveHierarchyUpdatesOfNodeFromMsgs(new\_msgs[n],$
$\{ackToRemove.nodeId\})$

$node\_messages \triangleq RemoveAddsOfOfflines(rhufm,\ addsFailed)$

$final\_msgs \triangleq [new\_msgs \text{ EXCEPT } ![n] = node\_messages]$

$new\_waiting\_acks \triangleq$
$\{[type \mapsto \text{“correct\_hierarchy”},$
$nodeId \mapsto x] : x \in (offlineDestinations \cup new\_connections)\}$

$acksToRemove \triangleq$
$\{ack \in node.waiting\_acks : \land ack.type = \text{“ack\_parent”}$
$\land ack.nodeId \in nodesFailed\}$
$\cup \{ackToRemove\}$

$updated\_log \triangleq$

```
        RemoveHierarchyUpdatesOfNodeFromLog(node.log,
                                            (DOMAIN nodesToAdd) ∪ nodesFailed)

    new_config ≜
        [configuration EXCEPT
            ![n].vv = updated_vv,
            ![n].datastore = DS,
            ![n].waiting_acks = (@ ∪ new_waiting_acks) \ acksToRemove,
            ![n].log = updated_log ∘ msgsToPropagate]
IN
    ∧ configuration' = new_config
    ∧ msgs' = final_msgs
    ∧ failedNodes' = FailuresNotHandled(new_config)
    ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)

ParentFailDetected(n, parent, ackToRemove) ≜
LET
    node ≜ configuration[n]
    vv ≜ node.vv

    nodesFailed ≜ GetNodesFailed(vv, {parent}, {}, n, NullNodeId)

    updated_vv ≜
        [RemoveNodesFromVV(vv, nodesFailed, NullVVEntry) EXCEPT
            ![n].executedOperations = @ + Cardinality(nodesFailed)]

    Messages to new parent informing about failed nodes

    msgPS ≜ [msgType ↦ "package_start", sourceId ↦ n]
    msgPE ≜ [msgType ↦ "package_end", sourceId ↦ n]

    setFailMsgsToPropagate ≜ {[msgType ↦ "node_failed",
                               nodeId ↦ nId,
                               sourceId ↦ n] : nId ∈ nodesFailed}

    failMsgsToPropagate ≜ SetToSequence(setFailMsgsToPropagate, ⟨⟩)

    finalFailedNodesMsgs ≜
        IF Len(failMsgsToPropagate) > 1
        THEN Append(⟨msgPS⟩ ∘ failMsgsToPropagate, msgPE)
        ELSE failMsgsToPropagate

    new_parent ≜ updated_vv[n].parent

    destinations ≜ {new_parent}

    offlineDestinations ≜
        {x ∈ destinations :
```

```
                configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
            [nId ∈ destinations ↦
                        IF nId = new_parent
                         THEN [msgs ↦ finalFailedNodesMsgs,
                               priority ↦ TRUE]
                         ELSE ⟨⟩]

        new_msgs ≜
            SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

        node_messages ≜
            RemoveAddsHierarchyUpdatesOfNodeFromMsgs(new_msgs[n],
                                                     {ackToRemove.nodeId})

        final_msgs ≜
            [new_msgs EXCEPT ![n] = node_messages]

        new_waiting_acks ≜
            {[nodeId ↦ x,
              type ↦ "correct_hierarchy"] : x ∈ (offlineDestinations
                                                 ∪ {new_parent})}

        new_config ≜
            [configuration EXCEPT
                ![n].waiting_acks = (@ ∪ new_waiting_acks) \ {ackToRemove}]
    IN
        ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
        ∧ configuration' = new_config
        ∧ msgs' = final_msgs
        ∧ UNCHANGED failedNodes


InformProbableParent(n, nodeFailed, ackToRemove, destination) ≜
    LET
        node ≜ configuration[n]

        msgToChildren ≜ [msgType ↦ "node_failed",
                         nodeId ↦ nodeFailed,
                         sourceId ↦ n]

        offlineDestinations ≜
            {x ∈ destination :
                  configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
```

```
        [nId ∈ destination ↦
                [msgs ↦ ⟨msgToChildren⟩,
                 priority ↦ TRUE]]

    new_waiting_acks ≜
        {[nodeId ↦ x,
          type ↦ "correct_hierarchy"] : x ∈ offlineDestinations}

    temp_msgs ≜ SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

    new_config ≜
        [configuration EXCEPT
            ![n].waiting_acks = (@ ∪ new_waiting_acks) \ {ackToRemove}]

    new_msgs ≜ temp_msgs
IN
    ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
    ∧ failedNodes' = FailuresNotHandled(new_config)
    ∧ configuration' = new_config
    ∧ msgs' = new_msgs


AckFail(n, nodeFailed, ackToRemove) ≜
LET
    new_config ≜ [configuration EXCEPT ![n].waiting_acks = @ \ {ackToRemove}]
    updated_messages ≜ RemoveHierarchyUpdatesOfNodeFromMsgs(Tail(msgs[n]),
                                                            {nodeFailed})
    new_msgs ≜ [msgs EXCEPT ![n] = updated_messages]
IN
    ∧ msgs' = new_msgs
    ∧ configuration' = new_config
    ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
    ∧ failedNodes' = FailuresNotHandled(new_config)


UnlockChild(n, destination, ackToRemove) ≜
LET
    node ≜ configuration[n]
    vv ≜ node.vv

    Message to the new neighbor to correct the hierarchy

    msgToCorrectHierarchy ≜ [msgType ↦ "correct_hierarchy",
                             nodesFailed ↦ {Head(msgs[n]).nodeId},
                             keysId ↦ GetNodeKeys(n, NullDatastoreEntry),
                             sourceId   ↦ n,
                             sourceVV ↦ vv]
```

```
        Send Messages

        offlineDestinations ≜
            {x ∈ {destination} : configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
            [nId ∈ {destination} ↦ [msgs ↦ ⟨msgToCorrectHierarchy⟩,
                                    priority ↦ TRUE]]

        new_msgs ≜
            SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

        new_waiting_acks ≜
            {[type ↦ "correct_hierarchy", nodeId ↦ destination]}

        new_config ≜
            [configuration EXCEPT
                ![n].waiting_acks = (@ \ {ackToRemove}) ∪ new_waiting_acks]
    IN
        ∧ configuration' = new_config
        ∧ msgs' = new_msgs
        ∧ failedNodes' = FailuresNotHandled(new_config)
        ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)

InformParentOfFailure(n, ackToRemove) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv

        update ≜ Head(msgs[n])
        nodeFailed ≜ update.nodeId
        parentOfFailure ≜ vv[nodeFailed].parent

        parentFail ≜
            configuration[vv[n].parent].status.connectionStatus = "offline"

        msg ≜ [msgType ↦ "node_failed",
               nodeId ↦ nodeFailed, sourceId ↦ n]

        offlineDestinations ≜
            {x ∈ {parentOfFailure} :
                configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
            [nId ∈ {parentOfFailure} ↦ [msgs ↦ ⟨msg⟩, priority ↦ TRUE]]

        temp_waiting_acks ≜
            {[nodeId ↦ x,
```

```
                type ↦ "correct_hierarchy"] : x ∈ offlineDestinations}

        new_waiting_acks ≜
            temp_waiting_acks ∪ IF parentFail
                                THEN {[nodeId ↦ parentOfFailure,
                                       type ↦ "correct_hierarchy"]}
                                ELSE {}

        new_msgs ≜ SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

        new_config ≜
            [configuration EXCEPT
                ![n].waiting_acks = (@ \ {ackToRemove}) ∪ new_waiting_acks]
    IN
        ∧ configuration' = new_config
        ∧ msgs' = new_msgs
        ∧ failedNodes' = FailuresNotHandled(new_config)
        ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)

NodeFailed(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv

        update ≜ Head(msgs[n])
        nodeFailed ≜ update.nodeId
        sourceId ≜ update.sourceId

        msgsRemovingNode ≜ MsgsRemovingNode(msgs[n], nodeFailed)
        ackToRemove ≜ [nodeId ↦ nodeFailed, type ↦ "correct_hierarchy"]

        nodesAboveInHierarchy ≜
            GetNodesAboveInHierarchy(vv, vv[n].parent, NullNodeId)

        childRelatedFail ≜
            IF ∧ vv[nodeFailed] ≠ NullVVEntry
               ∧ nodeFailed ∉ nodesAboveInHierarchy
               ∧ nodeFailed ∉ vv[n].childrenId
             THEN {GetChildRelated(vv, nodeFailed, vv[n].childrenId)}
             ELSE {}

        childRelatedSource ≜
            IF ∧ vv[sourceId] ≠ NullVVEntry
               ∧ sourceId ≠ n
               ∧ sourceId ∉ nodesAboveInHierarchy
               ∧ sourceId ∉ vv[n].childrenId
             THEN {GetChildRelated(vv, sourceId, vv[n].childrenId)}
             ELSE {}
```

```
childWithNoKeys ≜
    {c ∈ vv[n].childrenId :
              GetKeysChildIsInterested(n, c, KeyId,
                                       NullDatastoreEntry) = {}}

childrenFailed ≜
    {c ∈ vv[n].childrenId :
              configuration[c].status.connectionStatus ≠ "online"}
IN
    IF Len(msgsRemovingNode) > 0
     THEN IgnoreMessage([node EXCEPT !.waiting_acks = @ \ {ackToRemove}])

     ELSE IF ∧ vv[nodeFailed] = NullVVEntry
             ∧ childrenFailed = {}
             ∧ childWithNoKeys = {}
             ∧ ∨ n = sourceId
               ∨ vv[sourceId] = NullVVEntry
               ∨ ∃ ack ∈ node.waiting_acks : ∧ ack.type = "correct_hierarchy"
                                            ∧ ack.nodeId = sourceId
          Fail was already handled, the message and acknowledge are removed
     THEN AckFail(n, nodeFailed, ackToRemove)

     ELSE IF ∧ vv[nodeFailed] = NullVVEntry
             ∧ n ≠ sourceId
             ∧ vv[sourceId] ≠ NullVVEntry
             ∧ sourceId ∈ vv[n].childrenId
     THEN UnlockChild(n, sourceId, ackToRemove)

     ELSE IF vv[n].parent = nodeFailed
     THEN ParentFailDetected(n, nodeFailed, ackToRemove)

     ELSE IF nodeFailed ∈ nodesAboveInHierarchy
     THEN InformParentOfFailure(n, ackToRemove)


     ELSE IF nodeFailed ∈ vv[n].childrenId
     THEN HandleFailure(n, ackToRemove, nodeFailed, sourceId)

     ELSE IF ∧ nodeFailed ∉ vv[n].childrenId
             ∧ vv[nodeFailed] ≠ NullVVEntry
             ∧ nodeFailed ∉ nodesAboveInHierarchy
     THEN IF ∀ x ∈ childRelatedFail :
                   configuration[x].status.connectionStatus = "online"
          THEN InformProbableParent(n, nodeFailed, ackToRemove, childRelatedFail)
          ELSE HandleFailure(n, ackToRemove,
                             CHOOSE x ∈ childRelatedFail : TRUE,
                             sourceId)
```

```
        ELSE IF ∧ vv[nodeFailed] = NullVVEntry
                ∧ vv[sourceId] ≠ NullVVEntry
                ∧ childWithNoKeys = {}
                ∧ sourceId ∉ nodesAboveInHierarchy
        THEN IF ∀ x ∈ childRelatedSource :
                    configuration[x].status.connectionStatus = "online"
             THEN InformProbableParent(n, nodeFailed, ackToRemove, childRelatedSource)
             ELSE HandleFailure(n, ackToRemove,
                                CHOOSE x ∈ childRelatedSource : TRUE,
                                sourceId)

        ELSE IF ∧ vv[nodeFailed] = NullVVEntry
                ∧ childWithNoKeys ≠ {}
        THEN UnlockChild(n, CHOOSE c ∈ childWithNoKeys : TRUE, ackToRemove)

        ELSE IF childrenFailed ≠ {}
        THEN HandleFailure(n, ackToRemove, CHOOSE x ∈ childrenFailed : TRUE, sourceId)

        ELSE AckFail(n, nodeFailed, ackToRemove)

ReceiveAckHierarchyCorrected(n) ≜
    LET
        node ≜ configuration[n]
        vv ≜ node.vv

        update ≜ Head(msgs[n])
        sourceId ≜ update.sourceId
        sourceVV ≜ update.sourceVV

        updated_vv ≜
            UpdateVV(n, vv, sourceId, sourceVV, vv[n], NullVVEntry, NullNodeId)

        ackToRemove ≜
            {ack ∈ node.waiting_acks : ∧ ack.type = "correct_hierarchy"
                                       ∧ ack.nodeId = sourceId}
    IN
        ∧ configuration' = [configuration EXCEPT
                                ![n].vv = updated_vv,
                                ![n].waiting_acks = @ \ ackToRemove]
        ∧ msgs' = [msgs EXCEPT ![n] = Tail(@)]
        ∧ UNCHANGED ⟨offlineNodes, failedNodes⟩

This operation will trigger different actions depending if it was sent by the parent or by a child
ProcessMsgCorrectHierarchy(n) ≜
    LET
        node ≜ configuration[n]
```

$vv \triangleq node.vv$

$update \triangleq Head(msgs[n])$
$sourceId \triangleq update.sourceId$
$sourceVV \triangleq update.sourceVV$
$sourceKeys \triangleq update.keysId$

$nodesAboveInH \triangleq$
  $GetNodesAboveInHierarchy(vv,\ vv[n].parent,\ NullNodeId)$

$nodesAboveOfSource \triangleq$
  $GetNodesAboveInHierarchy(sourceVV,$
    $sourceVV[sourceId].parent,$
    $NullNodeId)$

$sourceIsHigherInHierarchy \triangleq$
  $\lor \land vv[sourceId] \neq NullVVEntry$
    $\land sourceId \in nodesAboveInH$
  $\lor \land vv[sourceId] = NullVVEntry$
    $\land n \notin nodesAboveOfSource$

$unknownNodesFailed \triangleq \{nId \in update.nodesFailed : vv[nId] = NullVVEntry\}$

$tmpRmv \triangleq \{nId \in update.nodesFailed :$
    $\land vv[nId] \neq NullVVEntry$
    $\land nId \in nodesAboveInH\}$

$nodesToRemove \triangleq$
  IF $sourceIsHigherInHierarchy$
  THEN $tmpRmv \cup (nodesAboveInH \setminus GetNodesAboveInHierarchy(sourceVV,$
      $n,$
      $NullNodeId))$
  ELSE $\{\}$

$nodesToRemoveAfter \triangleq$
  IF $sourceIsHigherInHierarchy$
  THEN $((update.nodesFailed \setminus unknownNodesFailed) \setminus nodesToRemove)$
  ELSE $\{\}$

$temp1\_vv \triangleq$ IF $nodesToRemove = \{\}$
    THEN vv
    ELSE $RemoveNodesFromVV(vv,\ nodesToRemove,\ NullVVEntry)$

If the node 'n' does not know the source, it must added it because the source will become its new parent

$addSource \triangleq temp1\_vv[sourceId] = NullVVEntry \land sourceIsHigherInHierarchy$

```
temp2_vv ≜
    IF addSource
     THEN [nId ∈ NodeId ↦
             IF nId ∈ nodesAboveOfSource ∪ {sourceId}
              THEN IF temp1_vv[nId] = NullVVEntry
                     THEN [sourceVV[nId] EXCEPT !.executedOperations = 0]
                     ELSE [sourceVV[nId] EXCEPT
                             !.executedOperations = temp1_vv[nId].executedOperations]

              ELSE IF nId ∈ nodesAboveInH
              THEN NullVVEntry

              ELSE IF nId = n
              THEN [temp1_vv[nId] EXCEPT !.parent = sourceId]
              ELSE temp1_vv[nId]]
     ELSE
         temp1_vv


numOpExecuted ≜
    Cardinality(nodesToRemove) + IF addSource THEN 1 ELSE 0

updated_nodeInfo ≜
    [temp2_vv[n] EXCEPT !.executedOperations = @ + numOpExecuted]


final_vv ≜
    IF sourceIsHigherInHierarchy
     THEN [temp2_vv EXCEPT ![n] = updated_nodeInfo]
     ELSE UpdateVV(n, temp2_vv, sourceId, sourceVV, updated_nodeInfo,
                   NullVVEntry, NullNodeId)
```

Select messages to re-propagate

```
numMsgsSourceKnowns ≜ Max(sourceVV[n].executedOperations, node.status.numOpOnline)

numExecutedOperations ≜ vv[n].executedOperations

msgsMissing ≜ SubSeq(node.log, numMsgsSourceKnowns + 1, numExecutedOperations)


newKeyMsgsMissing ≜ SelectSeq(msgsMissing, IsNewKey)

DS ≜ IF ∨ sourceIsHigherInHierarchy
        ∨ ∧ sourceIsHigherInHierarchy = FALSE
          ∧ ∨ final_vv[sourceId] = NullVVEntry
            ∨ sourceId ∉ final_vv[n].childrenId
     THEN node.datastore
     ELSE [k ∈ KeyId ↦ LET key ≜ node.datastore[k]
```

```
                                IN
                                    IF  ∨ key = NullDatastoreEntry
                                        ∨ k ∉ sourceKeys
                                     THEN key
                                     ELSE [key EXCEPT
                                               !.childInterested = sourceId]]

final_datastore ≜
    IF sourceIsHigherInHierarchy
     THEN DS
     ELSE UpdateDSWithNewKey(DS,
                             newKeyMsgsMissing,
                             {sourceId},
                             sourceVV,
                             NullVVEntry,
                             NullNodeId)

msgsMissingIfChild ≜
    IF sourceIsHigherInHierarchy
     THEN ⟨⟩
     ELSE SelectSeqToChildren(msgsMissing,
                              final_datastore,
                              final_vv,
                              sourceVV,
                              [x ∈ {sourceId} ↦ ⟨⟩],
                              NullVVEntry,
                              ⟨⟩,
                              nodesAboveInH)

msgsToRepeat1 ≜
    IF sourceIsHigherInHierarchy
     THEN SetToSequence({[msgType ↦ "node_failed",
                          nodeId ↦ x,
                          sourceId ↦ n] : x ∈ nodesToRemoveAfter}, ⟨⟩)
     ELSE SelectSeqOfNewNodes(msgsMissing, final_vv, NullVVEntry)

msgsToRepeat ≜
    IF Len(msgsToRepeat1) > 0 ∧ sourceIsHigherInHierarchy = FALSE
     THEN Append(msgsToRepeat1, [msgType ↦ "repeat_now", sourceId ↦ n])
     ELSE msgsToRepeat1
```

Final messages to source

```
tmpRC ≜ SelectSeq(node.log, IsRemoveChild)
tmpMsgRC ≜
```

```
    IF tmpRC = ⟨⟩
     THEN ⟨⟩
     ELSE ⟨[tmpRC[1] EXCEPT
                    !.newChildren =
                        [nId    ∈ NodeId ↦
                                IF nId ∈ final_vv[n].childrenId
                                 THEN GetKeysChildIsInterested(n,
                                                               nId,
                                                               KeyId,
                                                               NullDatastoreEntry)
                                 ELSE {}],
                    !.sourceVV = vv]⟩

tempMsgsToSource ≜
    IF sourceIsHigherInHierarchy
     THEN SelectSeqToParent(msgsMissing, sourceVV, n, NullVVEntry,
                            KeyId, NullDatastoreEntry) ∘ tmpMsgRC
     ELSE msgsMissingIfChild[sourceId]

msgsToSource ≜ RemoveOpRemoveNodeOfFailures(tempMsgsToSource, update.nodesFailed)

ackToSend ≜ IF sourceIsHigherInHierarchy
             THEN [msgType ↦ "correct_hierarchy",
                   nodesFailed ↦ update.nodesFailed,
                   keysId ↦ GetNodeKeys(n, NullDatastoreEntry),
                   sourceId   ↦ n,
                   sourceVV ↦ final_vv]
             ELSE [msgType ↦ "ack_hierachy_corrected",
                   sourceId   ↦ n,
                   sourceVV ↦ final_vv]


msgsToSourcet2 ≜ Append(msgsToSource, ackToSend)

msgPS ≜ [msgType ↦ "package_start", sourceId ↦ n]
msgPE ≜ [msgType ↦ "package_end", sourceId ↦ n]

finalMsgsToSource ≜ IF Len(msgsToSourcet2) > 1
                    THEN Append(⟨msgPS⟩ ∘ msgsToSourcet2, msgPE)
                    ELSE msgsToSourcet2
```

Messages informing about removed and added nodes

```
setMsgsToRemoveNodes ≜ {[msgType ↦ "remove_node",
                         nodeId ↦ nId,
                         sourceVV ↦ final_vv,
                         sourceId   ↦ n] : nId ∈ nodesToRemove}
```

$setMsgsToAddNodes \triangleq$ IF $addSource$
THEN $\{[msgType \mapsto$ "add_node_to_hierarchy",
$nodeId \mapsto sourceId,$
$sourceVV \mapsto final\_vv,$
$sourceId \mapsto n]\}$
ELSE $\{\}$

$updated\_log \triangleq$
$RemoveHierarchyUpdatesOfNodeFromLog(node.log,$
$nodesToRemove \cup$ IF $addSource$
THEN $\{sourceId\}$
ELSE $\{\})$

$setMsgsToPropagate \triangleq setMsgsToRemoveNodes \cup setMsgsToAddNodes$

$msgsToPropagate \triangleq SetToSequence(setMsgsToPropagate, \langle\rangle)$

$finalMsgsToPropagate \triangleq$ IF $Len(msgsToPropagate) > 1$
THEN $Append(\langle msgPS \rangle \circ msgsToPropagate, msgPE)$
ELSE $msgsToPropagate$

Send Messages
$destinations \triangleq \{sourceId, n\} \cup$ IF $sourceIsHigherInHierarchy$
THEN $final\_vv[n].childrenId$
ELSE $\{final\_vv[n].parent\} \setminus \{NullNodeId\}$

$offlineDestinations \triangleq$
$\{x \in destinations : configuration[x].status.connectionStatus =$ "offline"$\}$

$msgsToSend \triangleq$
$[nId \in destinations \mapsto$
IF $nId = sourceId$
THEN $[msgs \mapsto finalMsgsToSource, priority \mapsto$ TRUE$]$
ELSE IF $nId = n$
THEN $[msgs \mapsto msgsToRepeat, priority \mapsto$ TRUE$]$
ELSE $[msgs \mapsto finalMsgsToPropagate, priority \mapsto$ FALSE$]]$

$new\_msgs \triangleq SendMessages(msgs, n, msgsToSend, offlineDestinations,$ TRUE$)$

$temp\_setAcks \triangleq offlineDestinations \cup nodesToRemoveAfter$

$temp1\_acks \triangleq$
$node.waiting\_acks \cup \{[type \mapsto$ "correct_hierarchy",
$nodeId \mapsto x] : x \in temp\_setAcks\}$

$temp2\_acks \triangleq$ IF $Len(msgsToRepeat) > 0 \land sourceIsHigherInHierarchy =$ FALSE
THEN $temp1\_acks \cup \{[nodeId \mapsto n, type \mapsto$ "repeat_now"$]\}$

```
                    ELSE  temp1_acks

        ack  ≜  [nodeId ↦ sourceId, type ↦ "correct_hierarchy"]

        acksToRemove  ≜
            {ack} ∪ {ak ∈ node.waiting_acks :
                                ∧ ak.type = "correct_hierarchy"
                                ∧ ∨ vv[ak.nodeId] = NullVVEntry
                                  ∨ ak.nodeId ∈ GetNodesAboveInHierarchy(vv,
                                                                         vv[n].parent,
                                                                         NullNodeId)}

        new_config  ≜
            [configuration EXCEPT
                        ![n].vv = final_vv,
                        ![n].datastore  = final_datastore,
                        ![n].waiting_acks =
                                IF sourceIsHigherInHierarchy
                                 THEN (temp2_acks \ acksToRemove) ∪ {ack}
                                 ELSE  temp2_acks \ acksToRemove,
                        ![n].log = IF sourceIsHigherInHierarchy
                                     THEN updated_log ∘ msgsToPropagate
                                     ELSE @]

        final_msgs  ≜
            [new_msgs EXCEPT ![n] =
                    RemoveHierarchyUpdatesOfNodeFromMsgs(@,
                                                         unknownNodesFailed ∪ nodesToRemove)]
    IN
        ∧ msgs' = final_msgs
        ∧ configuration' = new_config
        ∧ offlineNodes' = GetOfflineNodes(new_config, final_msgs, NullVVEntry)
        ∧ failedNodes' = FailuresNotHandled(new_config)


ReceivedMsgCorrectHierarchy(n)  ≜
    LET
        node  ≜  configuration[n]
        vv  ≜  node.vv

        update  ≜  Head(msgs[n])
        sourceId  ≜  update.sourceId

        ack  ≜  [type ↦ "correct_hierarchy", nodeId ↦ update.sourceId]

        downstream  ≜  [msgType ↦ "node_failed",
                        nodeId ↦ update.sourceId,
```

```
                    sourceId ↦ n]

        destinations ≜ {update.sourceVV[update.sourceId].parent}
        offlineDestinations ≜
            {x ∈ destinations : configuration[x].status.connectionStatus = "offline"}

        msgsToSend ≜
            [x ∈ destinations ↦ [msgs ↦ ⟨downstream⟩, priority ↦ TRUE]]

        new_msgs ≜ SendMessages(msgs, n, msgsToSend, offlineDestinations, TRUE)

        new_config ≜ [configuration EXCEPT ![n].waiting_acks = @ \ {ack}]
    IN
        IF ∧ configuration[sourceId].status.connectionStatus ≠ "online"
           ∧ vv[sourceId] = NullVVEntry

         THEN ∧ configuration' = new_config
              ∧ msgs' = new_msgs
              ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
              ∧ failedNodes' = FailuresNotHandled(new_config)
         ELSE ProcessMsgCorrectHierarchy(n)


Fail(n) ≜
    LET
        node ≜ configuration[n]
        datastore ≜ node.datastore

        updated_datastore ≜
            [k ∈ KeyId ↦ IF datastore[k] = NullDatastoreEntry
                         THEN NullDatastoreEntry
                         ELSE [datastore[k] EXCEPT !.childInterested = NullNodeId]]


        destinations ≜ GetSourcesOfMessages(msgs[n]) \ {n}
        msgsToSend ≜
            [nId ∈ destinations ↦ [msgs ↦ ⟨[msgType ↦ "node_failed",
                                            nodeId ↦ n,
                                            sourceId ↦ n]⟩, priority ↦ TRUE]]

        new_msgs ≜
            [SendMessages(msgs, n, msgsToSend, {}, FALSE) EXCEPT ![n] = ⟨⟩]

        update_vv ≜
            [nId ∈ NodeId ↦
                IF nId = n
                 THEN [node.vv[nId] EXCEPT !.parent = NullNodeId,
                                           !.childrenId = {}]
                 ELSE NullVVEntry]
```

```
    new_config ≜
        [nId ∈ NodeId ↦
            IF nId = n
            THEN [configuration[nId] EXCEPT
                    !.vv = update_vv,
                    !.datastore  = updated_datastore,
                    !.waiting_acks = {},
                    !.status = [connectionStatus ↦ "offline",
                                numOpOnline ↦ node.vv[n].executedOperations]]
            ELSE IF nId ∈ destinations
            THEN [configuration[nId] EXCEPT
                    !.waiting_acks = (@ ∪ {[nodeId ↦ n,
                                            type ↦ "correct_hierarchy"]})
                                        \ {[nodeId ↦ n,
                                            type ↦ "ack_parent"]}]
            ELSE configuration[nId]]
    IN
        ∧ NoRemoveAck(msgs[n])
        ∧ ∀ ack ∈ node.waiting_acks :
                ∧ ack.type  ≠ "offline_operations"
                ∧ ack.type  ≠ "ack_remove"
        ∧ node.status.connectionStatus = "online"
        ∧ node.vv[n].parent ≠ NullNodeId
        ∧ msgs' = new_msgs
        ∧ configuration' = [new_config EXCEPT ![n].log =  RemoveRemoveChildMessage(@)]
        ∧ offlineNodes' = GetOfflineNodes(new_config, new_msgs, NullVVEntry)
        ∧ failedNodes' = failedNodes ∪ {n}
```

OFFLINE FUNCTIONS

An *offline* node can add and remove keys from the *datastore* to simulate different nodes

```
RemoveKeyOffline(n, k) ≜
    ∧ configuration[n].status.connectionStatus = "offline"
    ∧ configuration[n].datastore[k] ≠ NullDatastoreEntry
    ∧ ∃ kId ∈ KeyId \ {k} :
            configuration[n].datastore[kId] ≠ NullDatastoreEntry
    ∧ configuration' = [configuration EXCEPT
                            ![n].datastore  = [@ EXCEPT ![k] = NullDatastoreEntry],
                            ![n].vv = [@ EXCEPT !.executedOperations = @ + 1],
                            ![n].log = Append(@, [msgType ↦ "datastore_change"])]
    ∧ UNCHANGED ⟨offlineNodes, msgs, failedNodes⟩

AddKeyOffline(n, k) ≜
```

$\land$ $configuration[n].status.connectionStatus$ = "offline"
$\land$ $configuration[n].datastore[k] = NullDatastoreEntry$
$\land \exists\, nId \in NodeId : k \in GetNodeKeys(nId,\ NullDatastoreEntry)$
$\land$ $configuration' = [configuration$ EXCEPT
!$[n].datastore$ = [@ EXCEPT !$[k] = InitKey(k,\ NullNodeId)$],
!$[n].vv$ = [@ EXCEPT !$[n].executedOperations = @ + 1$],
!$[n].log = Append(@,\ [msgType \mapsto$ "datastore_change"])]
$\land$ UNCHANGED $\langle offlineNodes,\ msgs,\ failedNodes\rangle$

An *offline* node can also execute key updates. When a node enters the hierarchy, if it executed updates while *offline*, those updates must be applied. Those *log* entries are placed in the messages sequence, and the *Next* state will keep applying those operations until a message "offline_operations" is received.

$EndOfOfflineOperations(n)$ $\triangleq$
$\land$ $configuration' = [configuration$ EXCEPT
!$[n].waiting\_acks = @ \setminus \{[nodeId \mapsto n,\ type \mapsto$ "offline_operations"]$\}$]
$\land$ UNCHANGED $\langle offlineNodes,\ failedNodes\rangle$
$\land$ $msgs' = [msgs$ EXCEPT !$[n] = Tail(msgs[n])$]

An online node that receives a message will apply it, unless the node is waiting for an acknowledge. If the node is waiting for an acknowledge, it will wait until it receives it. Acknowledges are the only type of messages that is placed in the beggining of the sequence. The only exception is when a node is waiting for an remove acknowledge from its parent, and receives a message informing that it must change its parent. In this case the node must process the messages, until it changes the parent and re-propagate the message "remove_child" to the new parent

$ProcessMessage(n)$ $\triangleq$
$\land$ $Len(msgs[n]) > 0$
$\land$ $configuration[n].status.connectionStatus$ = "online"
$\land$ $\lor$ $configuration[n].waiting\_acks = \{\}$
$\lor \exists\, ack \in configuration[n].waiting\_acks$:
$\lor$ $\land$ $ack.type$ = "ack_remove"
$\land$ $ContainsMsgNewParent(msgs[n])$
$\lor$ $ack.type$ = "receiving_package"
$\lor$ $\land$ $ack.type$ = "repeat_now"
$\land \forall\, ack2 \in configuration[n].waiting\_acks \setminus \{ack\}$:
$ack2.type \neq$ "correct_hierarchy"
$\lor$ $\land$ $ack.type$ = "offline_operations"
$\land$ $configuration[n].waiting\_acks \setminus \{ack\} = \{\}$
$\lor$ $\land$ $ack.type$ = "correct_hierarchy"
$\land$ $\lor$ $\land$ $Head(msgs[n]).msgType$ = "new_parent"
$\land$ $Head(msgs[n]).nodeToRemove = ack.nodeId$
$\lor$ $Head(msgs[n]).msgType$ = "node_failed"

```
  ∨ LET head ≜ Head(msgs[n])
        tail ≜ Tail(msgs[n])
    IN
        ∨ head.msgType = "ack_parent"
        ∨ head.msgType = "ack_remove"
        ∨ head.msgType = "correct_hierarchy"
        ∨ head.msgType = "ack_hierachy_corrected"
        ∨ head.msgType = "node_failed"
        ∨ ∧ head.msgType = "package_start"
          ∧ tail ≠ ⟨⟩
          ∧ ∨ ContainsMsgAckParent(GetPackage(tail))
            ∨ ContainsMsgCorrectingHier(GetPackage(tail))
∧ LET
      msg ≜ Head(msgs[n])
      msgType ≜ msg.msgType
  IN
      IF msgType = "key_update"
       THEN IF ∧ msg.sourceId = n
               ∧ ∃ ack ∈ configuration[n].waiting_acks :
                                          ack.type = "offline_operations"
            THEN PUT(n, msg.keyId, msg.newValue)
            ELSE ReceivedKeyUpdate(n)

       ELSE IF msgType = "new_key_or_update"
       THEN ReceivedNewKeyOrUpdate(n)

       ELSE IF msgType = "new_key"
       THEN CreateKey(n,
                      msg.keyId,
                      msg.value,
                      msg.version,
                      msg.nodesInterested,
                      msg.sourceId,
                      msg.sourceVV)

       ELSE IF msgType = "new_node"
       THEN AddNode(n, msg.nodeId, msg.sourceVV, msg.sourceId)

       ELSE IF msgType = "add_node_to_hierarchy"
       THEN ReceiveNewHierarchyNode(n)

       ELSE IF msgType = "new_parent"
       THEN NewParent(n)

       ELSE IF msgType = "ack_parent"
       THEN ReceiveChildConfirmation(n)
```

ELSE IF $msgType =$ "package_start"
THEN $ReceiveStartOfMessagesPack(n)$

ELSE IF $msgType =$ "package_end"
THEN $ReceiveFinalOfMessagesPack(n,\ Head(msgs[n]))$

ELSE IF $msgType =$ "remove_child"
THEN $RemoveChild(n)$

ELSE IF $msgType =$ "remove_node"
THEN $NodeRemoved(n)$

ELSE IF $msgType =$ "ack_remove"
THEN $ReceiveRemoveAck(n)$

ELSE IF $msgType =$ "repeat_now"
THEN $ReceiveRepeatNow(n)$

ELSE IF $msgType =$ "offline_operations"
THEN $EndOfOfflineOperations(n)$

ELSE IF $msgType =$ "node_failed"
THEN $NodeFailed(n)$

ELSE IF $msgType =$ "correct_hierarchy"
THEN $ReceivedMsgCorrectHierarchy(n)$

ELSE IF $msgType =$ "ack_hierachy_corrected"
THEN $ReceiveAckHierarchyCorrected(n)$

ELSE $IgnoreMessage(configuration[n])$

$Init \triangleq$
LET
$initOfflineNodes \triangleq$ CHOOSE $x \in$ SUBSET $(NodeId) : Cardinality(x) = 1$

$onlineNodeIds \triangleq NodeId \setminus initOfflineNodes$

$newKeys \triangleq$ CHOOSE $nk \in$ SUBSET $(KeyId) : nk \neq KeyId \land Cardinality(nk) = 1$
$initialKeys \triangleq KeyId \setminus newKeys$

$rootId \triangleq$ CHOOSE $rId \in onlineNodeIds :$ TRUE
$k \triangleq$ CHOOSE $k \in initialKeys :$ TRUE
IN
$\land\ offlineNodes = initOfflineNodes$
$\land\ failedNodes = \{\}$

$\land\ msgs = [n \in NodeId \mapsto \langle\rangle]$
$\land\ \exists\, parent \quad \in [onlineNodeIds \to onlineNodeIds \cup \{NullNodeId\}] :$
$\quad \exists\, children \in [onlineNodeIds \to \text{SUBSET}\ (onlineNodeIds \setminus \{rootId\})] :$
$\quad \exists\, keys \in [onlineNodeIds \to \text{SUBSET}\ initialKeys] :$
Root contains all keys and has no parent
$\land\ keys[rootId] = initialKeys$
$\land\ parent[rootId] = NullNodeId$
Validation of nodes
$\land\ \forall\, n \in onlineNodeIds :$
$\land\ n \neq parent[n]$
$\land\ n \notin children[n]$
$\land\ parent[n] \notin children[n]$
$\land\ \forall\, c \quad \in children[n] : parent[c] = n$
$\land\ \lor\ n = rootId$
$\lor\ \land\ parent[n] \neq NullNodeId$
$\land\ n \in children[parent[n]]$
$\land\ keys[n] \subseteq keys[parent[n]]$
$\land\ keys[n] \neq \{\}$
$\land\ \forall\, key \in keys[n] :$
$Cardinality(\{c \in children[n] :$
$key \in keys[c]\}) < 2$

$\land\ configuration = InitState(children,\ keys,\ parent,$
$initialKeys,\ initOfflineNodes)$

$\land\ \text{IF}\ Cardinality(onlineNodeIds) < 3$
$\text{THEN}\ \forall\, n \in onlineNodeIds : k \in keys[n]$
$\text{ELSE}\ \text{LET}$
$otherNodes \triangleq onlineNodeIds \setminus \{rootId\}$
$sec \triangleq \text{CHOOSE}\ sId \in otherNodes : \text{TRUE}$
IN
$\land\ sec \in configuration[rootId].vv[rootId].childrenId$
$\land\ k \in GetNodeKeys(sec,\ NullDatastoreEntry)$

$\land\ \text{IF}\ Cardinality(onlineNodeIds) = 3$
THEN
$\land\ \text{IF}\ Cardinality(configuration[rootId].vv[rootId].childrenId) = 1$
$\text{THEN}\ \forall\, nId \in otherNodes \setminus \{sec\} :$
$k \in GetNodeKeys(nId,\ NullDatastoreEntry)$
$\text{ELSE}\ \text{TRUE}$
$\text{ELSE}\ \text{TRUE}$

$Next \triangleq$
$\text{LET}\ onlineNodes \triangleq$
$\{n \in NodeId : configuration[n].status.connectionStatus = \text{“online”}\}$

```
    nrp ≜
      {n ∈ onlineNodes :
       ∃ ack ∈ configuration[n].waiting_acks :
                    ∨ ack.type = "receiving_package"
                    ∨ ack.type = "repeat_now"
                    ∨ ∧ ack.type = "offline_operations"
                      ∧ configuration[n].waiting_acks \ {ack} = {}}
IN
  IF nrp ≠ {}
   THEN ProcessMessage(CHOOSE n ∈ nrp : TRUE)
   ELSE
       ∨ ∃ n ∈ onlineNodes :
             ∨ ProcessMessage(n)
             ∨ ∧ configuration[n].vv[n].executedOperations < MaxOperations
               ∧ ∨ ∃ k ∈ KeyId, newValue ∈ 33 .. 34 :
                                  ∧ configuration[n].waiting_acks = {}
                                  ∧ PUT(n, k,  newValue)
                 ∨ Remove(n)
                 ∨ Fail(n)

       ∨ ∃ n ∈ offlineNodes, k ∈ KeyId :
             ∧ configuration[n].vv[n].executedOperations < MaxOperations
             ∧ ∨ ∃ newValue ∈ 33 .. 34 :
                          ∧ configuration[n].datastore[k] ≠ NullDatastoreEntry
                          ∧ PUT(n, k,  newValue)
               ∨ RemoveKeyOffline(n, k)

       ∨ ∧ offlineNodes ≠ {}
         ∧ LET
             rootId ≜ CHOOSE n ∈ onlineNodes :
                            configuration[n].vv[n].parent = NullNodeId

             nOff ≜ CHOOSE n ∈ offlineNodes : TRUE
           IN
             ∧ configuration[rootId].waiting_acks = {}
             ∧ configuration[rootId].vv[rootId].executedOperations < MaxOperations
             ∧ AddNode(rootId, nOff, ⟨⟩, rootId)

       ∨ ∧ ∀ n ∈ onlineNodes : msgs[n] = ⟨⟩ ∧ configuration[n].waiting_acks = {}
         ∧ UNCHANGED ⟨configuration, msgs, offlineNodes, failedNodes⟩
```

Invariants

$TypeInvariant \triangleq$
$configuration \in [NodeId \to State]$

$Msgs\_Log\_Invariant \triangleq$
$\land msgs \in [NodeId \to Seq(MsgKeyUpdate \cup MsgUpdateUnknownKey \cup MsgNewKey \cup MsgNewNode$
$\cup MsgNewParent \cup MsgHierarchyChange$
$\cup MsgAckNewParent \cup MsgPackegesAndRemoveAck$
$\cup MsgRemoveChild \cup LogAutoRemove$
$\cup MsgNodeFailed \cup MsgCorrectHierarchy)]$

$\land \forall n \in NodeId : configuration[n].vv[n].executedOperations = Len(configuration[n].log)$

$Causality \triangleq$
$\forall n \in NodeId :$
LET $log \triangleq configuration[n].log$
$lastOp \triangleq Len(log)$
IN IF $lastOp < 2$
THEN TRUE
ELSE $ContainsKeyUpdateBreakingCausality(log[lastOp],$
$SubSeq(log,$
$1,$
$lastOp - 1)) =$ FALSE

If there are no messages of key update to be processed, then a key must have the same value on every node

$Convergence \triangleq$
LET
$onlineNs \triangleq \{n \in NodeId : configuration[n].status.connectionStatus =$ "online"$\}$
$rootId \triangleq$ CHOOSE $x \in onlineNs : configuration[x].vv[x].parent = NullNodeId$
$rootDS \triangleq configuration[rootId].datastore$
IN
$\lor failedNodes \neq \{\}$
$\lor \exists n \in onlineNs : ContainsMsgAffectingConvergence(msgs[n])$
$\lor \forall k \in KeyId :$
$\lor \exists n \in onlineNs : ContainsMsgToUpdateKey(msgs[n],\ k)$
$\lor \forall n \in onlineNs :$
$\lor configuration[n].datastore[k] = NullDatastoreEntry$
$\lor configuration[n].datastore[k].versionId = rootDS[k].versionId$

$vars \triangleq \langle configuration,\ msgs,\ offlineNodes,\ failedNodes \rangle$

$Spec \triangleq Init \land \Box[Next]_{vars}$

THEOREM $Spec \Rightarrow TypeInvariant \land Msgs\_Log\_Invariant \land Causality \land Convergence \land ValidHierarchy$

* Modification History
* Last modified *Thu Sep* 24 00:31:16 *BST* 2015 by *Miguel*
* Created *Tue Apr* 07 16:19:31 *BST* 2015 by *Miguel*

MODULE *LocalOperations*

EXTENDS *Naturals*, *Sequences*, *Integers*, *TLC*, *FiniteSets*

VARIABLES
configuration, *msgs*, *offlineNodes*, *failedNodes*

GENERAL AUXILIARY FUNCTIONS

$Max(n, m) \triangleq$ IF $n > m$ THEN n ELSE m

$Min(n, m) \triangleq$ IF $n < m$ THEN n ELSE m

RECURSIVE $GetNodesAboveInHierarchy(\_, \_, \_)$
$GetNodesAboveInHierarchy(vv, parent, null) \triangleq$
IF $parent = null \lor vv[parent].nodeId = null$
THEN $\{\}$
ELSE IF $vv[parent].parent = null$
THEN $\{parent\}$
ELSE $\{parent\} \cup GetNodesAboveInHierarchy(vv, vv[parent].parent, null)$

RECURSIVE $SetToSequence(\_, \_)$
$SetToSequence(set, seq) \triangleq$
IF $set = \{\}$
THEN seq
ELSE IF $Cardinality(set) = 1$
THEN $Append(seq,$ CHOOSE $x \in set :$ TRUE$)$
ELSE
LET $item \triangleq$ CHOOSE $x \in set :$ TRUE
$newSet \triangleq set \setminus \{item\}$
$newSeq \triangleq Append(seq, item)$
IN
$SetToSequence(newSet, newSeq)$

AUXILIAR FUNCTIONS TO HIERARCHY VERIFICATION

$GetNodeKeys(nodeId, NullDatastoreEntry) \triangleq$
LET
$datastore \triangleq configuration[nodeId].datastore$
$KeyId \triangleq$ DOMAIN $datastore$
IN
$\{nodeKey \in KeyId : datastore[nodeKey] \neq NullDatastoreEntry\}$

$$
\begin{array}{l}
GetKeysChildIsInterested(nodeId,\ childId,\ KeyId,\ NullDatastoreEntry) \triangleq \\
\quad \text{LET} \\
\qquad datastore \triangleq configuration[nodeId].datastore \\
\quad \text{IN} \\
\qquad \{nodeKey \in KeyId : \ \wedge\ datastore[nodeKey] \neq NullDatastoreEntry \\
\qquad\qquad\qquad\qquad\ \ \wedge\ datastore[nodeKey].childInterested = childId\}
\end{array}
$$

$$
\begin{array}{l}
IsParentOfChild(parentId,\ childId) \triangleq \\
\quad \text{LET} \\
\qquad parent \triangleq configuration[parentId] \\
\quad \text{IN} \\
\qquad childId \in parent.vv[parentId].childrenId
\end{array}
$$

$$
\begin{array}{l}
IsChildOfFather(parentId,\ childId) \triangleq \\
\quad \text{LET} \\
\qquad child \triangleq configuration[childId] \\
\quad \text{IN} \\
\qquad child.vv[childId].parent = parentId
\end{array}
$$

$$
\begin{array}{l}
IsSingleRoot(rootId,\ NullNodeId) \triangleq \\
\quad \text{LET} \\
\qquad otherNodes \triangleq (\text{DOMAIN}\ configuration) \setminus \{rootId\} \\
\quad \text{IN} \\
\qquad \forall\, nodeId \in otherNodes : \ \vee\ configuration[nodeId].vv[nodeId].parent \neq NullNodeId \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \vee\ configuration[nodeId].status.connectionStatus = \text{“offline”} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ \vee\ configuration[nodeId].status.connectionStatus = \text{“pending”}
\end{array}
$$

$$
\begin{array}{l}
ContainsMsgChangingHier(s) \triangleq \\
\quad \exists\, i \in 1\,..\,Len(s) : \ \vee\ s[i].msgType = \text{“ack\_remove”} \\
\qquad\qquad\qquad\qquad\qquad\ \ \vee\ s[i].msgType = \text{“new\_parent”} \\
\qquad\qquad\qquad\qquad\qquad\ \ \vee\ s[i].msgType = \text{“new\_key”} \\
\qquad\qquad\qquad\qquad\qquad\ \ \vee\ s[i].msgType = \text{“node\_failed”} \\
\qquad\qquad\qquad\qquad\qquad\ \ \vee\ s[i].msgType = \text{“correct\_hierarchy”} \\
\qquad\qquad\qquad\qquad\qquad\ \ \vee\ s[i].msgType = \text{“ack\_hierachy\_corrected”}
\end{array}
$$

FUNCTIONS TO CHECK *OFFLINE* NODES, AND NODES THAT HAVE FAILED

$$
\begin{array}{l}
GetNonEmptyDomains(dom) \triangleq \\
\quad \{n \in \text{DOMAIN}\ dom : dom[n] \neq \{\}\}
\end{array}
$$

$$
\begin{array}{l}
IsMsgAffectingNode(msg,\ n) \triangleq \\
\quad \vee\ \wedge\ msg.msgType \neq \text{“new\_parent”} \\
\qquad\ \wedge\ msg.msgType \neq \text{“offline\_operations”} \\
\qquad\ \wedge\ msg.sourceId = n
\end{array}
$$

```
    ∨ ∧ ∨ msg.msgType = "new_node"
        ∨ msg.msgType = "add_node_to_hierarchy"
        ∨ msg.msgType = "remove_node"
        ∨ msg.msgType = "node_failed"
      ∧ msg.nodeId = n
    ∨ ∧ msg.msgType = "new_parent"
      ∧ msg.parent = n ∨ msg.nodeToRemove = n
    ∨ ∧ msg.msgType = "remove_child"
      ∧ n ∈ GetNonEmptyDomains(msg.newChildren)
    ∨ ∧ msg.msgType = "correct_hierarchy"
      ∧ ∨ n = msg.sourceId
        ∨ n ∈ msg.nodesFailed
    ∨ ∧ msg.msgType = "ack_hierachy_corrected"
      ∧ n = msg.sourceId

NoMessagesAffectingNode(s, n) ≜
  ∀ i ∈ 1 .. Len(s) : IsMsgAffectingNode(s[i], n) = FALSE
```

To avoid conflicts, a node is only added to the set of *offline* nodes after being removed by all nodes.

```
GetOfflineNodes(config, newMsgs, NullVVEntry) ≜
  LET
    nodesToCheck ≜ DOMAIN configuration \ offlineNodes

    newOff ≜ {n ∈ nodesToCheck :
                ∧ config[n].status.connectionStatus = "offline"
                ∧ ∀ nId ∈ nodesToCheck \ {n} :
                    ∧ config[nId].vv[n] = NullVVEntry
                    ∧ ∨ config[nId].status.connectionStatus = "offline"
                      ∨ NoMessagesAffectingNode(newMsgs[nId], n)}
  IN
    offlineNodes ∪ newOff

FailuresNotHandled(config) ≜
  LET
    nodes ≜ DOMAIN config
    onlineN ≜ {n ∈ nodes : configuration[n].status.connectionStatus = "online"}
  IN
    {nId ∈ failedNodes :
     ∃ n ∈ onlineN : ∨ config[n].vv[n].parent = nId
                     ∨ nId ∈ config[n].vv[n].childrenId}
```

Updates a version vector of the node 'n' based on the 'sourceVV' and on the node that sent it, node 'sourceId'. 'sourceId' is needed to decide which nodes will be updated by the 'sourceVV', not all nodes will be.

```
UpdateVV(nodeId, vv, sourceId, sourceVV, nodeInformation, NullVVEntry, NullNodeId) ≜
  IF sourceId = nodeId ∨ sourceVV = ⟨⟩
   THEN [vv EXCEPT ![nodeId] = nodeInformation]
   ELSE
     LET
       nodesAbove ≜ GetNodesAboveInHierarchy(vv, vv[nodeId].parent, NullNodeId)

       sourceIsAboveInHierarchy ≜
             ∨ ∧ vv[sourceId] ≠ NullVVEntry
               ∧ sourceId ∈ nodesAbove
             ∨ ∧ vv[sourceId] = NullVVEntry
               ∧ nodeId ∉ GetNodesAboveInHierarchy(sourceVV,
                                                   sourceVV[sourceId].parent,
                                                   NullNodeId)
     IN
       [nodeVV ∈ DOMAIN vv ↦
                 IF nodeVV = nodeId
                  THEN nodeInformation
                  ELSE IF ∧ vv[nodeVV] ≠ NullVVEntry
                          ∧ ∨ ∧ sourceIsAboveInHierarchy
                              ∧ nodeVV ∈ nodesAbove
                            ∨ ∧ sourceIsAboveInHierarchy = FALSE
                              ∧ nodeVV ∉ nodesAbove
                          ∧ sourceVV[nodeVV] ≠ NullVVEntry
                  THEN [vv[nodeVV] EXCEPT !.executedOperations =
                                          Max(vv[nodeVV].executedOperations,
                                          sourceVV[nodeVV].executedOperations)]
                  ELSE vv[nodeVV]]


SendMessages(currentMsgs, n, msgsToSend, offlineDestinations, messageProcessed) ≜
  LET
    destinations ≜ DOMAIN msgsToSend

    waitingAcks ≜ configuration[n].waiting_acks

    unknownNodeFailures ≜ offlineDestinations

    nodeFailureMessages ≜
      SetToSequence({[msgType ↦ "node_failed",
                      nodeId ↦ x,
                      sourceId ↦ n] : x ∈ unknownNodeFailures}, ⟨⟩)

    currentNodeMsgs ≜ IF messageProcessed
```

```
                    THEN Tail(currentMsgs[n])
                    ELSE currentMsgs[n]
IN
  [x ∈ DOMAIN msgs ↦
      IF x = n
       THEN nodeFailureMessages
            ∘ IF x ∈ destinations
              THEN IF msgsToSend[x].priority
                   THEN IF ∧ currentNodeMsgs ≠ ⟨⟩
                           ∧ Head(currentNodeMsgs).msgType = "ack_remove"
                        THEN ⟨Head(currentNodeMsgs)⟩
                             ∘ msgsToSend[x].msgs
                               ∘ Tail(currentNodeMsgs)
                        ELSE msgsToSend[x].msgs ∘ currentNodeMsgs
                   ELSE currentNodeMsgs ∘ msgsToSend[x].msgs
              ELSE currentNodeMsgs
       ELSE IF x ∈ destinations
       THEN IF msgsToSend[x].priority
            THEN IF ∧ currentMsgs[x] ≠ ⟨⟩
                    ∧ Head(currentMsgs[x]).msgType = "ack_remove"
                 THEN ⟨Head(currentMsgs[x])⟩
                      ∘ msgsToSend[x].msgs
                        ∘ Tail(currentMsgs[x])
                 ELSE msgsToSend[x].msgs ∘ currentMsgs[x]
            ELSE currentMsgs[x] ∘ msgsToSend[x].msgs
       ELSE currentMsgs[x]]
```

AUXILIAR FUNCTIONS TO 'PUT' OPERATION

```
GetLogPositionOfVersion(currentKeyVersion, nodeId) ≜
  IF currentKeyVersion.versionNumber = −1 THEN −1
   ELSE IF currentKeyVersion.nodeId = nodeId THEN currentKeyVersion.versionNumber
   ELSE CHOOSE i ∈ 1 .. Len(configuration[nodeId].log) :
                  ∧ ∨ configuration[nodeId].log[i].msgType = "key_update"
                    ∨ configuration[nodeId].log[i].msgType = "new_key"
                  ∧ configuration[nodeId].log[i].version = currentKeyVersion


UpdateIsBreakingCausality(op, version) ≜
  ∧ ∨ op.msgType = "key_update"
    ∨ op.msgType = "new_key"
  ∧ op.version.nodeId = version.nodeId
```

$\land\ op.version.versionNumber > version.versionNumber$

$IsNotBreakingCausality(n,\ version,\ NullDatastoreEntry)\ \triangleq$
LET
$node\ \triangleq\ configuration[n]$
$datastore\ \triangleq\ node.datastore$
$log\ \triangleq\ node.log$
$keys\ \triangleq$ DOMAIN $datastore$
IN
$\land\ \forall\, k \in keys :$
$\lor\ datastore[k] = NullDatastoreEntry$
$\lor\ datastore[k].versionId.versionNumber = -1$
$\lor\ datastore[k].versionId.nodeId \neq version.nodeId$
$\lor\ \land\ datastore[k].versionId.nodeId = version.nodeId$
$\land\ datastore[k].versionId.versionNumber < version.versionNumber$

$\land\ \forall\, i \in 1\,..\,Len(log) : UpdateIsBreakingCausality(log[i],\ version) =$ FALSE

An update should be applied if :
– it is different from the current *update*
– the new update knew the current update OR it didn't knew the current update (conflict detected) and it won the conflict. A version wins a conflict if its creator is above in hierarchy

$IsToApplyUpdate(nodeId,\ vv,\ sourceIsAboveInHierarchy,\ sourceId,\ sourceVV,$
$newVersion,\ currentVersion,\ NullVVEntry,\ NullNodeId,\ NullDSEntry)\ \triangleq$
LET
$creatorOfCurrentVersion\ \triangleq\ currentVersion.nodeId$
$versionNum\ \triangleq\ currentVersion.versionNumber$
$lopPosition\ \triangleq$ IF $versionNum = -1$
THEN -1
ELSE $GetLogPositionOfVersion(currentVersion,\ nodeId)$

$nodesAbove\ \triangleq\ GetNodesAboveInHierarchy(sourceVV,$
$sourceVV[sourceId].parent,$
$NullNodeId)$
IN
$\land\ newVersion \neq currentVersion$
$\land\ IsNotBreakingCausality(nodeId,\ newVersion,\ NullDSEntry)$
Check if this is a new update
$\land\ \lor\ vv[newVersion.nodeId] = NullVVEntry$
$\lor\ newVersion.versionNumber > vv[newVersion.nodeId].executedOperations$
If the key had no previous update
$\land\ \lor\ versionNum = -1$
If the new update knew about the current update
$\lor\ \land\ sourceVV[creatorOfCurrentVersion] \neq NullVVEntry$
$\land\ sourceVV[creatorOfCurrentVersion].executedOperations \geq versionNum$
$\lor\ \land\ sourceVV[nodeId] \neq NullVVEntry$

$\land\ sourceVV[nodeId].executedOperations \geq lopPosition$

If the creator of the current version was lower in the hierarchy when the new update was created

$\lor\ \land\ sourceVV[creatorOfCurrentVersion] \neq NullVVEntry$
$\quad\land\ creatorOfCurrentVersion \notin nodesAbove$

If the new update didn't know the creator of the current update but the update came from a node higher in the hierarchy

$\lor\ \land\ sourceVV[creatorOfCurrentVersion] = NullVVEntry$
$\quad\land\ sourceIsAboveInHierarchy$

Returns set of nodes *Id* of nodes below the 'nodesRelated'. In the first execution of this function, the input 'nodesRelated' will have one element.

RECURSIVE $GetNodesBelow(\_, \_, \_)$
$GetNodesBelow(nodesRelated,\ vv,\ NullVVEntry) \triangleq$
LET
$newRelatedNodes \triangleq \{nId \in \text{DOMAIN } vv :$
$\land\ vv[nId] \neq NullVVEntry$
$\land\ vv[nId].parent \in nodesRelated\} \cup nodesRelated$
$newVV \triangleq [nId \in (\text{DOMAIN } vv) \setminus newRelatedNodes \mapsto vv[nId]]$
IN
IF $newRelatedNodes = nodesRelated$
THEN $newRelatedNodes$
ELSE $GetNodesBelow(newRelatedNodes,\ newVV,\ NullVVEntry)$

A child is interested in a key IF it is in the 'allNodesInterested' OR IF one of its descendents is. The created variable will only have one element. Only one of the children can be interested in the new key

$GetChildInterestedInKey(allNodesInt,\ childrenId,\ vv,\ NullNodeId,\ NullVVEntry) \triangleq$
LET
$setChildrenIntNewKey \triangleq allNodesInt \cap childrenId$
$setChildrenWithDesceIntNewKey \triangleq$
IF $setChildrenIntNewKey \neq \{\}$
THEN $\{\}$
ELSE $\{x \in childrenId :$
$(GetNodesBelow(\{x\},\ vv,\ NullVVEntry) \cap allNodesInt) \neq \{\}\}$
IN
IF $setChildrenIntNewKey \neq \{\}$
THEN CHOOSE $n \in setChildrenIntNewKey$: TRUE
ELSE IF $setChildrenWithDesceIntNewKey \neq \{\}$
THEN CHOOSE $n \in setChildrenWithDesceIntNewKey$: TRUE
ELSE $NullNodeId$

AUXILIAR FUNCTIONS TO 'ADD *NODE'*

```
RECURSIVE GetChildren(_)
GetChildren(newNodesInfo) ≜
    LET
        nodeInfo ≜ CHOOSE n ∈ newNodesInfo : TRUE
        otherNodes ≜ newNodesInfo \ {nodeInfo}
    IN
        IF otherNodes = {}
         THEN nodeInfo.childrenId
         ELSE nodeInfo.childrenId ∪ GetChildren(otherNodes)


AddEntriesToVV(vv, newNodesInfo, nodesToRemove, NullVVEntry) ≜
    LET
        newNodesIds ≜ DOMAIN newNodesInfo
        setNewNodesInfo ≜ {newNodesInfo[n] : n ∈ newNodesIds}
        newNodesParents ≜ {newNodesInfo[n].parent : n ∈ newNodesIds}
        newNodesChildren ≜ IF setNewNodesInfo = {} THEN {}
                           ELSE GetChildren(setNewNodesInfo) ∪ nodesToRemove

        tempVV ≜ [n ∈ DOMAIN vv ↦ IF vv[n] = NullVVEntry
                                  THEN vv[n]
                                  ELSE [vv[n] EXCEPT
                                        !.childrenId = @ \ newNodesChildren]]

        root ≜ CHOOSE n ∈ DOMAIN vv : {vv[n].parent} ∩ (DOMAIN vv) = {}

    IN
        [n ∈ DOMAIN vv ↦
           IF n ∈ newNodesIds
            THEN newNodesInfo[n]

            ELSE IF vv[n] = NullVVEntry ∨ vv[n].nodeId ∈ nodesToRemove
            THEN NullVVEntry

            ELSE IF n ∈ newNodesParents
            THEN [tempVV[n] EXCEPT
                    !.childrenId = @ ∪ {nId ∈ newNodesIds :
                                        newNodesInfo[nId].parent = n}]

            ELSE IF n ∈ newNodesChildren
            THEN [tempVV[n] EXCEPT
                    !.parent = CHOOSE nId ∈ newNodesIds :
                                 n ∈ newNodesInfo[nId].childrenId]

            ELSE tempVV[n]]

ChangeKeysChildInterested(datastore, keys, newChildInterested) ≜
```

```
    [k ∈ DOMAIN datastore  ↦ IF k ∈ keys
                             THEN [datastore[k] EXCEPT
                                     !.childInterested = newChildInterested]
                             ELSE datastore[k]]

GetNodesRelatedTo(nodesRelated, vv, NullVVEntry, nodesAbove) ≜
    GetNodesBelow(nodesRelated, vv, NullVVEntry) ∪ {n ∈ DOMAIN vv :
                                                      ∧ vv[n] ≠ NullVVEntry
                                                      ∧ n ∈ nodesAbove}

IsHierarchyUpdateOfNode(update, nIds) ≜
    ∨ ∧ ∨ update.msgType = "remove_node"
        ∨ update.msgType = "new_node"
        ∨ update.msgType = "add_node_to_hierarchy"
        ∨ update.msgType = "node_failed"
      ∧ update.nodeId ∈ nIds
    ∨ ∧ update.msgType = "new_parent"
      ∧ update.parent ∈ nIds
    ∨ ∧ update.msgType = "remove_child"
      ∧ update.sourceId ∈ nIds


RemoveHierarchyUpdatesOfNodeFromLog(s, nIds) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
         ELSE IF IsHierarchyUpdateOfNode(s[i], nIds)
         THEN Append(F[i − 1], [msgType ↦ "ignore_old_hierarchy_change"])
         ELSE Append(F[i − 1], s[i])
  IN
     F[Len(s)]

RemovePreviousHierarchyUpdatesOfNode(n) ≜
    [nId ∈ DOMAIN configuration ↦
              [configuration[nId] EXCEPT
                    !.log = RemoveHierarchyUpdatesOfNodeFromLog(@, n)]]


IsRemoveOfNode(msg, n) ≜
    msg.msgType = "remove_node" ∧ msg.nodeId = n

NodeWasRemoved(s, n) ≜
    ∃ i ∈ 1 .. Len(s) : IsRemoveOfNode(s[i], n)

IsRemoveAck(op) ≜ op.msgType = "ack_remove"
```

```
NoRemoveAck(s) ≜
    ∀ i ∈ 1 .. Len(s) : IsRemoveAck(s[i]) = FALSE

SendToParent(update, sourceVV, nId, NullVVEntry) ≜
    ∨ update.msgType = "new_key_or_update"
    ∨ update.msgType = "remove_node"
    ∨ ∧ update.msgType = "key_update"
      ∧ LET
            key ≜ configuration[nId].datastore[update.keyId]
        IN
            key.versionId = update.version
      ∧ ∨ sourceVV[update.version.nodeId] = NullVVEntry
        ∨ sourceVV[update.version.nodeId].executedOperations < update.version.versionNumber
    ∨ ∧ update.msgType = "add_node_to_hierarchy"
      ∧ configuration[nId].vv[update.nodeId] ≠ NullVVEntry
      ∧ sourceVV[update.nodeId] = NullVVEntry

IsRemoveChild(op) ≜
    ∧ op.msgType = "remove_child"

SelectSeqToParent(s, sourceVV, n, NullVVEntry, KeyId, NullDatastoreEntry) ≜
  LET
    node ≜ configuration[n]
    vv ≜ node.vv
    children ≜ vv[n].childrenId
    nodes ≜ DOMAIN configuration
      F[i ∈ 0 .. Len(s)] ≜
        IF i = 0
        THEN ⟨⟩
        ELSE IF s[i].msgType = "remove_child"
        THEN Append(F[i − 1],
                    [s[i] EXCEPT !.newChildren =
                                 [nId ∈ nodes ↦
                                     IF nId ∈ children
                                     THEN GetKeysChildIsInterested(n,
                                                                   nId,
                                                                   KeyId,
                                                                   NullDatastoreEntry)
                                     ELSE {}],
                                 !.sourceVV = vv])
        ELSE IF SendToParent(s[i], sourceVV, n, NullVVEntry)
        THEN Append(F[i − 1], s[i])
        ELSE F[i − 1]
  IN  F[Len(s)]
```

AUXILIAR FUNCTIONS TO PACKAGES

```
RECURSIVE GetPackage(_)
GetPackage(msgss) ≜
    IF Head(msgss).msgType = "package_end" ∨ Tail(msgss) = ⟨⟩
     THEN ⟨Head(msgss)⟩
     ELSE IF Head(msgss).msgType = "package_start"
     THEN ⟨⟩
     ELSE ⟨Head(msgss)⟩ ∘ GetPackage(Tail(msgss))

GetLastMessage(msgss) ≜
    Head(SubSeq(msgss, Len(msgss), Len(msgss)))

RemoveLastMessage(msgss) ≜
    SubSeq(msgss, 1, Len(msgss) − 1)

RECURSIVE PositionOfPackageStart(_, _)
PositionOfPackageStart(msgss, pos) ≜
    IF GetLastMessage(msgss).msgType = "package_start"
     THEN pos
     ELSE PositionOfPackageStart(RemoveLastMessage(msgss), pos − 1)

GetSizeOfMsgsPack(msgss) ≜
    Len(msgss) − PositionOfPackageStart(msgss, Len(msgss))

IsPackageStart(operation) ≜
    operation.msgType = "package_start"

IsPackageEnd(operation) ≜
    operation.msgType = "package_end"
```

AUXILIAR FUNCTIONS TO 'REMOVE *NODE*'

```
RECURSIVE JoinSets(_)
JoinSets(bigSet) ≜
    LET
        set ≜ CHOOSE s ∈ bigSet : TRUE
    IN
        IF Cardinality(bigSet) = 1
         THEN set
         ELSE set ∪ JoinSets(bigSet \ {set})
```

```
RemoveNodeFromVV(vv, removeNodeInformation, NullVVEntry) ≜
    IF removeNodeInformation = NullVVEntry
     THEN vv
     ELSE
         LET
             nodeId ≜ removeNodeInformation.nodeId
             nodeParent ≜ removeNodeInformation.parent
             nodeChilren ≜ removeNodeInformation.childrenId

             newChildrenOfParent ≜
                     IF vv[nodeParent] ≠ NullVVEntry
                      THEN (vv[nodeParent].childrenId \ {nodeId}) ∪ nodeChilren
                      ELSE {}
         IN
             IF vv[nodeId] ≠ NullVVEntry
              THEN
                 [nId ∈ DOMAIN configuration ↦
                           IF nId = nodeId
                            THEN NullVVEntry

                            ELSE IF nId ∈ nodeChilren ∧ vv[nId] ≠ NullVVEntry
                            THEN [vv[nId] EXCEPT !.parent = nodeParent]

                            ELSE IF nId = nodeParent ∧ vv[nId] ≠ NullVVEntry
                            THEN [vv[nId] EXCEPT !.childrenId = newChildrenOfParent]

                            ELSE vv[nId]]
              ELSE vv

RECURSIVE RemoveNodesFromVV(_, _, _)
RemoveNodesFromVV(vv, setNodesToRemove, NullVVEntry) ≜
    LET
        nodeInfo ≜ CHOOSE nodeInfo ∈ setNodesToRemove : TRUE

        updated_vv ≜ IF vv[nodeInfo] ≠ NullVVEntry
                      THEN RemoveNodeFromVV(vv, vv[nodeInfo], NullVVEntry)
                      ELSE vv

        otherNodes ≜ setNodesToRemove \ {nodeInfo}
    IN
        IF otherNodes = {}
         THEN updated_vv
         ELSE RemoveNodesFromVV(updated_vv, otherNodes, NullVVEntry)

IsChildOfNode(op, n, oldChild, oldChildKeys, NullVVEntry) ≜
    ∧ op.msgType = "new_node"
```

```
    ∧ configuration[op.nodeId].status.connectionStatus = "online"
    ∧ configuration[n].vv[op.nodeId] = NullVVEntry
    ∧ op.keyIds ⊆ oldChildKeys
    ∧ op.nodeId ∈ configuration[oldChild].vv[oldChild].childrenId

SelectSeqOfChildrenNotKnown(n, s, oldChild, oldChildKeys, NullVVEntry) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
        ELSE IF IsChildOfNode(s[i], n, oldChild, oldChildKeys, NullVVEntry)
        THEN Append(F[i − 1], s[i])
        ELSE F[i − 1]
  IN  F[Len(s)]


InheritChildren(n, vv, nodeToRemove, scId, sourceVV,
                NullVVEntry, NullNodeId, NullDatastoreEntry) ≜
  LET
      temp_vv ≜ UpdateVV(n, vv, nodeToRemove, sourceVV, vv[n],
                         NullVVEntry, NullNodeId)

      actualChildren ≜ configuration[nodeToRemove].vv[nodeToRemove].childrenId

      tempChildrenInherit ≜ vv[scId].childrenId

      nodes ≜ DOMAIN configuration

      childrenToAdd ≜ actualChildren \ {nId ∈ nodes : vv[nId] ≠ NullVVEntry}

      OpOnline ≜ configuration[scId].status.numOpOnline

      entriesToAdd ≜
          [nId ∈ childrenToAdd ↦
              [configuration[scId].vv[nId] EXCEPT
                  !.executedOperations = Min(@, OpOnline)]]

      nodesBelowChildren ≜
          GetNodesBelow({nodeToRemove}, vv, NullVVEntry) \ {nodeToRemove}

      nodesRemove ≜
          {nId ∈ nodesBelowChildren :
              configuration[nodeToRemove].vv[nId] = NullVVEntry}

      temp2_vv ≜ IF childrenToAdd = {}
                 THEN temp_vv
                 ELSE AddEntriesToVV(temp_vv,
                                     entriesToAdd,
                                     {},
                                     NullVVEntry)
```

```
        end_vv ≜ RemoveNodesFromVV(temp2_vv,
                                   nodesRemove ∪ {nodeToRemove},
                                   NullVVEntry)

        keys ≜ DOMAIN configuration[n].datastore

        newChildrenKeys ≜
            [nId ∈ actualChildren ↦
                GetKeysChildIsInterested(nodeToRemove, nId, keys, NullDatastoreEntry)
                    ∪ {k ∈ keys :
                            configuration[nId].datastore[k] ≠ NullDatastoreEntry}]

        keysOfNewChildren ≜
            IF actualChildren = {}
             THEN {}
             ELSE JoinSets({newChildrenKeys[nId] : nId ∈ actualChildren})
    IN
        [vv ↦ end_vv,
         newChildrenKeys ↦ newChildrenKeys,
         keysOfNewChildren ↦ keysOfNewChildren,
         nodesAdded ↦ childrenToAdd,
         nodesRemoved ↦ nodesRemove]
```

The input 'newChildren' has the *Ids* of the keys in which every children is interested in. This function returns which of those new children is interested in the key 'k', if any

```
NewChildInterestedInKey(k, newChildren, NullNodeId) ≜
    LET
        interested ≜ {c ∈ DOMAIN newChildren : k ∈ newChildren[c]}
    IN
        IF interested = {}
         THEN NullNodeId
         ELSE CHOOSE x ∈ interested : TRUE


IsNewKey(operation) ≜
    operation.msgType = "new_key"
```

This function is executed when a message of a "new_key" must be re-propagated due to the removal of a node from the hierarchy. The message is re-propagating to a new node, if that node is interested on the key. In that case, the current node must update its own key information, updating the child interested in that key

```
RECURSIVE UpdateDSWithNewKey(_, _, _, _, _, _)
UpdateDSWithNewKey(datastore, newKeyMsgsMissing, newChildren, srcVV,
                   NullVVEntry, NullNodeId) ≜
    IF newKeyMsgsMissing = ⟨⟩
     THEN datastore
```

```
    ELSE
      LET
          op ≜ Head(newKeyMsgsMissing)
          k ≜ op.keyId
          nodesInt ≜ op.nodesInterested

          nodeInterested ≜
              IF datastore[k].childInterested = NullNodeId
               THEN GetChildInterestedInKey(nodesInt,
                                            newChildren,
                                            srcVV,
                                            NullNodeId,
                                            NullVVEntry)
               ELSE NullNodeId

          updated_datastore ≜
              IF nodeInterested = NullNodeId
               THEN datastore
               ELSE [datastore EXCEPT
                          ![k] = [@ EXCEPT
                                    !.childInterested = nodeInterested]]

          tail ≜ Tail(newKeyMsgsMissing)
      IN
          UpdateDSWithNewKey(updated_datastore, tail, newChildren, srcVV,
                             NullVVEntry, NullNodeId)


IsToRepeatNewNode(op, vv, seq, NullVVEntry) ≜
    ∧ op.msgType = "new_node"
    ∧ vv[op.nodeId] = NullVVEntry
    ∧ configuration[op.nodeId].status.connectionStatus = "pending"

SelectSeqOfNewNodes(s, vv, NullVVEntry) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0
         THEN ⟨⟩
         ELSE IF IsToRepeatNewNode(s[i], vv, SubSeq(s, i + 1, Len(s)), NullVVEntry)
         THEN Append(F[i − 1], s[i])
         ELSE F[i − 1]

  IN  F[Len(s)]
```

'logOperations' are the messages that must be re-propagated and will decrease on every execution of the function. Every execution of the function, one message is selected, and it is decided if the message should be re-propagated to any of the new children. Every children will only receive the messages of interest.

```
RECURSIVE SelectSeqToChildren(_, _, _, _, _, _, _, _, _)
SelectSeqToChildren(logOperations, datastore, vv, sourceVV, msgsToChildren,
                    NullVVEntry, msgNewParent, nodesAbove) ≜
  IF logOperations = ⟨⟩
   THEN [nId ∈ DOMAIN msgsToChildren ↦ msgsToChildren[nId] ∘ msgNewParent]
   ELSE
     LET op ≜ Head(logOperations)

         children ≜ DOMAIN msgsToChildren

         new_msgsToChildren ≜
           IF ∧ op.msgType = "key_update"
              ∧ LET
                    key ≜ configuration[op.sourceId].datastore[op.keyId]
                IN
                    key.versionId = op.version
              ∧ datastore[op.keyId].childInterested ∈ children
            THEN [msgsToChildren EXCEPT
                    ![datastore[op.keyId].childInterested] =
                       Append(msgsToChildren[datastore[op.keyId].childInterested],
                              op)]

            ELSE IF ∧ op.msgType = "new_key"
                    ∧ datastore[op.keyId].childInterested ∈ children
            THEN [msgsToChildren EXCEPT
                    ![datastore[op.keyId].childInterested] =
                       Append(msgsToChildren[datastore[op.keyId].childInterested],
                              op)]

            ELSE IF ∧ op.msgType = "add_node_to_hierarchy"
                    ∧ vv[op.nodeId] ≠ NullVVEntry
                    ∧ op.nodeId ∈ nodesAbove
            THEN [msgC ∈ children ↦ Append(msgsToChildren[msgC], op)]

            ELSE IF op.msgType = "remove_node" ∧ sourceVV[op.nodeId] ≠ NullVVEntry
            THEN [msgC ∈ children ↦ Append(msgsToChildren[msgC], op)]

            ELSE msgsToChildren
     IN
        SelectSeqToChildren(Tail(logOperations),
                            datastore,
                            vv,
                            sourceVV,
                            new_msgsToChildren,
                            NullVVEntry,
                            msgNewParent,
                            nodesAbove)
```

```
RemoveRemoveChildMessage(s) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
         ELSE IF s[i].msgType = "remove_child"
         THEN Append(F[i − 1], [msgType ↦ "offline"])
         ELSE Append(F[i − 1], s[i])
  IN
     F[Len(s)]

IsCorrectHierarchy(msg) ≜
    msg.msgType = "correct_hierarchy"

IsNewParent(msg) ≜
    msg.msgType = "new_parent"

IsNodeFailure(msg) ≜
    msg.msgType = "node_failed"
```

AUXILIAR FUNCTIONS TO 'FAIL *NODE'*

```
RECURSIVE GetNodesFailed(_, _, _, _, _)
GetNodesFailed(vv, nodeFailed, handled, nodeId, NullNodeId) ≜
    LET
        n ≜ CHOOSE n ∈ nodeFailed \ handled : TRUE

        nextConnection ≜
            IF n ∈ GetNodesAboveInHierarchy(vv, vv[nodeId].parent, NullNodeId)
             THEN {vv[n].parent}
             ELSE IF vv[n] = [nodeId ↦ NullNodeId]
             THEN {}
             ELSE vv[n].childrenId

        consOffline ≜
            {nId ∈ nextConnection :
                    configuration[nId].status.connectionStatus = "offline"}

        new_handled ≜ handled ∪ {n}
        new_nodeFailed ≜ nodeFailed ∪ consOffline
    IN
        IF new_nodeFailed = new_handled
         THEN new_nodeFailed
         ELSE GetNodesFailed(vv, new_nodeFailed, new_handled, nodeId, NullNodeId)

IsNewNodeOnline(op, n, childFail, newConnections,
                    NullNodeId, NullVVEntry, NullDatastoreEntry) ≜
```

```
    LET
        nodesAbove ≜
            GetNodesAboveInHierarchy(configuration[op.nodeId].vv,
                                     configuration[op.nodeId].vv[op.nodeId].parent,
                                     NullNodeId)
    IN
        ∧ op.msgType = "new_node"
        ∧ configuration[op.nodeId].status.connectionStatus = "online"
        ∧ configuration[n].vv[op.nodeId] = NullVVEntry
        ∧ op.keyIds ⊆ GetNodeKeys(childFail, NullDatastoreEntry)
        ∧ ∨ newConnections = {}
          ∨ ∧ newConnections ≠ {}
            ∧ ∀ nId ∈ newConnections :
                  ∨ configuration[op.nodeId].vv[nId] = NullVVEntry
                  ∨ nId ∉ nodesAbove

SelectSeqOfNodesAddedButNotKnown(s, n, childFail, newConnections,
                                 NullNodeId, NullVVEntry, NullDatastoreEntry) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0
        THEN ⟨⟩
        ELSE IF IsNewNodeOnline(s[i], n, childFail, newConnections,
                                NullNodeId, NullVVEntry, NullDatastoreEntry)
             THEN Append(F[i − 1], s[i])
             ELSE F[i − 1]
  IN F[Len(s)]

GetNewChildren(newUnknownNodes, knownNodes, nodeId, NullNodeId, NullVVEntry) ≜
    LET
        all ≜ newUnknownNodes ∪ knownNodes

        newNodes ≜
            {n ∈ newUnknownNodes :
             ∀ nId ∈ all \ {n} :
                 ∧ ∨ configuration[n].vv[nId] = NullVVEntry
                   ∨ nId ∉ GetNodesAboveInHierarchy(configuration[n].vv,
                                                    configuration[n].vv[n].parent,
                                                    NullNodeId)
                 ∧ ∨ configuration[nId].vv[n] = NullVVEntry
                   ∨ n ∈ GetNodesAboveInHierarchy(configuration[nId].vv,
                                                  configuration[nId].vv[nId].parent,
                                                  NullNodeId)}
    IN
        [n ∈ newNodes ↦
            LET
```

```
        node ≜ configuration[n]
        vv ≜ configuration[n].vv
    IN
        [vv[n] EXCEPT
            !.executedOperations = node.status.numOpOnline,
            !.childrenId =
                {nId ∈ knownNodes :
                    ∧ vv[nId] ≠ NullVVEntry
                    ∧ nId ∉ GetNodesAboveInHierarchy(vv,
                                                     vv[n].parent,
                                                     NullNodeId)},
            !.parent = nodeId]]

IsAddsHierarchyUpdateOfNode(update, nIds) ≜
    ∨ ∧ ∨ update.msgType = "new_node"
        ∨ update.msgType = "add_node_to_hierarchy"
        ∨ update.msgType = "node_failed"
      ∧ update.nodeId ∈ nIds
    ∨ ∧ update.msgType = "new_parent"
      ∧ update.parent ∈ nIds
    ∨ ∧ update.msgType = "correct_hierarchy"
      ∧ update.sourceId ∈ nIds

RemoveAddsHierarchyUpdatesOfNodeFromMsgs(s, nIds) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
        ELSE IF IsAddsHierarchyUpdateOfNode(s[i], nIds)
        THEN F[i − 1]
        ELSE Append(F[i − 1], s[i])
  IN
    F[Len(s)]

RemoveHierarchyUpdatesOfNodeFromMsgs(s, nIds) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
        ELSE IF IsHierarchyUpdateOfNode(s[i], nIds)
        THEN F[i − 1]
        ELSE Append(F[i − 1], s[i])
  IN
    F[Len(s)]

RemoveInvMsgsUpdatesOfNode(n) ≜
    [nId ∈ DOMAIN msgs ↦
            RemoveHierarchyUpdatesOfNodeFromMsgs(msgs[nId], n)]
```

```
IsRemovingNode(op, n) ≜
    ∨ ∧ op.msgType = "new_parent"
      ∧ op.nodeToRemove = n

MsgsRemovingNode(s, n) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
                 ELSE IF IsRemovingNode(s[i], n) THEN Append(F[i − 1], s[i])
                                                 ELSE F[i − 1]
  IN  F[Len(s)]


SelectSeqOfAddsFailed(s, n, keyIds, NullNodeId, NullVVEntry) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
         ELSE IF ∧ s[i].msgType = "new_node"
                 ∧ configuration[s[i].nodeId].status.connectionStatus ≠ "online"
         THEN Append(F[i − 1], s[i])
         ELSE F[i − 1]
  IN  F[Len(s)]

RemoveAddsOfOfflines(s, nodesFailed) ≜
  LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
         ELSE IF ∧ s[i].msgType = "add_node_to_hierarchy"
                 ∧ s[i].nodeId ∈ nodesFailed
                 ∧ configuration[s[i].nodeId].status.connectionStatus ≠ "online"
         THEN F[i − 1]
         ELSE Append(F[i − 1], s[i])
  IN  F[Len(s)]

RECURSIVE GetSourcesOfMessages(_)
GetSourcesOfMessages(sequence) ≜
    IF Len(sequence) = 0
     THEN {}
     ELSE LET
             head ≜ Head(sequence)
             sourceId ≜ IF head.msgType = "new_parent"
                        THEN {head.parent}
                        ELSE IF ∨ head.msgType = "package_start"
                                ∨ head.msgType = "package_end"
                                ∨ head.msgType = "offline_operations"
                                ∨ head.msgType = "datastore_change"
                        THEN {}
                        ELSE {head.sourceId}
```

```
        IN
            sourceId ∪ GetSourcesOfMessages(Tail(sequence))

RECURSIVE GetChildRelated(_, _, _)
GetChildRelated(vv, nodeToFind, children) ≜
    IF vv[nodeToFind].parent ∈ children
     THEN vv[nodeToFind].parent
     ELSE GetChildRelated(vv, vv[nodeToFind].parent, children)


IsRemoveNodeOf(op, nodesFailed) ≜
    ∧ op.msgType = "remove_node"
    ∧ op.nodeId ∈ nodesFailed

RemoveOpRemoveNodeOfFailures(s, nodesFailed) ≜
    LET F[i ∈ 0 .. Len(s)] ≜
        IF i = 0 THEN ⟨⟩
                 ELSE IF IsRemoveNodeOf(s[i], nodesFailed) THEN F[i − 1]
                                                           ELSE Append(F[i − 1], s[i])
    IN  F[Len(s)]

MsgAffectingReProp(op) ≜
    ∨ op.msgType = "new_parent"
    ∨ op.msgType = "remove_child"
    ∨ op.msgType = "node_failed"
    ∨ op.msgType = "correct_hierarchy"
    ∨ op.msgType = "ack_hierachy_corrected"

NoIdConflicts ≜
    ∧ failedNodes = {}
    ∧ LET
        nodes ≜ DOMAIN configuration
        onlineNodes ≜
            {n ∈ nodes :
                configuration[n].status.connectionStatus = "online"}
      IN
        {n ∈ onlineNodes :
         ∃ i ∈ 1 .. Len(msgs[n]) :
                MsgAffectingReProp(msgs[n][i])} = {}
```

AUXILIAR FUNCTIONS TO 'PROCESS *MESSAGE*'

```
ContainsMsgNewParent(s) ≜
    ∃ i ∈ 1 .. Len(s) : s[i].msgType = "new_parent"
```

$ContainsMsgAckParent(s) \triangleq$
$\exists\, i \in 1\,..\,Len(s) : s[i].msgType =$ "ack_parent"

$ContainsMsgCorrectingHier(s) \triangleq$
$\exists\, i \in 1\,..\,Len(s) : \lor\, s[i].msgType =$ "correct_hierarchy"
$\lor\, s[i].msgType =$ "ack_hierachy_corrected"
$\lor\, s[i].msgType =$ "node_failed"

AUXILIAR FUNCTIONS TO INVARIANTS

$IsKeyUpdateWithVersionAfter(op,\ version) \triangleq$
$\land\, op.msgType =$ "key_update"
$\land\, op.version.nodeId = version.nodeId$
$\land\, op.version.versionNumber > version.versionNumber$

Verifies, for every update, if any update executed before, had a version created after this one
$ContainsKeyUpdateBreakingCausality(op,\ seq) \triangleq$
$\land\, op.msgType =$ "key_update"
$\land \exists\, i \in 1\,..\,Len(seq) : IsKeyUpdateWithVersionAfter(seq[i],\ op.version)$

$ContainsMsgToUpdateKey(s,\ k) \triangleq$
$\exists\, i \in 1\,..\,Len(s) : \land\, s[i].msgType =$ "key_update"
$\land\, s[i].keyId = k$

$ContainsMsgAffectingConvergence(s) \triangleq$
$\exists\, i \in 1\,..\,Len(s) : \lor\, s[i].msgType =$ "new_parent"
$\lor\, s[i].msgType =$ "ack_remove"
$\lor\, s[i].msgType =$ "node_failed"
$\lor\, s[i].msgType =$ "correct_hierarchy"

---

* Modification History
* Last modified *Wed Sep* 23 23:55:21 *BST* 2015 by *Miguel*
* Created *Tue* May 12 17:52:02 *BST* 2015 by *Miguel*