MASARYK UNIVERSITY
FACULTY OF INFORMATICS

# Formal design of a distributed hash table

MASTER'S THESIS

**Jakub Senko**

Brno 2016

## Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Jakub Senko

**Advisor:** RNDr. Vojtěch Řehák, Ph.D.

## Acknowledgement

I would like to thank... TODO

## Abstract

This thesis is focused on a design of a distributed hash table model and formal verification of its behaviour under failure conditions [using PlusCal/TLA+ languages]. The model matches algorithms and protocols used in the Infinispan project. Advantages and limitations of the model and the used technique are described.

## Keywords

formal software verification, model checking, distributed hash table, Infinispan

# Contents

# 1 Introduction

Testing is an integral part of software development - Several types of testing - Model checking - Having a proper model can be beneficial during the design and analysis phases, because it helps developers to catch potential design errors, especially in complex concurrent and distributed systems. - There are several projects at Red Hat where this is important. - One of these is Infinispan, a its core a distributed key-value storage. Having a model of a hash-table with the same basic design would help developers to prevent design errors. - This thesis was motivated by a successful use of model checking tools for development of Amazon AWS S3 product and is aimed to provide benefits of these techniques to Infinispan developers.

# 2 Model Checking

In this chapter, [motivation, considerations] overview of the model checking techniques is presented.

## 2.1 System Verification

Techniques:

## 2.2 Characteristics of Model Checking

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

TODO:

- system specification
- Comparison to other testing & verification techniques
- Disadvantages (state explosion, model realism - Any verification using model-based techniques is only as good as the model of the system.)

## 2.3 Transition systems

Transition systems are often used in computer science as models to describe the behavior of systems. They are basically directed graphs where nodes represent states, and edges model transitions, i.e., state changes. We will use the following formal definition of the TS [cit]:

A transition system (TS) is a tuple $(S, Act, \rightarrow, I, AP, L)$ where

- $S$ is a set of states,
- $Act$ is a set of actions,
- $\rightarrow \in S \times Act \times S$ is a transition relation,
- $I \in S$ is a set of initial states,
- $AP$ is a set of atomic propositions,
- and $L : S \rightarrow 2^{AP}$ is a labeling function.

TS is called finite if $S$, $Act$, and $AP$ are finite.

As an example, consider an hour-clock. It may be represented by 12 different states, with transitions representing actions the clock takes. This includes incrementing the counter, or overflowing from „12" to „1" state. In the conventional (and isolated) computer system, the states correspond to every possible contents of the memory and internal state of the processor.

The transition system may resemble other formalisms from computational theory, such as a non-deterministic finite state machine. However, there are important distinctions [cit, wiki]:

- The set of states is not necessarily finite, or even countable,
- the set of transitions is not necessarily finite, or even countable,
- there are no initial or final states defined.

TODO

The notion of the labeling function is very important, since it allows for expressing formal assertions about states. For instance $[hour = 12]$ is a labeling for a valid clock state, while $[hour > 12]$ is not. The function $L$ relates a set $L(s) \in 2^{AP}$ of atomic propositions to any state $s$. It intuitively stands for exactly those atomic propositions $a \in AP$ which are satisfied by state $s$. Given that $\Phi$ is a propositional logic formula, then s satisfies the formula $\Phi$ if the evaluation induced by $L(s)$ makes the formula $\Phi$ true; that is: $s \models \Phi$ iff $L(s) \models \Phi$. This enables reasoning about the entire TS using *temporal logic formulas*.

## 2.4 Executions and Behaviors

We represent the execution (or behavior[1]) of a system as a sequence of states (and actions). We specify a system by providing a set of executions representing a correct behavior of the system [cit]. For example[2],

$$\ldots \rightarrow^{tick} [hour = 1] \rightarrow^{tick} [hour = 2] \rightarrow^{tick} \ldots \rightarrow^{tick} [hour = 12] \rightarrow^{tick}$$
$$[hour = 1] \rightarrow^{tick} \ldots$$

1. This terminology is used in the TLA+ specification, and is interchangeble with „execution".
2. For convenience, an element of the transition relation may be expressed as an arrow between two states, labeled with the action from $Act$.

More formally, execution is an *initial* and *maximal* execution *fragment*. Execution fragment is an alternating sequence of states and actions ending with a state. Maximal means that the fragment can not be prolonged, that is, it is either infinite, or ends with a state from which no further transition is possible. Initial fragment is defined as starting in $s \in I$.

## 2.5 TODO

Program Graph, Non-determinism, Concurrency

## 2.6 Linear Temporal Logic

Linear Temporal Logic (LTL) is an extension of propositional logic with two temporal modal operators. The notion of time within the context of LTL is not related to a „continuous real time line", but rather a discreet sequence of steps ordered by „precedence". This makes it ideal to reason about properties of TS executions, and indeed, it was first proposed specifically for the formal verification of computer programs [cit, wiki]. Given an execution $\rho$, LTL formula $F$ assigns a boolean value to $\rho$. In other words, $\rho$ satisfies $F$, expressed as $F \models \rho$, iff $F$ is evaluated as $true$ for $\rho$.

Syntax of LTL can be inductively defined by extending propositional logic (PL) syntax. If $\Phi$, $\Psi$ are well-formed PL formulas, then:[3]

- $\Phi$, $\Psi$ are LTL formulas,
- $(\mathbf{X}\Phi)$ is an LTL formula (pronounced „next $\Phi$"),
- $(\Phi\mathbf{U}\Psi)$ is an LTL formula („$\Phi$ until $\Psi$").

The semantics can be informally descibed by providing executions that satisfy given formulas. We define $any$ to be a placeholder for an arbitrary propositional formula:

$$([\Phi]_0 \to [any]_1 \to [any]_2 \to [any]_3 \to \ldots) \models \Phi$$

$$([any]_0 \to [\Phi]_1 \to [any]_2 \to [any]_3 \to \ldots) \models \mathbf{X}\Phi$$

$$([\Phi \wedge \neg\Psi]_0 \to \ldots \to [\Phi \wedge \neg\Psi]_i \to [\Psi]_{i+1} \to [any]_{i+2} \to \ldots) \models \Phi\mathbf{U}\Psi$$

3. Temporal operators have precedence and parentheses can be ommited if the result is not ambiguous.

Moreover, we can use $\mathbf{U}$, to derive two additional important operators:

- „Eventually",$\Diamond\Phi$ which allows for expressing an assertion that must be true at some point in the future, defined as $true\mathbf{U}\Phi$,
- „always", $\Box\Phi$ for formulas that must be satisfied during the entire execution, equivalent to $\neg\Diamond\neg\Phi$.

TODO bridge to TLA+

## 2.7 Liveness and Fairness

- regular properties
- omega properties

# 3 Distributed Hash Table

Hash table is a data structure that maps keys to values (associative array), using a hashing function applied on keys. This is ordinarily used for a small sized in-memory data storage embedded inside a program. However, some use cases require storing of a large amount of unstructured data, which can use persistent key-value databases that store the data on hard drive storage. However, this reduces speed -> is not suitable for caching. To avoid the memory size limitation, data can be distributed across many nodes that together provide a large pool of fast (but volatile) memory.

The main issues arise from managing the distrubution of data on the nodes while maintaining high degree of consistency and availability.

- Formal definition, hash function

## 3.1 Associative Array

Associative array (also dictionary or map) is an abstract data structure that can be represented as a relation between a set of keys and a set of values: $R \subseteq K \times V$, such that every key is mapped (in relation) to at most one value.

There are three basic operations defined for an associative array: *insert*, *search* and *delete*. [intro to alg.]. Insert operation updates a dictionary instance to contain a key-value pair.[1] Search retrieves a value associated with a given key if it exists and delete removes the key-value pair from the dictionary.

The are many possible implementations of an associative array, but the most common are hash map and search tree. Typically, hash map is used due to the constant amortized cost for each operation, but search tree has better worst case performance (logarithmic vs. linear) and provides additional features, such as ordered iteration. [cit]

To illustrate the concept of dictionary, we will briefly describe a simple Direct-address table. [cit] Given $T$ is a contiguous array of size $|K|$, let each position correspond to $k \in K$. It is assumed that if $K$ is not of a form $K' = \{0, 1, ..., |K|\}$ there exists a bijective mapping function $m : K \to K'$. If the table contains key $k$, position $m(k)$ holds a reference to the chosen value $v \in V$, otherwise there is a special $nil$ value.

This implementation is practical only if $K$ is reasonably small, or most of the keys are in use. The dictionary operations would then reduce to those

1. We are letting the failure handling to be implementation-specific.

of the contiguous array, e.g $Insert(T, k, v) \triangleq T[k] := v$

## 3.2 Hash Table

The problems of the Direct-address table can be resolved by reducing the contiguous array to a reasonable size. This would result in some $k$ not having an assigned slot, however we can replace $m$ with a surjective *hash* function $h$ that maps the set of keys to a smaller set $K'$ of indices into $T$. This would require that more than one value may be stored in a single position (a *collision*), but the slot can be used to reference a linked list that will contain the overflowing key-value pairs. This would increase the *worst-case* complexity of *search* operation to $(n)$, but the *amortized* cost remains $(1)$, if the hash function is properly chosen. [cit]

„A good hash function satisfies (approximately) the assumption of simple uniform hashing: each key is equally likely to hash to any of the [...] slots, independently of where any other key has hashed to." [cit, intro to alg]

One of the commonly used hash functions is *division remainder*

$h(k) = kmodz$

where $z$ should be a prime number not close to some power of 2. [TODO]

Although this method reduces the wasting of space when compared to the Direct-address table, in some cases the size of $V$ is too large for a physical memory of a single computer. However, because of the advantagions properties of hash functions, it is feasible to split the table into several chunks and distribute it across nodes in a distributed system. [reword]

## 3.3 Parallelism and Concurrency

In the previous section, we have descibed a use case where it is advantageous to use multiple physical machines (processors) to deal with resource-intensive tasks. Parallelization has become the only major way to keep increasing the processing power of modern computers sufficiently to meet the demand, apart from increasing transistor density.[2] This includes scaling-up, by adding multiple processing units into a single CPU, and scaling out by using multiple physical computers connected by a network. Creating algorithms that can safely work in a parallell environment has therefore become one of the most important tasks of software design.

2. Not considering alternative models of computations, in particular the quantum computing, which is not yet practical to be in common use.

[TODO]

## 3.4 Distributed Algorithm

TODO common problems and algorithms

- Coordinator selection
- Distributed locking

## 3.5 CAP theorem

it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- Consistency (all nodes see the same data at the same time)
- Availability (every request receives a response about whether it succeeded or failed)
- Partition tolerance (the system continues to operate despite arbitrary partitioning due to network failures)

## 3.6 Algorithms

TODO

## 3.7 Consistent Hashing

In the section describing a hash table, an idea of partitioning and distributing the data onto multiple separate nodes was proposed. To ensure effective read/write operations, all nodes must be aware of where value corresponding to each key should be stored, otherwise they would have to broadcast read requests or cause fragmentation when choosing where to write at random. Moreover, this mapping cannot be stored individually for each key or the space needed to store the keys would be linearly dependent on the number of values and defeating the point of distributed storage. Therefore, given that the image $K'$ of the hashing function $h$ is a set of integers in the form $\{0, 1, ..., |K|\}$ , it is necessary to store that assignment as a set of ranges of keys $R$, where each element is in the form $r_{<i,j>} = \{i, (i+1), (i+2), ..., j\}$ and all keys are covered: $\forall k \in K, \exists r \in R : k \in r$.

The mapping $p : N \rightarrow R$ from the set of nodes $N$ to the key-ranges (and reverse) may be created in numerous ways. There are two main properties that affect a choice of $p$:

- *Load factors* that determines a percentage of keys that are assigned for each individual node, usually to reflect the amount of computing resources available to store the values on a node, and
- *replication parameter*, $1 \geq rp \geq |N|$, that specifies the number of nodes that each key is assigned to.

Some of the implementations of DHTs allow for setting more complex load factors setting, but for the our purposes, we will use a weak requirement to balance data equally among the nodes. Replication parameter reflects a tradeoff between the extra storage space and higher write speed available with lower values and resiliency and improved read speed with higher values. Additional requirement for a good algorithm for computing $p$ in the presence of unreliable environment is that in the event of node failure, it minimizes the number of keys, and with them the amount of data, that must be relocated to other nodes, called *rebalancing*. Techniques that respect the described requirements are called *consistent hashing* and one of the most used, elegant, and simple concepts used to implement it is *hash wheel*.

## 3.8 Leader Election

[TODO]

# 4 Infinispan

„Infinispan is a distributed in-memory key/value data store [...]. It can be used both as an embedded Java library and as a language-independent service accessed remotely over a variety of protocols [...]. It offers advanced functionality such as transactions, events, querying and distributed processing." [cit, inf. web page] Provided to customers by Red Hat as JBoss Data Grid, it is a solution „to store information for very fast, low-latency response time and very high throughput". [jdg page] It can be used as a „distributed cache, NoSQL database, and event broker" [jdg page] to improve data storage and processing capabilities of massive-scale applications. Most of these featues are built on top of a modern distributed hash table implementation, therefore, for the purpose of this work, it is a great model example to explore. As a result, this chapter focuses on analyzing the core parts of this project.

## 4.1 Infinispan as a Cache

Cache is a data structure, which is used to temporarily store information that would be more expensive to compute or retrieve separately and repeatedly for each request. The most basic cache implementations provide the same interface as an associative array. There are three main caching techniques that can be used: [cit computerweekly.com]

- *Write-through* cache directs expensive write operation onto cache and through to underlying permanent storage before confirming completion to the host,
- *write-around* cache is similar, but data is written directly to permanent storage, and are cached upon subsequent reading, and
- *write-back* cache where the information being written is directed to cache and completion is immediately confirmed to the host and synchronized to the backing storage later.

Infinispan supports all of these techniques. [cit inf. docs]

## 4.2 Clustering in Infinispan

Infinispan offers four modes of operation, which determine how and where the data is stored:

- *Local,* where entries are stored on the local node only, regardless of whether a cluster has formed,

- *invalidation,* a write-around mode, where all entries are stored in a permanent store and cache is used to alleviate intensive read operations. Clustering is used only to invalidate data on all nodes on change.

- *Replication,* where all entries are replicated to all nodes. In this mode, Infinispan doesn't offer an increased space but a faster read times.

- *Distribution,* in which case entries are distributed to a subset of the nodes only. This option provides increased heap space by distributing the data on multiple, but not all nodes. Uses topology aware consistent hashing to offer improved durability in case of node crashes or network outages.

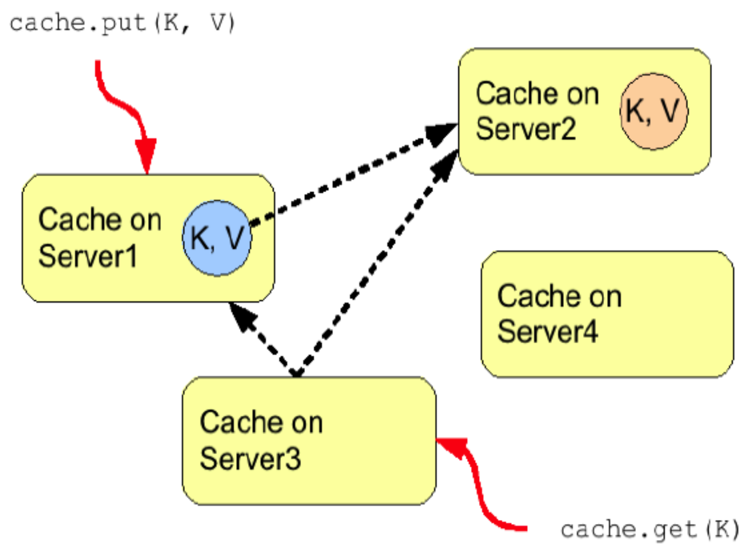
Last three options can use synchronous or asynchronous modes. In synchronous mode (write-through), client waits until the operations are confirmed complete, asynchronous (write-back) offers the fastest non-blocking access.

## 4.3 Distributed mode

Every entry is stored on a subset of the nodes in the grid, providing increased storage capacity for increased latency.

Compared to replication, distribution offers increased storage capacity, but with increased latency to access data from non-owner nodes, and durability (data may be lost if all the owners are stopped in a short time interval). Adjusting the number of owners allows you to obtain the trade off between space, durability, and latency.

Infinispan also offers a topology aware consistent hash which will ensure that the owners of entries are located in different data centers, racks, or physical machines, to offer improved durability in case of node crashes or network outages. The cache should be configured to work in distributed mode (either synchronous or asynchronous), and can otherwise be configured as normal.



CAP theorem - Infinispan has traditionally been biased towards Consistency and Availability, sacrificing Partition-tolerance.

A service that is consistent operates fully or not at all. Gilbert and Lynch use the word "atomic" instead of consistent in their proof, which makes more sense technically.

## 4.4 Architecture

Core infinispan functionality is implemented in several components:

- JGroups transport layer used to form the cluster reliable communications between members provides information about topology change
- RPC framework on top of the transport layers each action is modeled as a separate *command object*, that is replicated and transported to the target cluster members, encapsulates cahce operations
- DataContainer The main internal data structure which stores entries
- LockManager An interface to deal with all aspects of acquiring and releasing locks for cache entries.

IoC container that stores many other components CommandsFactory EvictionManager ExpirationManager InterceptorChain TransactionManager RpcManager LockManager PartitionHandlingManager etc.

## 4.5 JGroups

Java library for reliable clustering. It is used by Infinispan as the transport layer. It consists of:

- abstract channels (JChannel)
- stack of protocols (up, down), on the lowest level are TCP or UDP sockets

Protocols http://www.jgroups.org/manual/html/protlist.html

- transport - UDP, TCP
- discovery PING, TCPPING, MPING
- merging finds nodes merge node leaves cluster & then joins again (MERGE3)
- failure detection FD_ALL multicast heartbeat FD_SOCK
- reliable transmission ordering NAKACK, UNICAST3
- message stability STABLE
- group members GMS - leaves, joins
- flow control UFC, slowing down
- fragmentation FRAG2, split & reassebles large msgs synchronous sending RSVP state transfer STATE_TRANSFER, STATE

API overview

- Group - A group is identified by its name. Groups do not have to be created explicitly; when a process joins a non-existing group, that group will be created automatically. Processes of a group can be located on the same host, within the same LAN, or across a WAN. A member can be part of multiple groups.
- Adress - class uniquely identifying a node
- JChannel - This is the core interface of JGroups. Requires stack configuration (e.g. using an xml file). Connect method takes group name (uses UDP multicast for discovery) or adress of an existing member. Send method requires an adress (from view) to unicast, multicast does not.

- View - nodes receive view objects containing a snapshot of the current list of nodes, changes each time a node joins or leaves the cluster. First node in the list is the coordinator.
- Operations connect to cluster send message receive message

TODO, config xml for Infinispan, org.infinispan.remoting.transport.jgroups

## 4.6 Remoting

**ReplicationQueue** Periodically (or when certain size is exceeded) takes elements and replicates them. Queue of Replicable Command objects, the core of the command-based cache framework. Commands correspond to specific areas of functionality in the cache.

**Transport** An interface that provides a communication link with remote caches. Also allows remote caches to invoke commands on this cache instance. Has a number of invokeRemotely* methods, sync and async, that invoke an RPC call on other caches in the cluster. Returns a map of responses from each member contacted. Uses a Total Order protocol. Remembers if this cache is a coordinator, current adress, adresses of members in the cluster view. It is implemented as JGroupsTransport, which uses JGroups to transmit command objects to the other nodes.

## 4.7 Commands

All cache operations are represented as command objects, both locally and as a form of RPC can be sent to other nodes in the cluster. ReplicableCommand is the core interface of the command-based cache framework. Commands correspond to specific areas of functionality in the cache, and can be replicated using the org.infinispan.remoting.rpc. RpcManager. Main method is *Object perform(InvocationContext ctx) throws Throwable*, which performs the primary function of the command. Infinispan uses visitor (interceptor) pattern, to handle the command in several steps. This method will be invoked at the end of interceptors chain. Every command has an unique ID that identifies it when serialized to be sent via the network.

# 5 Model checking tools (techniques?)

There are several types of techniques for model checking and tools that implement each approach. After considering the available options, one is chosen for the final implementation.

## 5.1 Model checking techniques

- Explicit-state Model Checking - TLC (TLA+/PlusCal), SPIN (Promela - Process/Protocol Meta Language)
- Symbolic Execution - Java Path Finder (Java/bytecode) http://babelfish.arc.nasa.gov/trac/jp
- JPF Implementation of the JVM, executes the bytecode directly.
- SPIN Processes that communicate via channels (message passing)

# 6 TLA+

In this chapter TLA+ is introduced.

## 6.1 Basic Mathematics

TLA stands for the Temporal Logic of Actions, but it has become a shorthand for referring to the TLA+ specification language and the PlusCal algorithm language, together with their associated tools.TODO reword TLA+ is especially well suited for writing high-level specifications of concurrent and distributed systems. [cit, web].

TLA system (TLAS) is complex, so only selected parts are described.[1] but the main idea is to enable users to express mathematical formulas using a clean and human-readable LaTeX-like syntax. TLA+ is based on the idea that the best way to describe things formally is with simple mathematics, and that a specification language should contain as little as possible beyond what is needed to write simple mathematics precisely. [cit] The are several built-in mathematical operators for expressing:

- Equality,
- Set Theory (Zermelo–Fraenkel [cit]) operations,
- Propositional, Predicate and LTL logic theorems.

With a definition operator, user can provide additional mathematical objects and operations. The basic TLA does not contain arithmetic operations on natural numbers, for example, and their definitions must be imported from a standard module library. Every data structure is *conceptually*[2] a set. This includes natural numbers, that are defined using Peano axioms. The library modules include: Bags, FiniteSets, Integers, Naturals, Peano, ProtoReals, Reals, and Sequences.

## 6.2 Specifications

Model specification is in the TLA system defined using a single temporal formula, most commonly

$$Spec \triangleq Init \land \Box[Next]_{\langle vars \rangle}$$

1. TODO explain
2. For performace reasons, this may not be true for an actual implementation of TLAS.

written as:[3]

```
Spec == Init /\ [][Next]_<<vars>>
```

This formula must hold for all executions, therefore specifying exactly that transition system from an abstract space of all possible systems that represents the model. The *Init* formula establishes a set of initial states and the *Next* must be true for all subsequent states. However, if this was an ordinary LTL formula, it would not be possible to describe how „information" can change between subsequent states. To solve this problem, TLA extends LTL by allowing to reference not only variables present in the current state, but also the variables in the next state. This is done by „priming" the variable name. For instance, if we define:

$$Init \triangleq x = 0$$

$$Next \triangleq x' = x + 1$$

we are specifying a variable that must be incremented during each step.

3. In subsequent text, monospace font is used to display related source code.

# 7 PlusCal

PlusCal is an algorithm language that can be translated into an expression in TLA+. The purpose of PlusCal is to make writing of specifications easier, since it is often simpler to express an algorithm in an imperative way using a C-like or Pascal-like syntax. This chapter provides an overview of the language as well as an explanation of the ideas behind the translation process.

## 7.1 Control Flow Graph

The main idea behind PlusCal is to convert a collection of atomic algorithmic steps, usually variable assignments, and control flow statements, most importantly the conditional `if` and unconditional `goto` jump into an abstract control flow graph.

Control flow graph (CFG) is a representation, using graph notation, of all paths that might be traversed through a program during its execution. [cit, wiki]. Nodes in CFG represent a block of code that is executed sequentially, and directed edges represent possible control flow changes. Node should contain at most one logical control flow statement, e.g `for(...)`. Edge starts in a node with such a statement, and ends with a node containing a possible jump target.

In Figure 7.1, a PlusCal algorithm for the hour clock problem described previously [TODO ref chapter] is presented. Note that it starts with a declaration of global variable `hr`, and contains two control flow statemens (infinite `for` loop and an `if` conditional) and two assignments. In addition, the algorithm contains labels (such as `s01:` to mark „atomic" statements, that represent a single step in the resulting TLA+ execution. As a result, there must be at most one assignment to the same variable per label. In addition, they are also used to designate nodes in the CFG, and therefore are required at places where the control flow changes.

The final graph, shown in Figure 7.2 contains five nodes (there is always a special *Init* node) and 6 edges. Each of these nodes is translated into a separate TLA+ definition. To determine which path in the algorithm is taken, an additional `pc` variable (program counter) is defined for each process (PlusCal can be used to define multiple concurrently executed processes) and works like a instruction pointer register in common processors. Each TLA+ node-definition contains a condition, that requires the program counter value to contain a name of the label it implements. Moreover, it

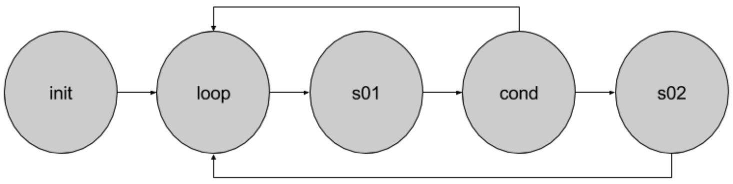
Figure 7.1: PlusCal algorithm for hour clock

```
──────────────── MODULE HourClockPC ────────────────
EXTENDS Integers

  Based on the HourClock TLA+ example in Ch. 2 of “Specifying Systems”

--algorithm HourClock{
  variables
    hr ∈ 1 . . 12 ;     hr is a randomly chosen integer in range 1 . . 12
  {
loop:  while ( TRUE ) {
s01:     hr := hr + 1 ;
cond:    if ( hr > 12 ) {
s02:       hr := 1 ;
         }
       }
  }
 }
```

Figure 7.2: Program Graph for hour clock algorithm



sets the counter value to enable transition to one of the subsequent nodes in CFG. As a result, the final TLA+ `Next` formula is a conjunction of statements for each label, as prezented in Figure 7.3.

## 7.2 Readers-Writers Problem

Readers-writers is a classical „problem of the mutual exclusion of several independent processes from simultaneous access to a *critical section*. [...] There are two distinct classes of processes known as *readers* and *writers*. The readers may share the section with each other, but the writers must have exclusive access.” [Curtois] Therefore, the critical section contains read or write operations on a shared variable performed by readers and writers, respectively. There may be additional liveness and fairness constrains, resulting in several possible solutions.

- [wiki -> Courtois et al.]
- first readers-writers problem, in which the constraint is added that no reader shall be kept waiting if the share is currently opened for reading. This is also called readers-preference
- This is the motivation for the second readers-writers problem, in which the constraint is added that no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called writers-preference.
- Therefore, the third readers-writers problem is sometimes proposed, which adds the constraint that no thread shall be allowed to starve; that is, the operation of obtaining a lock on the shared data will always terminate in a bounded amount of time.

Readers-preference

## 7.3 Dining Philosophers

TODO

- Demonstrate deadlock detection

Figure 7.3: Translated PlusCal algorithm for hour clock

```
VARIABLES $hr,\ pc$

$vars \triangleq \langle hr,\ pc \rangle$

$Init \triangleq$  Global variables
          $\land\ hr \in 1 \,..\, 12$
          $\land\ pc = \text{“loop”}$

$loop \triangleq$  $\land\ pc = \text{“loop”}$
          $\land\ pc' = \text{“s01”}$
          $\land\ hr' = hr$

$s01 \triangleq$  $\land\ pc = \text{“s01”}$
         $\land\ hr' = hr + 1$
         $\land\ pc' = \text{“cond”}$

$cond \triangleq$  $\land\ pc = \text{“cond”}$
          $\land$ IF $hr > 12$
                THEN $\land\ pc' = \text{“s02”}$
                ELSE $\land\ pc' = \text{“loop”}$
          $\land\ hr' = hr$

$s02 \triangleq$  $\land\ pc = \text{“s02”}$
         $\land\ hr' = 1$
         $\land\ pc' = \text{“loop”}$

$Next \triangleq loop \lor s01 \lor cond \lor s02$

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

# 8 [Infinity DHT] Model Design

In this chapter, top-down desing of the model is presented.

## 8.1 High-level design

TODO this is no longer vaid

System is an interaction between a cluster of data (server) nodes a set of clients issuing read and write requests to each data node. More formally, system is modeled as a 4-tuple Q=(C, S, N, M) where

- C is a set of client nodes
- S is a set of server nodes
- N is the network which handles the communication
- M is the set of possible messages that can be transported over the network

Let X be some well-defined component of the sytem. states(X) be a set of all possible states that X can be in. Because the entire system must be a finite-state machine with a reasonable size, we must have a mechanism to estimate the state space size. We can encode each state in a unique a bit-string of length log2(|M|). Because the model represents a storage system, it is not possible to model data table with a reasonable capacity and bounds. Therefore the parameters that influence |states(Q)| must be carefuly selected. This includes the size of the separate components of Q, such as the number of nodes.

TODO expand

We will now explore components M of the system in increasingly more detail, select parameters that which we want to include in the model and estimate their effect on |states(Q)|.

## 8.2 Network

What type of network is it useful to simulate? -> connectionless datagram network like UDP, not TCP (although that is available for infinispan)

## 8.3 Network Reliability

Communication over a network should be as reliable as possible. In a perfect world, nodes in a distributed system should always succeed in sending messages to others and in receiving them back. In reality however, communication protocols must deal with messages arriving late, in a different order than they were sent, or not at all. When designing and specifying such protocols, it is important to consider which failure modes the networking part of the model should simulate, especially when keeping the number of its states as small as possible is a necessity. On the other hand, to be useful, the model must be verified under failure conditions that occur in the real world.

- The network model simulates *message order change*, but not *latency*. This means, that if some message has been sent to a node, it shall not receive a „no message available" signal. However, if there is more that one packet „in the network", any of them may be delivered first.
- It is necessary to simulate network partition and recovery, but random packet loss is not simulated.This may be included after all. This means that network connection between nodes may be lost only in a specific points in the model execution. TODO explain, Two Generals problem, TCP/IP in real world?
- *Double delivery* is not simulated. This increases model complexity and can be avoided in implementation, by using suficiently large monotonic sequence to mark packets.
- Undelivered packets are removed on connection loss.

## 8.4 Message Model

Message structure is inspided by UDP/IP packets. Each message contains three fields: recipient, sender and data. The data type is therefore $msg \in (AddressSpace \times AddressSpace \times DataSpace)$. There is a special singleton $NoMessage$ message for signalling that the network has no packets available for the requester. This includes a case of network failure. Because TLA requires sets to be enumerable, it is important to estimate numer of possible messages: $|AddressSpace|^2|DataSpace|$. If we need 3 nodes and one unclaimed address to model simplest DHT system, and consider read and write operations for 12 possible keys and 2 possible valuest, the lower bound is 4^2 * 2 * 12 * 2 = 768 messages. The content of the network may be up to 2^768. We will have to make sure this will we far less in reality. Eg. cluster-wide write locking (max NodeCount write messages), discovery-protocol-locking, artificial packet cap (max number of „in network" messages for each connection, e.g. 3: 3 * N * (N-1) = 3 * 4 * 3 = 36, i.e (768 choose 36) + (768 choose 35) + ... + 1 =~ 9.17*10^61?

## 8.5 Network Model

From an highly abstract (not a good term) point of view, computer network is an undirected graph, with nodes representing addressable elements and edges connections between them. Since the network has not only a topology, but also a state, each .

Constructing simplified model is necessary. First, topology - it is unnecessary to consider internal structure of the network, including routing elements. It can be simplified into a subgraph of a connected graph. Additionaly, if there is a path from A to C via B, it is good to assume that A to C is also connected. Algorithm for relay can be easily implemented, and there is no reason to not use it in a general case. Therefore, each connected component becomes a complete graph. This information can be more effectively stored as a set of network partitions, each partition is a set containing nodes, where a single node must be in exactly one partition. If we remove a need for routing (better to say forwarding) nodes, network state may be simplified to a list of messages that are being tranfered via each connection. However, it would be overkill to assign a buffer to each connection, since the messages already contain receiver and sender addresses, and therefore all messages may be stored in the same buffer for the entire partition. However even more simply, since we already have the information whether sender and receiver are in the same partition, we can actually store all messages that are waiting to be received in a single set for the entire network. We can therefore check if the message is stored in the buffer or thrown away. To keep the buffer smallest possible, if a partition happens, throw away now undeliverable messages.

Does this conform to the previously stated failure constraints?

Therefore: (copy paste parts of network spec)

„Operations" (TODO define as special predicates):

- send (packet/data)
- receive

- internal clean
- do_partition() // args?, original, new one?
- do_nondet_partition
- do_mend + nondet
- broadcast // should be modeled

Space size estimation -> limit this, (some states may not even be reachable, or make sure they are not)

What to do if network capacity reached? -> throw away (can we do that?)

## 8.6 Node

TODO how node is structured, overview

## 8.7 View Distribution Protocol (Leader Election)

## 8.8 Distributed Read Protocol

See readers-writers problem

## 8.9 Distributed Write Protocol

see readers-writers problem

## 8.10 Distributed Read and Write

In one of the previous chapters, we have described a solution to the problem of ensuring that concurrent read and write access to the shared variablecon-sistent

## 8.11 Paxos

Paxos is a family of algorithms for reaching consensus in a distributed system operating in an unreliable environment. Given a set of nodes (processes), a consensus algorithm ensures that a single one among the proposed values is chosen. [paxos made simple] The safety requirements for consensus are:

- Only a value that has been proposed may be chosen,
- only a single value is chosen, and
- a process never learns that a value has been chosen unless it actually has been. [cit]

To describe the algorithm, three roles are used: proposer, acceptor, and learner. In many versions of Paxos, nodes asume more than one role. For the purposes of DHT design, where the nodes are equal and run the same software, this results in all nodes having each role. There are following assumptions about unreliability of nodes and the network:

- Each node runs asynchronously at different speeds and may stop or restart.
- Messages may be delayed, lost or reordered, but there are no Byzantine errors (they may not be corrupted).

The central idea of the Paxos is to extend three-phase-commit protocol (3PC) with the concept of quorums.

A single round of 3PC consists of three phases, with participating nodes having two possible roles, *coordinator* or *agent*. Coordinator is the initiator and manager of the transaction, while agents can vote to *accept* the transaction and *learn* about the results.

In the first phase, coordinator receives a request to initiate a transaction, and sends <prepare> messages to all agents, containing the transaction information and waits for the resposes, or aborts on time-out. Agents receive the message, and decide whether they are ready for the transaction to start, and send <accept> or <reject>

Quorum is a subset of acceptors, large enough that any such subset must have at least one acceptor in common with the choice in the previous round. This ensures that there is always some acceptor that has learnt the newest value and can rejects an invalid proposal.

[–]

Second, getting a consensus protocol right is hard. Simple solutions don't work very well, exhibiting undesirable behaviours and occasionally acting incorrectly. Mike Burrows, inventor of the Chubby service at Google, says that "there is only one consensus protocol, and that's Paxos" – all other approaches are just broken versions of Paxos.

# 9 Execution and Analysis

TODO:

- Verification of standalone modules
- Limitations on state space
- Test on Amazon AWS, x1.16xlarge 64vcpu 976gbram 1 x 1920gb SSD, $6.669 per Hour = 168.16czk

# Bibliography

[1] BAIER Christel, KATOEN Joost-Pieter. Principles of Model Checking, Cambridge, Massachusetts: The MIT Press, ISBN 978-0-262-02649-9.

# Appendix A
# Archive Content

This archive contains contents of the project's git repository, currently hosted on GitHub:

- TODO