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.)

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 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)

4 Infinispan design

There are three main caching techniques that can be deployed, each with their own pros and cons.

- Write-through cache directs write I/O onto cache and through to underlying permanent storage before confirming I/O completion to the host.
- Write-around cache is a similar technique to write-through cache, but write I/O is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write I/O that will not subsequently be re-read, but has the disadvantage is that a read request for recently written data will create a "cache miss" and have to be read from slower bulk storage and experience higher latency.
- Write-back cache is where write I/O is directed to cache and completion is immediately confirmed to the host. This results in low latency and high throughput for write-intensive applications, but there is data availability exposure risk because the only copy of the written data is in cache.

4.1 Overview

Infinispan is at its core a distributed in-memory hashtable.

More complex features are built on top of that:

- persistence (file storage)
- Apache Lucene queries (fulltext indexes)
- map/reduce engine
- enterprise integration (Java EE servers, CDI, Spring)
- management

There are two main use-cases (modes):

- embedded - used as a library in a Java SE application
- server - standalone deployment (in an application server) that the clients communicate with using REST API or binary protocol

4.2 Concurrency 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. In this mode Infinispan is typically operating as a local cache
- invalidation (write-around cache) - where all entries are stored in a permanent store (such as a database) and cache is used to alleviate intensive read operations. When a node needs the entry it will load it from a cache store. 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 is typically operating as a data grid or a temporary data store, but doesn't offer an increased heap space.
- distribution - where entries are distributed to a subset of the nodes only. In this mode Infinispan is typically operating as a data grid providing an increased heap space. Uses topology aware consistent hashing, 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.

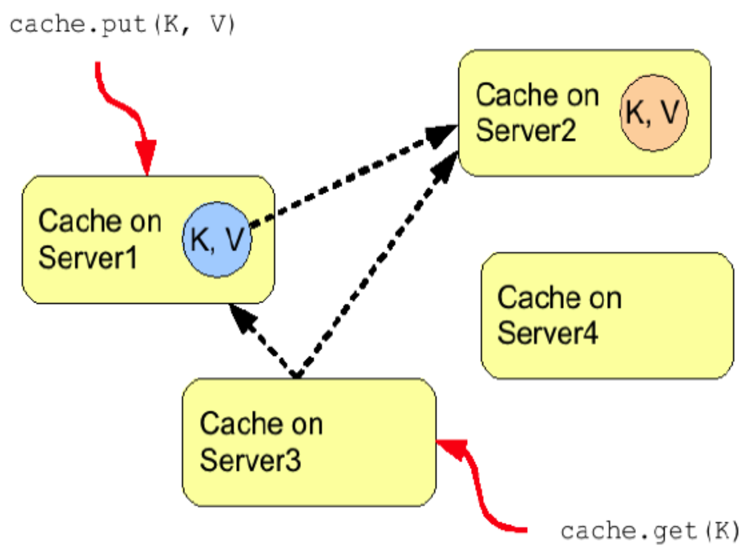
Last three options can use synchronous or asynchronous modes. In synchronous mode, client is blocked until the operations complete and are confirmed.

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 Modeling languages and tools

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)

TODO

6 Model Design

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

6.1 High-level design

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)|.

6.2 Network

Physical computer network can be abstracted into a graph G with vertices representing nodes, edges possible connections: V(G)=(C\cupS\cup). Physical networks also consist of routing and other devices, however for our initial attempt, we can let the graph be a clique and simulate network failures by removing the edges or otherwise changing their the properties.

Properties to check:

- unreliable message delivery (relation to JGroups?, view change?)
- message dropping
- asynchronous delivery, e.g. unordered messages
- delay - network partitions

These can be modeled by adding a non-deterministic decision to drop a message, and a decision to delay a message up to a specified amount of time. If the network edge is modeled as a priority queue with delays for each direction, this would also model message reordering. This queue must be size-limited and excessive messages dropped.

Not-well formed messages would be hard to model, and since the network layer is implemented using JGroup which provide some guarantees to Infinispan, it depends on the level of detail that JGroups will be modelled.

Size and the resulting capacity of the must be kept reasonaly small, as well as the total number of possible messages.

6.3 Data node

Data node contains:

- The hash table
- Locks and synchronization mechanisms
- Network command receiving and execution loop

6.4 Client node

The purpose of client nodes is to connect to data nodes and issue read-write operations. Verification of the system properties uses both the state of the clients when issuing a command and the resulting effects on data nodes and other clients that may read the data.

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