# Task distribution framework using Zookeeper

Using hierarchical state machines to design and implement Zookeeper recipes

Albert Yera Gómez

Master of Science Thesis
Stockholm, Sweden 2013

TRITA-ICT-EX-2013:10

# Abstract

To develop a distributed system is not an easy task. Not only do we need to understand the problems that may arise and solve them, but also we need to know how to implement them. The first part of the thesis tries to build some general knowledge about distributed systems. The second part shows how to use the ZeroMQ library in order to create an actor library for C++ and how to use the Zookeeper service in order to implement reliable distributed systems. The main contribution of this thesis is a novel method which uses hierarchical extended state machines in order to improve how to model Zookeeper's algorithms. As a proof of concept, an internet scale task distribution framework is described and prototyped. However, everything started in the reverse order. The purpose of the thesis was to create a task distribution framework, and things were discovered while trying to develop it.

# Terminology

Distributed systems have existed for more than 30 years. Although many papers have been written, there is no a common vocabulary framework. Terms like node, process or actors are mixed. We find important to give an exact definition of the terms we are going to use through the thesis, in order to dissipate any confusion.

A **machine** refers to a single computational resource, which may have a different number of processors and different amount of **locally shared** memory. It refers to a physical computer.

Inside a machine, a **process** is the basic unit of concurrency. Each process may have internal units of concurrency, which we'll call thread. We'll treat threads as the atoms of concurrency, although we will see later on that it is possible to create even smaller units of concurrency.

A **node** will be anything (process, thread, or subunit of thread) that can perform calculations and send and receive messages. This definition will be expanded.

Many charts are going to be used to make it easier to understand the different concepts presented. All of them will follow the same convention:

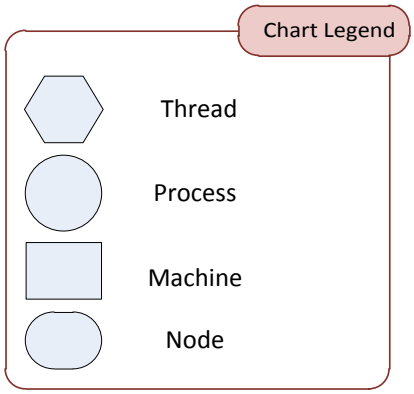



Figure 1 Chart Legend

# Acknowledgements

To David Rijsman for its uncoditional support and for trusting in me from the first moment we met

To Marcel Fransen for answering all my endless questions about everything

To John van Roij and Johan Montelius for supervising my work

To the Quintiq R&D department, just because all the people there are awesome, with special thanks to Catalina and Alejandro

A la meva família per animar-me des de tan lluny

A Dani y Teo, por haberme acompañado durante todo el tiempo

# Contents

# List of Figures

# 1 Introduction

The main purpose of this thesis is to research alternatives and prototype a task-distribution framework. The goal of such a framework is to distribute discrete units of work, the so called tasks, to workers which may be on the same machine or spread through different machines, making it is possible to increase the computing power of a system using cheap desktop computers. These tasks will be **self-contained** and **independent.** With self-contained we mean tasks where all the information is embedded in the message invoking the task (there is no need to share data between machines). With independent we mean tasks that do not have dependence relations between each other: the order of execution is non-important and outputs from one task will not be inputs for others.

We will see that these kinds of frameworks are usually thought to be executed in highly controlled environments (like clusters in private LANS) where specialized hardware is used. Our purpose is to propose an internet scale task-distribution framework with no single point of failure, where nodes are regular desktop computers and help each other in order to distribute the tasks among them. In an internet scale environment, nodes have a high churn rate (they may appear and disappear from the network at any point of time due to connection problems, machines crashing or users disconnecting).

However, programming such a framework is not an easy task. Much attention has been devoted to the process of programming a distributed system using a general purpose programming language (C++): Which technologies to use, how to use them or which programming abstraction is the best to program these kind of systems are some of the questions that will be answered.

In order to facilitate the development of the framework, a general distributed system library will be created using ZeroMQ and Zookeeper. As it will be shown, the library allows to program many kinds of different systems, and will ease the process of prototyping the task-distribution framework. The prototype will be able to run CPLEX[1] algorithms distributedly, although the service is abstracted in such a way that the user can program its own tasks.

The first section acts as an introduction to Distributed and Parallel systems, explaining their properties and focusing on the problems that must be solved. It also explains how researchers try to understand those systems using different **programming models**.

[1] CPLEX is an optimization software package used in Mathematical Programming

Another important purpose of the thesis is to introduce the company Quintiq into the distributed systems world. An overview of the Quintiq system architecture is provided in the second section. In this section we will also devise where distributed systems concepts and solutions could be applied in its architecture. From it we will see that the task distribution framework is a good candidate, and we will list all the requirements that Quintiq has.

The third section studies different solutions available in the market in order to learn how are they structured and which facilities do they provide. The solutions studied are: Grid Engine, Condor, MapReduce, Dryad and BOINC. After this study, we will see that although some frameworks fit our needs, they introduce too many complexities to our system and could not be used in other parts of the Quintiq architecture.

The fourth and fifth sections try to be a guide to implement and interpret distributed systems, and shows all the steps carried to implement the distributed system library. As we said, two third party libraries are used: Zookeeper (which provides coordination primitives to implement fault-tolerant applications) and ZeroMQ (which can be used as a message passing and a threading library). Although ZeroMQ has a fantastic documentation, Zookeeper's one is not that good. We have carefully analyzed Zookeeper paper and API, and an effort is made to understand the guarantees that Zookeeper provides and to explain how to successfully use its API. **We will present a novel methodology to design and implement Zookeeper algorithms (recipes) using hierarchical extended state machines as its base foundation.**

Finally, the task distribution framework is presented as a Zookeeper algorithm.

# 2 Background

A task distribution framework can be treated from mainly two different perspectives: from the parallel systems or the distributed systems perspective. In **parallel computing**, referred from now on as massively parallel (MP) systems, a large number of computers (each with multiple processors) are interconnected so that each one can work simultaneously to solve a smaller part of a problem that is too big (time-expensive, big data[2]…) to be solved by just one computer. The main purpose of an MP system is to increase the computational performance relative to the performance obtained by the serial execution of the same program. That is, if we have a sequential problem that can be divided into smaller parts, then we can run these parts in parallel:

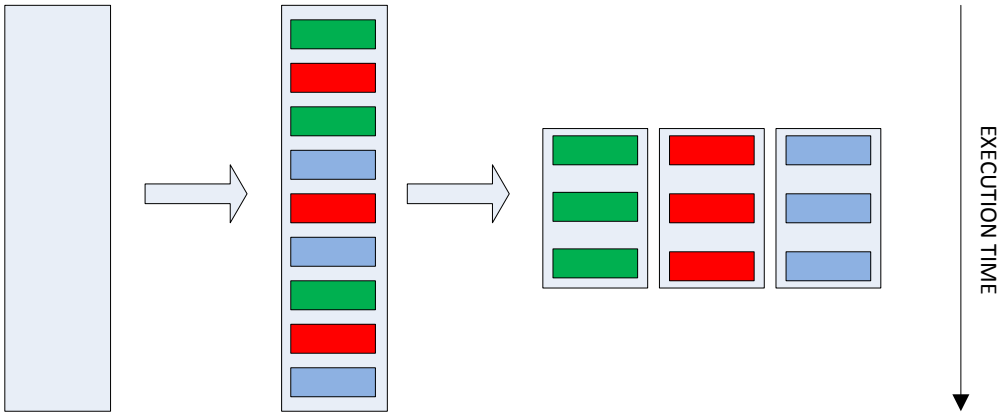



Figure 2 From sequential to parallel problem

Making them run in parallel allows us to run them faster using more resources (threads, processors, machines…). These resources are materialized in hardware architectures, which nowadays follow three different architectures, all of them used with Multiple Instruction Multiple Data (MIMD)[3] processors: shared memory, distributed memory and hybrids (combinations of shared and distributed memory). The hybrid approach is commonly referred as Symmetric Multiprocessing (SMP) cluster.

A **distributed system** can be described as a set of nodes, connected by a network, which appear to its users as a single coherent system. The key-element is that the only way to communicate between nodes is through message-passing (sending messages over the network). We can see that this definition is much more open than the MP one, in the sense that it encompasses more possible systems.

[2] Petabytes or Zettabytes of data cannot be stored on an individual machine
[3] Each processors is independent from the others, executing its own subprogram

Other definition worth to mention of a distributed system is the one given by Leslie Lamport, one of the fathers of this area:

> "A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable"

For the end-user (which in this case will be an application programmer), all the complexities derived from the fact that a system is distributed should be hidden. If a distributed framework solves (and hides) a determined "complexity", it is said that provides a **transparency.** The ISO Reference Model for Open Distributed Processing (RM-ODP) [1] defines the following transparencies:

- **Access transparency:** Makes the differences between heterogeneous nodes (operating system, different hardware and data representation…) to be unnoticed, for both the users and developers.
- **Location transparency**: Applications do not need to specify the node where a specific service or data is located.
- **Migration transparency:** A specific functionality or value can be moved from one computer to another while the system is running. The programmer just needs to trigger the movement, but does not need to program the logic to perform it. A good example can be found in the Agent programming model (see page 7).
- **Relocation transparency:** When a value or functionality is migrated, other components of the system should somehow be notified about this movement, and references to what was migrated must be updated.
- **Replication transparency:** Different nodes can support the same interface. Clients can execute the interface without needing to know who or what will execute the request.
- **Persistence transparency:** The state of objects is saved and restored within the system.
- **Transaction transparency:** When multiple nodes try to access to the same data, it is necessary to restrict their concurrency to prevent unexpected outcomes. The result of those operations should be consistent.
- **Failure transparency:** Hides faults, errors and recoveries from the affected nodes.

Distributed system toolkits hide complexity providing different levels of transparency to the programmer.

Besides the different transparent provided, we can distinguish two different groups of applications in which distributed system concepts apply: the ones that have **inherent distribution** and the ones that create **distribution as an artifact** [2]**.** In the first group we find information dissemination (publish-subscribe paradigm), process control (processes connected to various sensors, which may cooperate to control the global process), cooperative work (the users of a service may be in different locations) and distributed storage and databases (like the big-data mentioned before). In the second group we find applications that are not inherently distributed, but use distributed abstractions to satisfy specific requirements. Among this requirement we can find fault-tolerance, load balancing or fast sharing.

One important conclusion from these descriptions is that Parallel and Distributed systems overlap in many aspects. It is really important to find and understand the differences to proceed.

The main difference is found in the system environment. Distributed systems usually consist of a set of heterogeneous workstations (different processors, amount of memory, different OS or connectivity), whereas an MP system is purchased and tailored for high-performance parallel applications, where all the machines are more or less homogeneous (furthermore, these machines are not full-fledged workstations, but ripped versions of them with normally just a CPU, memory and a network interface). Hence, an MP must be used in highly controlled environments: The network of an MP system is typically contained in a single room, and the nodes are arranged in scalable topologies to minimize the distance between processors (hypercube, torus..).

Distributed systems can span a single room, a building, a country or a continent[4]. Moreover, this network often consists of several physical media (Ethernet, telephone network, satellite connections) and different transport stacks (IP/TCP, ATM, PGM…). It is normally impossible to predict how the nodes are interconnected. We just know that they are connected. Even though it is not possible to control the physical connection of nodes, many Peer to Peer (P2P) research is focused on how to control the logical connection between nodes (creating an overlay network), being possible to minimize network hops, latency, or any other metric. If the reader is interested, one of the most exciting papers is T-Man, a gossip-based overlay topology manager [3].

However, we are not longer going to talk about P2P systems. They are a subfield of distributed systems, where all the nodes behave both as server and

[4] Some of them, span the entire World, like the Spanner: Google's Globally-Distributed Database

client. Without a central server, there is no single-point of failure in the system. It is not the aim of this thesis to create a task-distribution framework over p2p networks. **Nonetheless, the no single-point of failure characteristic must be considered in the design of our framework.**

Consequently, MP can exploit parallelism at a much finer granularity than distributed systems (they try to parallelize low level structures of a programming language, like for-loops). To do this, nodes need to communicate more frequently than distributed systems, and thanks to the high controlled network infrastructure, they can. Even so, MP systems are more **static** than distributed systems. From any given system, it is given at boot time the number of nodes there are and their configurations. If some of the nodes crash, the basic approach is to replace the faulty component and restart the system. A distributed system must have the ability to tolerate changes: nodes come and go[5] (many things can fail, the nodes themselves or the network) and may change their configuration. The last point is that it has also the potential to contain thousands and hundreds of thousands of nodes.

Before explaining the basic requirements of our task-distributed framework target, it is also important to summarize the existing programming models for both parallel and distributed systems:

## 2.1 MP systems programming models

- **Message-Passing model:** Processes send and receive messages to communicate with other processes. Messages are the only way to share data or state between processes. Message Passing Interface (MPI) is the standard API used for message passing. However, we have seen that MP systems always live in controlled environments, which many times contain many processors sharing memory. MPI cannot take advantage of this shared-memory, and is mainly used for program-level parallelization (*large-scale*). One of the most well-known implementations of MPI is OpenMPI. In fact, MP systems using MPI can be considered as a subset of a Distributed System. The main disadvantage of the MPI standard is that it is not fault-tolerant. OpenMPI provides from version 1.3 some kind of fault-tolerance (application checkpointing, see section 4.1.2), but this is often not enough, and it is not compliant with the MPI specification, so it cannot be used with other MPI implementations.
- **Directives Based Data Parallel Model:** Programming languages make serial code parallel by adding directives that appear as comments in the serial code. These directives tell the compiler how to distribute data and work across the

[5] This fact is so important that has its own name: **churn rate**

processor, and are mainly used for parallelizing loops (*small-scale*). This model is mainly used in shared-memory space.

- **Hybrid approach:** A newer approach tries to take the best from MPI and shared memory, but the programming model tends to become extremely complex, so we will not discuss it any more.

## 2.2 Distributed systems programming models

- **Shared data:** Values (data) appear to be directly accessible from multiple nodes. This model is based on the shared-memory paradigm, also used in MP systems, where a global memory available for all the nodes in the system is *emulated* (normally using message-passing). Being able to emulate the shared data programming model using message-passing techniques has an important consequence: It is possible to use all the shared data algorithms in distributed systems.A different thing though, is how well (or bad) do they perform.
- **Message-passing:** Again, we find the ubiquitous message-passing technique. Upon this technique it is possible to build the **actor model**. It is a mathematical model that treats actors as the primitives for concurrent computation. Hence, if threads were the atoms of concurrency, then actors can be thought to be the electrons. A thread can execute more than one actor in parallel. An actor is a reactive entity. In response to messages that it receives, an actor can change its local state, create more actors and/or send a message. In fact, the actor model matches perfectly to one of the most used theoretical models to study distributed systems: the one proposed by Attiya and Welch, which treats nodes (or actors, since now we see that they have the same definition) as state transition systems (STS) [4].
- **Remote Procedure Call (RPC):** Tries to emulate local method invocation. This model creates the idea of services. Nodes implement services, exposing an interface to use them. Other nodes can use this interface from their own thread of execution, and the RPC system will have the responsibility to contact with the service provider and execute the code there. Even though these systems are really easy to comprehend and use, they normally have really difficult implementations, define extensive protocols for the interfaces and communication, and are normally synchronous (a thread executing a remote method will wait until receiving the response). One of the best known RPC systems is JAVA-RMI, although it is language dependent. Language independent implementation also exist, like CORBA and COM, but have a really high complexity. Nowadays RPC systems use some form of SOAP binding (creating the concept of web-service).
- **Mobile agents:** Focuses on the movement of agents throughout the system. Agents are autonomous entities with state that can communicate with other agents in the same local machine. When an agent needs information (or needs

to communicate with another agent) that is not on the local machine, it moves to the machine containing the information/actor. Agents extend the idea of actors and mix it with other disciplines like Artificial Intelligence.

## 2.3 Requirements for the task-distribution framework

The simplest model we can think of a centralized task-distribution framework is the following one:

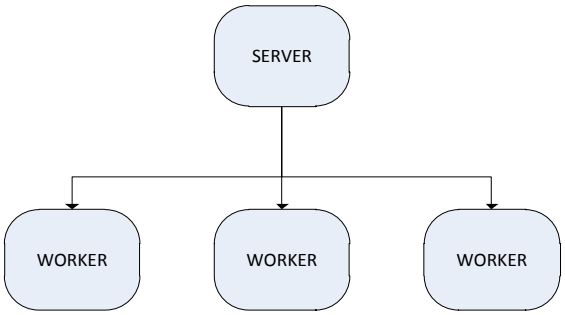



Figure 3 Simple task-distribution model

A server receives tasks and distributes them among some workers using a defined strategy. These are the requirements that we want from this system:

- **Independent of thread/process/CPU/machine:** The server and workers must be treated as nodes, and they should work regardless where are they executed. This allows a really high flexibility in the deployment of the system:
    - If the amount of work is small, or we are working in a machine with a lot of processors, we could deploy it in one process using different threads. In fact, this is exactly what the Java or the C# Executor Framework do.

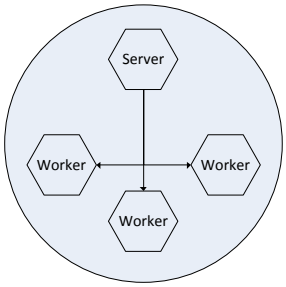



Figure 4 Intraprocess architecture

- Many times a thread is not enough. Maybe the worker is a full-fledged program that needs its own process and manages its own threads. It should be fairly easy to create this architecture:

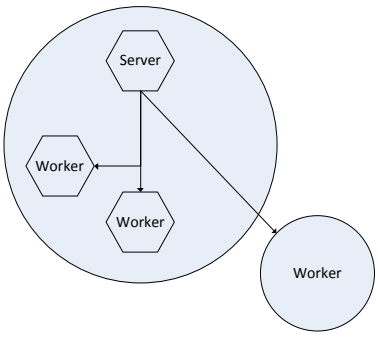



Figure 5 Interprocess communication

- If we want to outperform the capacity of a machine, we will need to place pools of workers in different machines. This should also be possible:

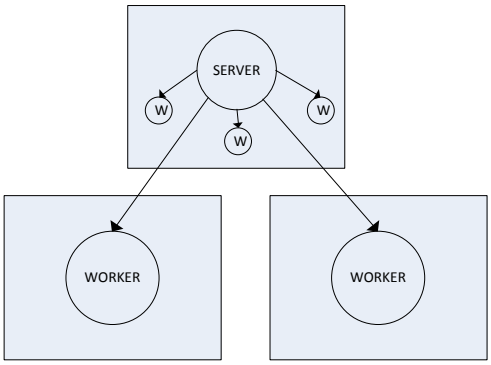



Figure 6 Intermachine communication

- Highly related to the last one, **the framework must support different transport technologies**. The communication cannot depend on one unique transport technology. Since there are many possible deployment environments, each environment should use the best transport mechanism. The architecture in Figure 6 should be able to use TCP/IP or a multicast communication, whereas the one in Figure 4 should be able to use TCP/IP, but also interprocess communication, much faster than TCP. The main point is that we want to use the same (or almost the same) code for all the different topologies. How or where are the worker deployed should be just a configuration step.

The software architecture should be very similar in all the cases, that is, we want to implement the system only once.

- **Heterogeneity:** The system must work with no previous knowledge of the machine's capabilities. It should be transparent to mix 40 core machines using Windows and 2 core machines using UNIX.
- **Dynamicity/Elasticity:** The number of nodes will not be known at boot time. It should be possible to add new nodes to the system to increase its performance without the need of restarting the system. Moreover, this flexibility should be provided without complex configurations. Workers must be plug-and-play components in the system. In the same way, it should be fairly easy to remove a node from the system.
- **Scalability:** The system should support from one worker to thousands of them. Moreover, the performance of the system, theoretically, should increase linearly with the number of machines. Of course, the simple model from Figure 3 does not scale well. The architecture has to evolve to allow such scalability. A common approach to solve this problem is to use message brokers. However, in this system the server is still a bottleneck. It will be quite difficult to remove this bottleneck if we do not want to use a p2p architecture.

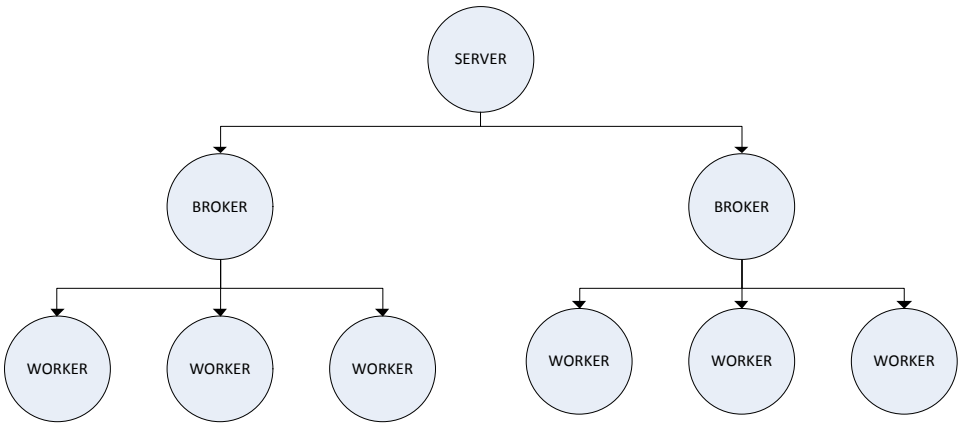



Figure 7 Using message brokers to improve scalability

- **Fault-tolerant:** The system must be reliable. This means that the system should handle a certain set of failures. Unfortunately, there are many possible sources of failure.
    - Application code can fail: It can crash and exit, stop working, run too slow, exhaust all memory… In this case, the application code is the task to be executed
    - System code (worker-broker-server) can fail. Moreover, they may have a Byzantine behavior (respond with non-correct messages or

misusing the communication protocol). We will not consider the Byzantine behavior (although the environment is not highly controlled, it is not hostile).

- The network can fail temporarily, causing message loss. Depending on how the system is implemented, nodes can interpret that other nodes have died when the only problem is the network. Network partitions (causing the division of a connected set of nodes into two or more unconnected sets of nodes) could also happen, and the system should remain in a consistent state.
- Hardware can fail, causing all processes on it to die. We can think that this is not important, because it is not common to happen. Well, we just need to do some basic calculus to show that we were mistaken: Let us suppose that the mean time between failures for one node is 3 years. Then, if the system has 1000 nodes, the mean time between failures in the system will be just 1 day. Every day a machine will fail!
- The end of the world

It is important to try to handle as many failures as possible. In the architecture shown in Figure 7, there is one single failure point, the server. The system should be able to recover from a server crash automatically (**self-regulated system**).

- **Private and public cluster:** We should be able to deploy the system in highly controlled environments, but also in public environments. It should be easy to expand the system in the cloud, or use idle computers at night within a company.

### *From a programming point of view:*

- **Cross-platform:** The framework should work on all the common OS platforms (Windows, Linux and OSX).
- **Language-independent:** There should be bindings for at least C++ and Java. Moreover, nodes programmed with one language should be able to at least communicate with nodes programmed in another language. Hence, the framework needs to have a **formalized message protocol on the wire**. The prototype will be implemented with C++.
- **Easy for developers:** The API should be easy to use. In addition, the source code of the framework should be well documented and easily modifiable.
- **Source available and maintainable**

- **Small footprint:** The framework should be as small as possible, and should not have many dependencies on other softwares or libraries.
- **Licensing:** Quintiq should be able to sell products using the framework.
- **Traceable and debuggable:** The framework should not be a black box. It must be easy to detect, understand and solve problems.
- **Secure:** Communication between nodes should be secure. If the framework does not provide it, it should let the user to put a secure layer above it.
- **Fast communication and low latency**

# 3 Quintiq

Quintiq is a Dutch company that provides software solutions for advanced planning, scheduling and supply chain optimization that help its clients reduce costs, increase efficiency and improve bottom-line results. As a leading Advanced Planning and Scheduling (APS) vendor in the targeted markets (Metals and Manufacturing, Logistics and Workforce), Quintiq offers one standard software package that can provide solutions to all its clients due to increased flexibility and customer configurability.

## 3.1 The Quintiq vision[6]

According to Quintiq's view, APS problems (or puzzles) are best solved by focusing on three areas:

- **Modelling:** although companies may look similar from a general point of view, their particular details make them unique. In order to achieve optimal results for each company, these specific aspects must be taken into account. Moreover, the scheduling problem must be correctly understood and defined for each company. It is therefore extremely important to rightly devise the **business model** (how the planning puzzle looks like) and the **business logic** (the rules and constraints that govern the functioning of the business model). In order to specify both of them, a 5th generation programming language is used, the Quintiq Logic Language (QUILL).
- **Interaction**: the human user or the planning system itself should be able to take into consideration any dynamic information that is relevant to the puzzle. This means that the planning system should not act like a black box, but as a tool that provides full insight on the status of the scheduling procedure, allowing the user to control the decision making process.
- **Optimization**: the system should assist the user in working out the planning puzzle, and even provide a solution when required, by making use of various optimization algorithms.

Careful analysis of the challenges entailed by APS problems revealed that in order to succeed in this field, trying to find a generic planning and scheduling solver, that would be applicable to different companies is not the answer; instead, one should make sure that every conceivable aspect of a business can be modelled,

[6] This subsection is taken from [5], a Bachelor's thesis that was also done at Quintiq.

thus making it possible to build flexible solutions for each client with common tools.

As a consequence of this conclusion, Quintiq emerged not as a system with a standard answer, but as a tool that can be used to specify any business model and build the framework for solving any APS puzzle. This means that Quintiq is implemented with the following characteristics in mind:

- **High level of abstraction**: the entities comprising the business model must be different and clearly separated from the technical ones. In Quintiq, a special application (the Business Logic Editor) provides the tools for specifying the business model, while hiding the implementation details.
- **Declarative business logic:** defining the impact on the model that each possible action might have tends not to scale; to solve this problem, Quintiq uses declarative logic: one describes what should happen, and not how it should be achieved.
- **Standard functionality:** Quintiq defines a very large set of ready to use components. This way, one can concentrate on defining the actual business model, as efficiently and quickly as possible.

From an Object Oriented point of view, both business model and logic specify a class. Instances of it (the actual data) are called **datasets.** Hence, multiple data sets can exist for the same model. Datasets can be **copied** and **merged**.

## 3.2 Dissecting the Quintiq system

The Quintiq software is a three-tier client/server architecture. Figure 8 represents all the elements in a Quintiq system in a typical configuration.

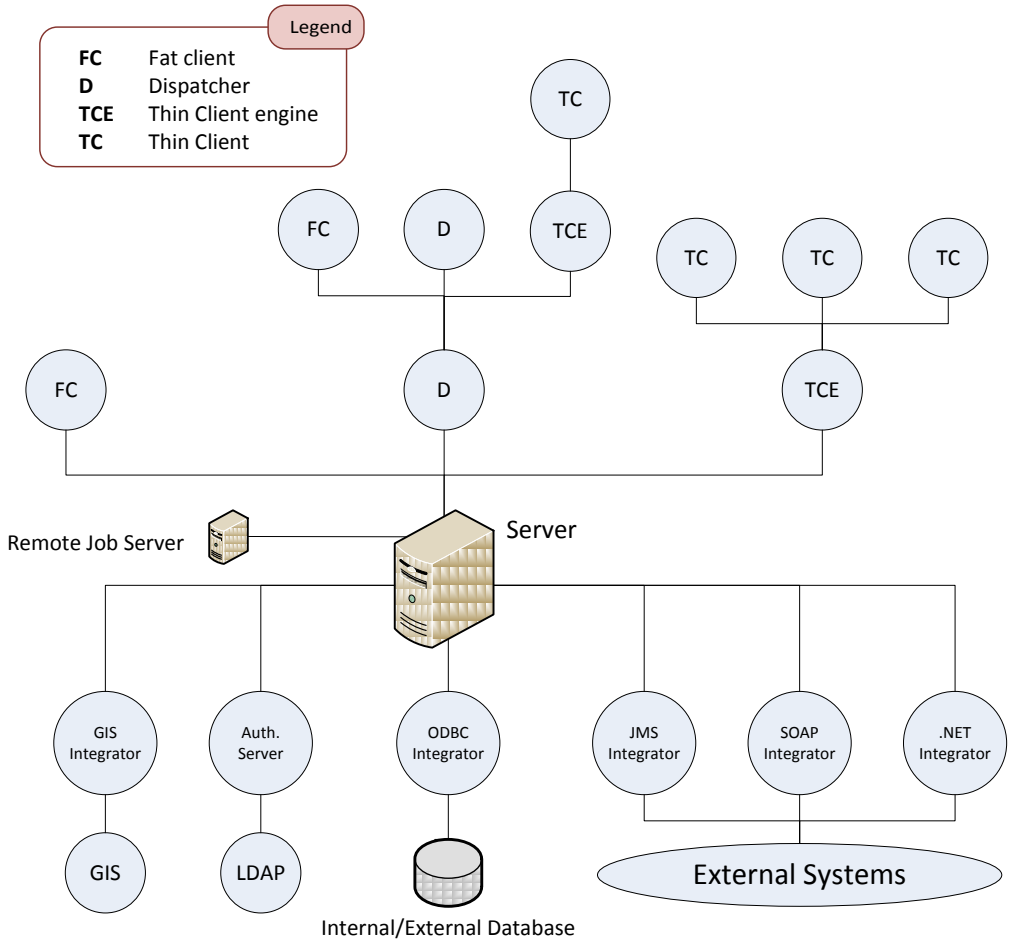



Figure 8 System architecture

The **server** is the central point of the Quintiq system. It contains both the model and all datasets created. It is the only element in the network capable of modifying a dataset. The consistency of a dataset is achieved by sequentializing all the writing operations over them through the server, using a **one writer- multiple readers** locking system. For performance purposes, the server maintains the model and all the datasets loaded into memory.

Different kinds of **integrators** are provided in order to connect the Quintiq server to other external servers. The most common is the ODBC Integrator, which translates the data sets into SQL statements.

### 3.2.1 Clients

Clients connect to the server in order to retrieve and set data. A client is said to be **interested in a dataset** when it wants to work with it. Interest in a certain dataset can be specified at boot time (in a configuration file) or at runtime. When a client is interested in a dataset, the server sends an entire copy of it (and the model

associated with it). Nevertheless, this model does not contain the business logic (since it can be executed just in the server). Clients store the data in a memory data structure referred as the **External Object Model** (EOM).

Once a client has received the complete dataset, posterior updates to that dataset are not sent as a whole dataset. The server just sends the differences (**deltas**) between the old dataset and the new one. It is important to understand the message-flow between the server and the clients. Each client has a **client ID** and the server contains a map between client IDs and interests. Deltas are just sent to the clients interested in them. In fact, this system can be thought as one following a **publisher-subscriber** pattern. The clients subscribe to datasets, and the server publishes changes through the network. There are mainly two options here: The server broadcast all the messages to its clients, and the clients filter the messages based upon its own interest, or the server maintains an interest table for all the clients. Quintiq uses the second approach. In the literature, the first approach is referred as a **broker-based event flooding** approach, and the second one as a **broker-based subscription flooding** approach. Both the TCEs and the Dispatchers act as brokers in the system.

There are 3 different kinds of clients which can be connected directly to the server:

**Fat clients** (FC) are full-fledged clients. They provide a GUI in order to allow users to access and change datasets (access to business logic, listing objects using filters…).

**Dispatchers** (D) have the same interface as the servers, although their internal data representations are different. Dispatchers, as fat clients, use also the EOM representation for the datasets. Their main purpose is to be used as **filters**. Filters can be specified in order to let the connected clients to just access a partial view of a dataset. If a FC connects to a D, and shows interest for a particular dataset, it will receive just the filtered part of the dataset. It is also possible to connect different dispatcher in sequence, allowing different layers of filtering. Moreover, dispatchers are also used for caching purposes (decrease the number of clients connected directly to the server).

Fat clients have a huge problem. They need to store the EOM in order to work. This means that the computer running the client must have enough memory. Moreover, there is a performance degradation if too many fat clients need to be connected to the server. The solution was to create a third kind of client, which in fact is a pair: the **Thin Client Engine** (TCE) and the **Thin Client** (TC). Multiple TCs (which are light weight Java processes, containing basically a GUI and communication facilities) can be connected to the same TCE (which

contains a single EOM). The TCE maintains a **session** for each TC connected to it. This session stores the state of the TCs. The combination of one TC and one TCE behaves like a FC. Nevertheless, the main difference is that while fat clients have a complete copy of the EOM, many thin clients can connect to the same TCE, sharing just one copy of the EOM:

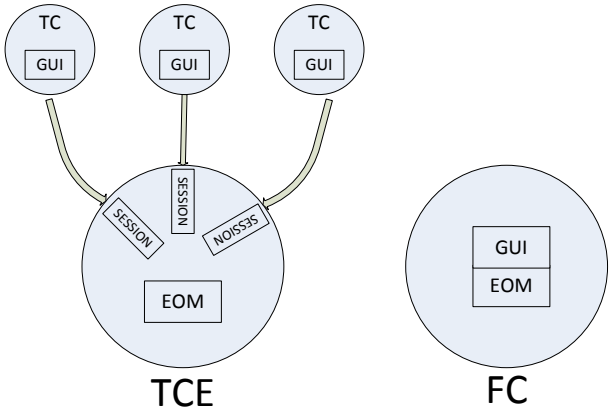



Figure 9 TCE and FC architecture

Whereas the server implements a one writer – multiple reader locking system, all the clients implement a Software Transactional Memory system, where it is possible to have writers and readers at the same time. For more information on this topic, see [5].

## 3.3 Publish-subscriber message flow

It is important to understand how connections are handled and messages are sent in order to fully-understand the system architecture and be able to propose alternatives and/or improvements.

Each intermediate node (dispatcher and thin client engine) and the server contain an **Interest Manager (IM).** The interest manager is responsible to store the relation between interests and node IDs. These relations are represented as tuples in the following figures. Each node is represented as: **Type {ID}**

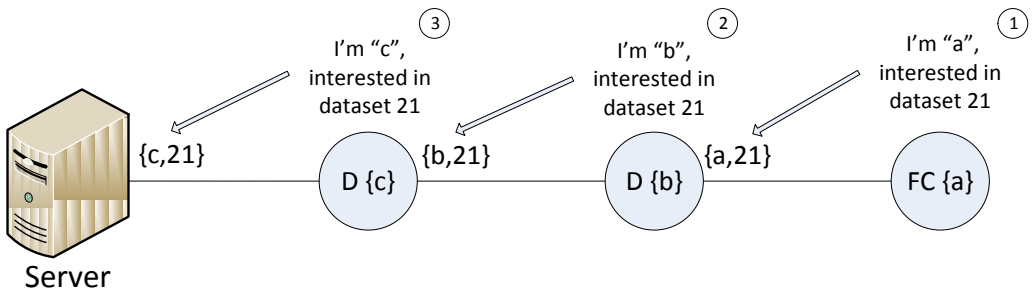



Figure 10 A node subscribes to a dataset

When a client needs to modify something in a dataset, it cannot do it directly. It has to do it using a transaction in the server. The transaction is sent from the client to the server, and then the server emits the response in the form of a delta:

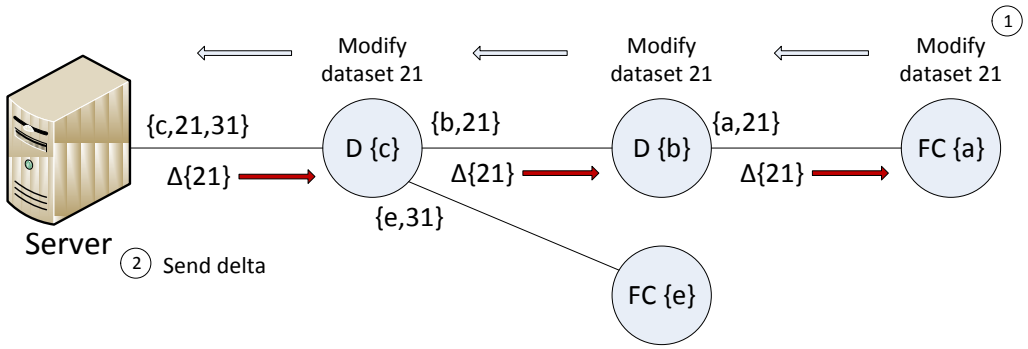



Figure 11 Writing transactions and responses from the server

When a client does an operation which ends up creating a transaction on the server, the client must wait until the transaction has finished to proceed its operations. Hence, that transaction is **synchronous**. Nevertheless, sometimes clients were interested in performing **asynchronous** operations. The **remote job server** (RJS) was introduced in the architecture to provide that service. The RJS contains an EOM representation of the dataset in which it is working on. Since the publish-subscriber system works with a dataset granularity (you cannot subscribe to something smaller than a dataset), a client that wants a response back from the RJS must subscribe to the **remote job dataset**.

## 3.4 Connecting to Quintiq from the exterior

A typical Quintiq system will be deployed in an enterprise network. Nevertheless, a typical scenario will be to have many TCs trying to connect from an external (or/and untrusted) network (workers needing to work from home or a cafeteria).

For this purpose, two different components were introduced into the network. The first one was a **multiplexer**, which lets multiple TCs connect through a firewall to the DMZ zone of the enterprise network. Nevertheless, this multiplexer has problems with proxies, and does not provide any mechanism for load-balance the network. A TC connecting to the network must provide the service name of the TCE they want to connect to. Hence, a second node was introduced: the **gateway**. The gateway lets different TC connect to different TCEs, but instead of using plain TCP connections, it uses HTTP URL Requests as a mechanism of TCP tunneling (in order to pass through the majority of the proxies used on the internet). Moreover, different load-balance methods are provided:

- A TC can specify different TCE service name to connect to. The first successful connection will be used. A connection could not be successful if the TCE does not exist, the gateway does not know it, the proxy drops packets…
- The gateway can have multiple TCE nodes assigned to the same service name. For each new connection, a TCE is mapped using a round-robin strategy. It is also possible to add weight to each TCE (useful if a TCE has more computing power than another one).

## 3.5 What can go wrong?

If none of the machines fail (and they usually do not fail), the system works as expected. Nevertheless, the room for improvement is very big:

- **Static configuration of the network**. The system architecture is hard coded. Every single node needs to know the node to connect to. This information must be provided at runtime. For example, TC clients need to know the IP and port of the TCE which contains the dataset they are interested in. This does not allow automatic scalability. Let us suppose that 100 TC clients are connected to the TCE, and that TCE is using the 100% of its CPU. If we want to connect a new TC, the system administrator will have to run a new TCE process, and connect the TC there. What is wrong with this approach?
    - The system is not well balanced. Since we need two TCE, it would be better to load balance the number of clients: 50 in one TCE and 51 in the other. But, the architecture is hard coded in the configuration file, and it is not possible to do this. Although the gateway provides some sort of load balancing, it is not enough for this scenario.
    - If the number of clients is reduced, we could run all the clients using just one TCE. Nevertheless, we cannot do this without restarting TCE and changing the configuration. Maybe to restart a TCE is not a

lengthy operation, but doing the same operation with a group of dispatcher would be a performance disaster.

However, load balancing is not the only problem:

- Let us suppose that one TC wants to access one specific dataset. It is the responsibility of the user to connect it to the proper TCE.
- A client wants to access a specific service, for example a map from a GIS provider. There are two different ways to do this. Hardcoding all the GIS providers into the configuration files, or contacting the server (which will have also hardcoded the list of all the GIS providers). It would be better to directly connect the service a node wants to use.

- **Points of failure everywhere**, and no way to recover from them. The network connection follows a tree structure. The server is the root, and if it fails the entire system goes down. If other nodes crash, the load-balance mechanisms would solve also this problem while the crashed node is manually restarted. In the next section we will talk about what do we need to achieve this.
  - The **order** on how do you start components should not matter. Right now, if the server does not exist, the other components fail to connect to it and have to be restarted (trying the connection later). The end-user should not need to be aware about how the service is implemented. For him, nodes of the system should be **plugable**.
  - The **configuration** of each machine is completely **manual**. The user has to go machine by machine applying the configuration files. A configuration service could be used. This service would allow configuration of all the elements in the network from virtually every node with sufficient privileges to configure that service.
  - Software updates and upgrades are big. Again, the user has to go machine by machine downloading the file and installing it. A **distributed installation** could be possible using P2P techniques. Some solutions have succeeded using bittorent to distribute the installation and configuration files among the clients, like the Twitter Murder project [6]. These solutions reduce the bandwidth used by the server and also improves the distribution time (see Figure 12).

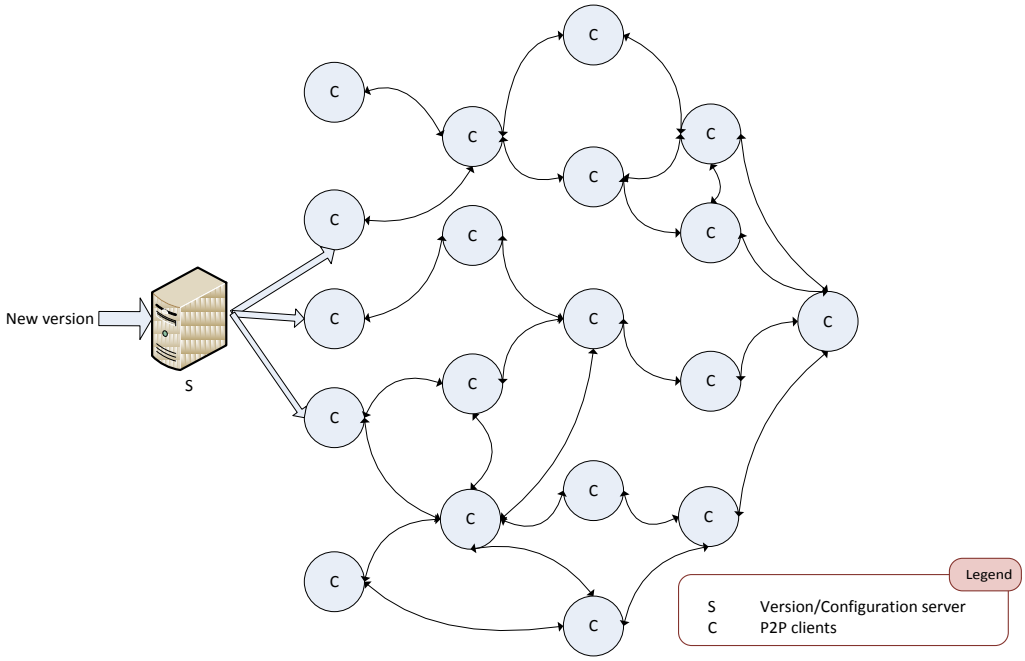



Figure 12 Distributing information in a P2P network

## 3.6 Running algorithms

One of the key points of the Quintiq software is its ability to optimize problems. Although the object model and variables are specified using Quill, it is important to realize that the different algorithms used are completely independent from Quill. The steps that must be followed to optimize a problem are the following:

- Choose the algorithm going to be used: Path Optimization Algorithm (POA), Mathematical Programming ( MP, which uses the external CPLEX library to solve it), Constraint Logic Programming (CLP, which uses Gecode) or Graph Algorithms.
- **Initialization:** Each algorithm has its own input **data**. Hence, it is needed to map data from the Quill model (the dataset) to the specific algorithm **constructs**.
- **Execution:** The algorithm engine is fed with the constructs and runs.
- **Resolution:** The engine produces an output. It is necessary to map the output constructs to object attributes in the business model.

The algorithm can run in two different modes: **synchronous** and **asynchronous.**

In the synchronous mode, a lock is kept on the dataset for the entire execution of the steps. Hence, if the algorithm takes a long time to execute, no user will be able to work on the dataset. One possible solution is to copy a dataset, and let the algorithm work on the copy:

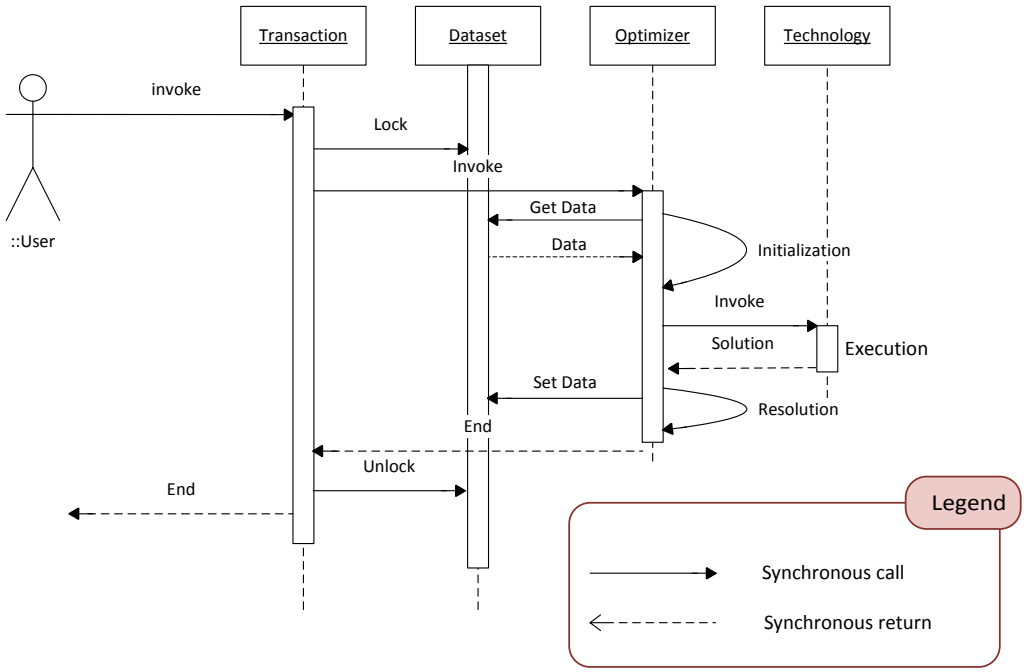



Figure 13 Running algorithms in synchronous mode

Using an asynchronous invocation, it is possible to reduce the amount of time the dataset is locked. Nevertheless, the user must take special care to any object instance that is being used as an argument of the algorithm, since they should survive the invocation. A UML sequence diagram for this execution mode can be seen in Figure 14.

In both modes, the real execution of the algorithm (Technology) is executed on the server, as part of a transaction or as a transaction itself. These executions are the perfect fit for our definition of "task". Figure 14 shows a possible interaction between the Quintiq system and the framework. Note that the communication flow is completely asynchronous.

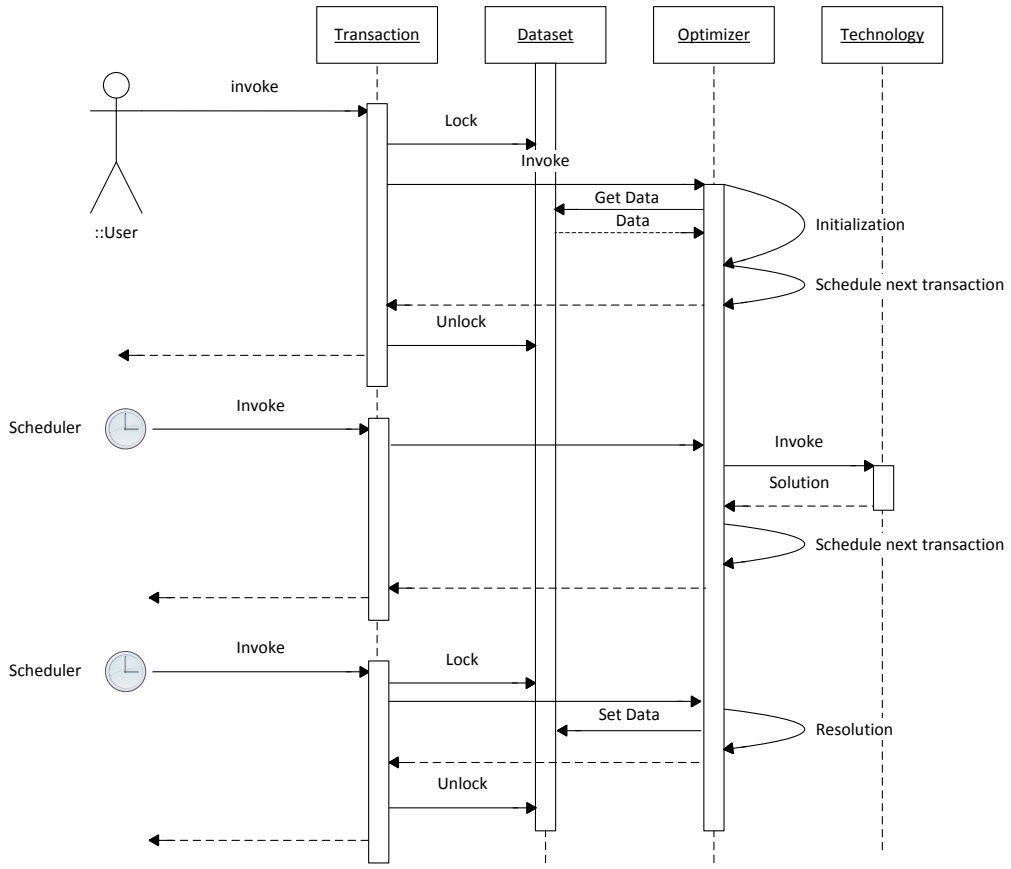



Figure 14 Running algorithms in asynchronous mode

# 4 Task distribution frameworks

In the last section we saw how Quintiq can run asynchronously different algorithms over the same dataset, minimizing the time the data is locked. Although the algorithms are executed in the same machine, there is nothing in the system that stops us to execute them wherever we want (see in Figure 15 how the asynchronous operations can be executed on different computers). In fact, limiting ourselves to execute them in the same machine is a clear bottleneck in the system: it can run efficiently up to N-1 concurrent algorithms (where N is the number of processors of the machine). Ideally, if a machine has N processors and there are M machines in the system, we would like to be able to run up to M*(N-1) concurrent algorithms.

Running algorithms distributedly is not just about increasing the capacity of the system: many times it makes sense to execute the same algorithm over the same dataset using different strategies and use the results from the strategy that finishes first (and hence, obtaining the solution faster). Other times, it makes sense to execute different instances of the same algorithm running in different parts of the data, and be able to create an optimal solution using this distributed information obtained from the different solutions. As we stated previously, we will consider that algorithms are independent of each other. This solution allows the user to use different strategies over the same dataset, but not to find solutions distributedly. Finding distributed solutions is highly coupled with the algorithm used and it is not our purpose to find a general solution for this.

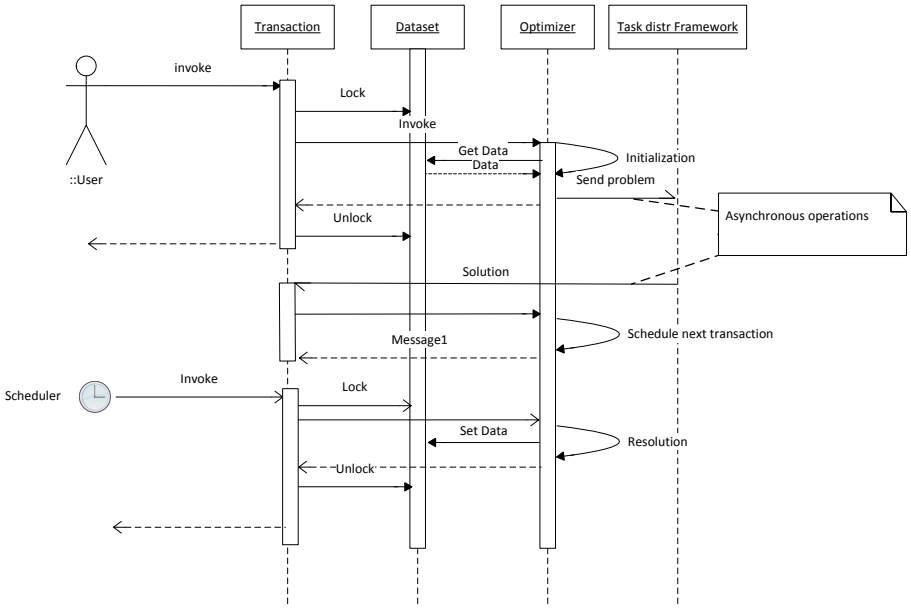



**Figure 15 Running algorithms using the task distribution framework**

Although a typical dataset size goes from a couple of hundreds of MBs to a few GBs, the algorithms will act over a set of objects (or more specifically, over a small set of object attributes) of those datasets. The size of the data fed to the algorithm engine is supposed to be small enough to be sent over the network to a different node. For us, small enough will mean that:

$$\frac{Transmission\ time}{Algorithm\ execution\ time} \ll 1$$

Nevertheless, here we are treating just one option. For now we have just thought about performing the execution part (see section 3.6) in a distributed fashion. We might go further, and execute the three steps (initialization, execution and resolution) in a distributed manner. In the Quintiq architecture exists one component capable of doing that: The remote job server. Nevertheless, it is not scalable: it maintains a full copy of the dataset in memory (which is synchronized with the one on the server) to avoid locking the dataset while executing time-intensive jobs. However, if the remote job server fails everything has to start again. Having a distributed dataset is something that will be briefly explained in section 4.2. For now, let us go back to the first idea of running the resolution part.

If we think carefully, it does not matter if what we want to execute remotely is an algorithm or the minesweeper game. What we want is to be able to execute different **tasks** on different computers and receive asynchronously the output of those tasks. There are many systems which provide the ability that we need: being able to execute tasks in remote nodes, we refer them as **task distribution frameworks**.

The job of a task distribution framework is not just moving and executing tasks from one place to another. Management, failure handling, security and many others must also be considered. In this section we are going to explore the most well-known systems. We are going to divide them into two different groups:

- **Distributed Resource Management systems**, also known as Batch Queinig Systems, or Job Schedulers
- **Data Parallel Execution Engines**

## 4.1 Distributed Resource Management (DRM) Systems

Regular desktop users usually work with their machine moving windows, editing documents, playing or visiting web sites. Nevertheless, a server computer tends to be used for two different purposes: running services or processing workloads. On one hand, services are always expected to be there. They do not usually move between hosts, and they are supposed to be long-running. On the other hand, we have workloads, which can be thought as performing calculations (**tasks**). Those tasks are usually done on demand, and it usually does not matter where are they calculated. This kind of work is referred as a **batch, offline** or **interactive** work in the literature, but we will refer them as **task** to keep the consistence of the document**.**

### *Managing the execution*

The main goal of DRM systems is to provide a high throughput computing environment [7], that is, providing a large amount of fault-tolerant computational power by utilizing all resources available to the system. Many times dedicated machines are used with the sole purpose of running tasks (like in Oracle Grid Engine and its open source version Open Grid Scheduler). Other times, frameworks provide opportunistic computing, utilizing resources whenever they are available (like BOINC). This means using desktop computers to perform expensive calculations when they are detected to be idle (no keyboard strokes, no mouse movement…). Other frameworks provide both facilities (Condor). Inside a

company with a 200 CPUs cluster and 500 desktop machines, the opportunistic computing could theoretically provide x2.5 computing performance than a company without using it (for example, at night desktop computers could be used with 100% of availability, since the users are sleeping). In fact, within Quintiq a third party tool is used to use those desktop machines to build the Quintiq system at night (Incredibuild). Incredibuild in its own uses a really interesting approach, virtualizing machines on demand. Nevertheless, it is no opensource and it is not possible for us to get information about the system.

Organizing a number of tasks on a set of machines is complex even if the number of tasks is equal to the number of machines. Not all the machines may be able to perform all the tasks. If we treat a task as an executable with a number of inputs and outputs, then that executable will need a minimum amount of RAM, processors, will be only able to run on a determined architecture or OS… If a task has a list of requirements, each worker must have also a list of characteristics in order to let the system match tasks to workers.

But, if the number of tasks is bigger than the number of machines, then we need to not only match tasks with workers, but also organize them in such a way that the utilization of the system is optimal (in terms of computing capacity of the entire system). Hence, a scheduler is needed, which controls when a task has to be submitted to a determined worker.

DRM systems try to be as general as possible. Hence, tasks are not delimited to a small amount of calculations (at Quintiq we are, since we just want to execute a limited kind of algorithms). They can be any kind of program, accepting any kind of input, with any kind of side effect (writing / reading files, communicating over the network, creating multiple threads or spawning new processes). It is important to protect the execution environment of the worker, using techniques like sandboxing or virtualization[7]. Not all the users will be able to execute tasks on the system, so authentication and authorization are also needed.

A list of services that a DRM system must (or could) provide is:

- Distributed job scheduler: allowing to schedule virtually unlimited amount of work to be performed when resources become available.
- Resource balance and run-time management
- Authentication and authorization
- Resource and job monitoring: Monitor submitted jobs and query which cluster nodes they are running on.

[7] This is a security problem beyond our knowledge, so in our implementation we will trust in the code executed by the task

- Fault tolerance

#### *Accessing and moving data*

If a task needs a particular file, there are mainly two options to provide it: Either the file is stored in a distributed file system, or the file is located in the local file system of the node that submitted the task, and has to be transmitted somehow to the worker.

We are going to analyze how three different well known frameworks which provide these facilities. It is not the aim of this thesis to do an extensive study and comparison of all the different approaches, but just to understand different approaches to the same problem.

### 4.1.1 Grid Engine

The Grid Engine software is a distributed resource management system developed by Sun Microsystems that was finally acquired by Oracle. In 2009, the project was made available under an open-source license and started to be maintained by the Open Grid Scheduler project. The Grid engine has a rich history and is a very mature software. In this section we will study its basic architecture and facilities that it provides. If the reader is interested, please refer to the official documentation [8]–[10], from which we have based this section.

#### *Architecture*

A Grid Engine cluster is composed of execution machines (workers), a master machine and zero or more shadow master machines, as shown in Figure 16.

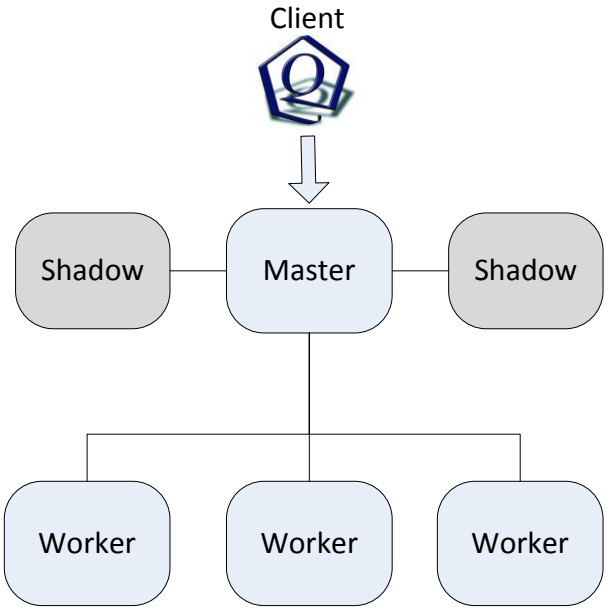



Figure 16 Grid Engine architecture

### *Fault-tolerance*

Every node has a spool parameter. **Spool** refers to the process of placing data in a temporary working area. This data can be used by the node that wrote it, or by different nodes. The spool parameter can choose between different spooling techniques:

- Local disk
- Database (local or remote)
- Network File System (NFS)

The master writes its configuration and the status of the running cluster using one of these systems. Shadow servers can only be used if they can access that data (the master uses a remote database or a NFS).

**Failure detection of the master:** The master generates a **heartbeat file**, and updates its value every 30 seconds. The shadow masters check for this file periodically, and if it is not updated, they consider the master as a failed node and try to become the new master.

**Leader election of the master:** There is also a **shadow master host file**, which is a list that contains the name of the primary master host followed by the names of the shadow master hosts. The order of the shadow master hosts is

important. If the master fails, the first shadow master on that list will become the new master. If the first shadow master fails, the second one will be elected as the master.

**Failure detection of the workers:** Each worker provides a number of slots (each of them capable of executing one task). Although the number of slots provided by a worker is not limited, it usually matches the number of CPUs of the host where the worker is executing. When a task submitted to a worker has completed, the worker notifies the master so that a new task can be scheduled on the now empty slot. Moreover, at a fixed interval each worker sends a report of its status to the master. Those reports are used by the master as heartbeats. If no reports are received in a certain amount of time, the master marks the worker as no longer available and removes it from the list of available task scheduling targets. Each worker uses its spool directory to store internal state. Hence, if a worker crashed it can try to reconstruct its previous state.

### *Scheduling*

There exists the concept of a **queue**. A queue is a logical abstraction that aggregates a set of task slots across one or more workers. Queues define where and how tasks are executed. On one hand it contains attributes specifying **task attributes**, for example:

- Whether if tasks can be migrated from one worker to another.
- Signal used to suspend a task
- Executable used to prepare the task execution environment.

On the other hand, it also specifies attributes related to policy, which provide priorities between tasks. The easiest way to understand queues it to think about them as particular nodes on the system, to which workers subscribe (see Figure 17). Nevertheless, in Grid Engine it is a little bit more complex (see Figure 18)

Hosts are grouped using **host groups.** A host group is just a list of hosts, and is expressed by the following notation: **@name**. The @allhosts group refers to all the hosts in the cluster (all the workers). There is also the concept of **resources**. Resources are abstract components that model properties of workers or the compute environment. There are three different types of resource, which can be associated to a particular queue, worker or group of workers:

- **Static:** Represented by a pair of two values, and are assigned by the administrator: Architecture of the host (AMD, Intel…), OS, amount of memory…
- **Dynamic:** Values monitored on each worker: Available memory, run time….
- **Consumable:** Model components that are available in fixed quantities and that are consumed by running tasks. When a task starts, the number of consumable resources is decreased by one. When the task ends, the number of consumable resources is increased by one. A classic example is a set of licenses of an executable. When workers are using all the licenses, no more workers can execute the program until a license is available.

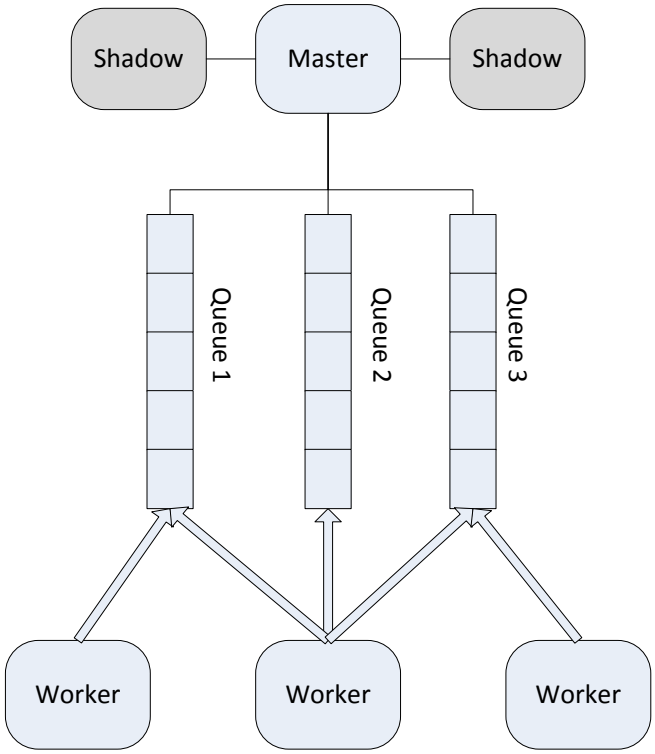



Figure 17 Queues as nodes in the system

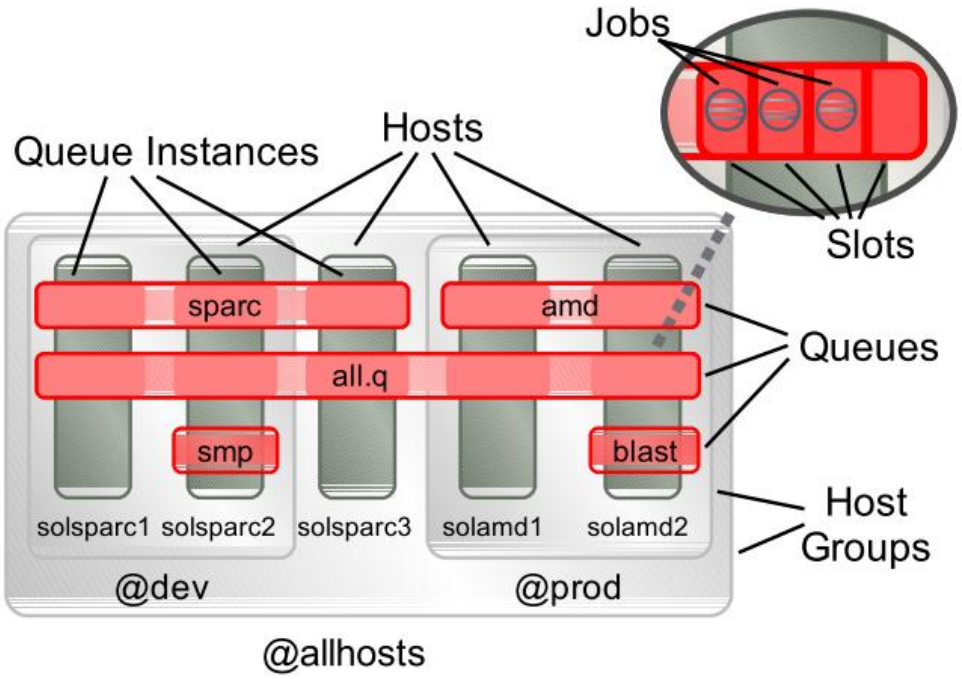



Figure 18 Queues in Grid engine (figure taken from [8])

After these explanations, we are able to explain all the steps followed to schedule a job:

1. A user submits a task to the master, which stores it in a pending list. User metadata and a list of resources required (requested) by the task is also stored.
2. A scheduling function may be triggered in 4 different ways:
    a. A task has been submitted by a user
    b. A notification is received from a worker saying that one or more tasks have finished executing.
    c. Periodically
    d. Triggered explicitly by an administrator
3. Task selection: Every task in the pending list is assigned a priority, and the entire list is sorted according to priority order.
4. Task scheduling: Assigns a task to a slot according to the resources requested by it and by the state of the queues with available slots. This process is divided into four steps:
    a. List of queue instances filtered according to **hard resource requests**, like operating system. If the request is not fulfilled, the task cannot be executed.
    b. The remaining list is sorted according to **soft resource requests**, like amount of virtual memory.
    c. The top queues from that list is sorted according to a static sequence number (defined by the administrator)

d. The top tier is again sorted according to the worker load, using a defined load formula (which can take into account the number of processes in the OS, number of CPU…)

Nevertheless, other algorithms could be used to schedule tasks, like a simple round-robin or a least-recently used worker.

#### *Data management*

The Grid engine does not manage user data by default. Hence, if a task needs user data, it must be accessible from the workers, normally using some kind of distributed file system or database. We will see that other systems provide a better way to handle data availability.

#### *Types of task*

In addition to batch tasks (independent tasks), Grid engine can also manage interactive tasks (logging into a worker machine), parallel tasks (typically using MPI environments) and array tasks ( tasks that are interrelated but independent of each other, that is, tasks do not communicate with each other). Nevertheless, to execute parallel tasks it is needed to configure a parallel environment. A parallel environment will be typically a group of workers with a master worker, which will handle all the parallel tasks (a subcluster).

### 4.1.2 Condor

Condor is a specialized opensource workload management system for compute-intensive jobs. It is the product of many years of research by the Center for High Throughput Computing in the Department of Computer Sciences at the University of Winsconsin-Madison. It is interesting to study because its architecture is completely different from the one found in Grid engine. It has three kinds of nodes:

- **Resource:** Act as workers. Each resource can handle one task. The resource provides a list of properties (similar to resources in the grid engine).
- **Agent:** User submits jobs to an agent. It is responsible for remembering jobs in persistent storage while finding resources willing to run them. Each task has a requirement list (again, similar to resources in the grid engine).
- **Matchmaker:** Responsible for introducing potentially compatible agents and resources

### Scheduling

The list of properties of resources and requirement of agents is referred as **classified advertisements** (ClassAds). ClassAds has its own language to specify properties, but it is out of scope. Its specification can be found in [11], [12]. Theses documents also describe extensively the condor architecture and functionality. The steps followed to schedule a task is the following one:

1. Resources and agents send its ClassAds to the matchmaker.
2. The matchmaker scans the known ClassAds and creates pairs that satisfy each other constraints and preferences
3. The matched agent and resource establish contact.
4. They negotiate (since their own ClassAds could have changed in the meantime). They can also negotiate further terms.
5. **Cooperate** to execute the job

As we can see, the steps are totally different from the ones in Grid Engine and deserve a deeper explanation.

Although resources and agents are independent processes, they are usually executed in the same host (being the host both a server and a client). We can see that more than one task can be submitted to the same agent. When a matched agent establishes connection with a matched resource, the agent creates a **shadow** process and the resource creates a **sandbox** process. Of course, an agent can have multiple shadows (multiple tasks matched with a resource) and a resource can have multiple sandboxes (one per tasks being executed). This process is depicted in Figure 19.

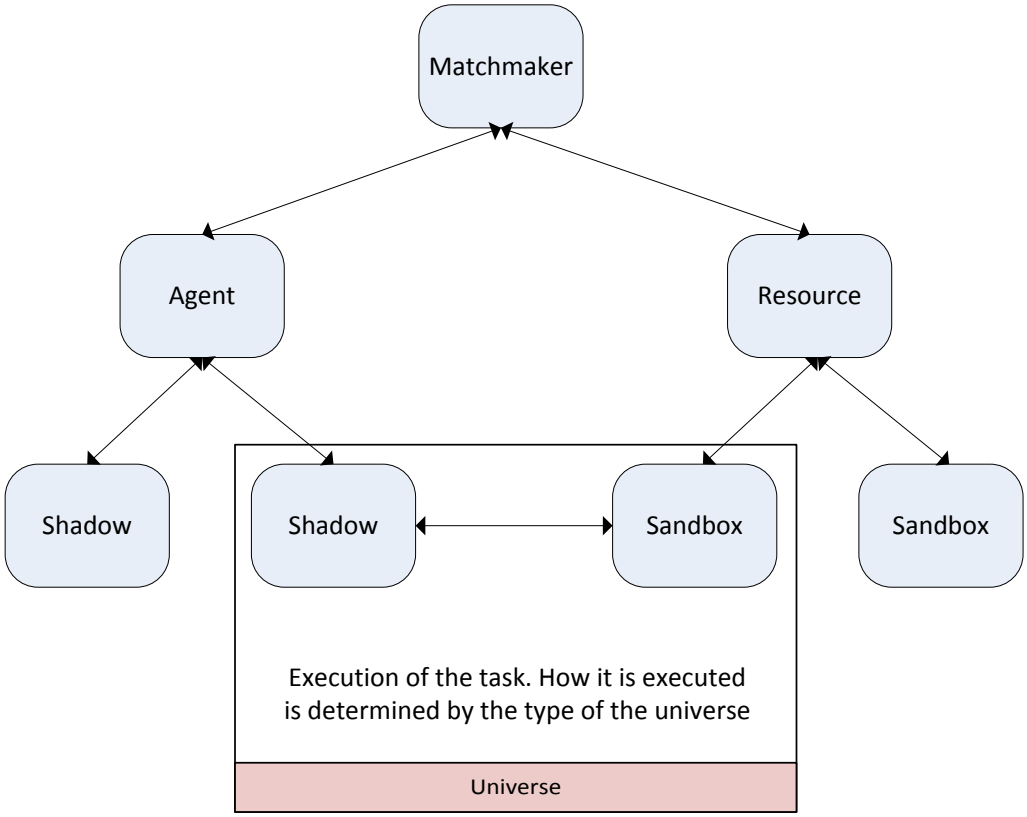



Figure 19 Condor architecture

### *Execution of tasks and universes*

The group formed by the shadow and the sandbox processes is referred as a **universe**. Each universe defines a runtime environment, which provide different abilities or services. For example, the **standard universe** provides migration and reliability, the Java universe allows users to run jobs written for the Java Virtual Machine and the parallel universe can run programs that require multiple machines to perform one task. We think that it is interesting to understand how some of these universes work.

#### Standard Universe

This universe provides **checkpointing** and **remote system calls.** When the task executable tries to do a system call (like for example opening a file), the sandbox redirects that system call to the shadow. Hence, if a task needs to open a file stored on the submitting machine (the one that runs the agent), the shadow will find the file and send the data to the sandbox. Nevertheless, to be able to do that,

it is needed to have access to the object files of the executable and relink the system libraries to the ones provided by condor (these system libraries have the same interface as the c/c++ runtime libraries, but with their own implementation. For instance, the fopen() function is implemented to transmit files from the shadow to the sandbox). Of course, with commercial applications it is not possible to do (we cannot access the object files), and hence it is not possible to use this universe.

Checkpointing allows the task to be migrated from one resource to another. The sandbox periodically creates snapshots of the state of the running task. If the machine fails, the last version of that image is copied to a new machine, and the task can be restarted from where it was left by the failed machine. Figure 20 , taken from [12], shows how does the standard universe works.

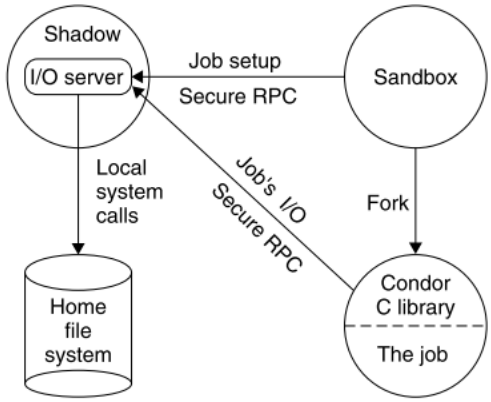



Figure 20 Standard universe

However, there are restrictions on the system calls a task can execute (multi-process tasks not allowed, no interprocess communication, brief network communication, timers not allowed…). For a complete list, please refer to the condor manual [11].

Checkpointing any kind of application is currently being developed in the Berkeley Lab Checkpoint/Restart project [13]. The focus is to provide checkpointable MPI libraries, a thing that could improve the fault-tolerance of it.

### Vanilla Universe

The Vanilla Universe is intended for programs that cannot be re-linked, or for shell scripts. Hence, no migration or remote system call is possible. Like in Grids Engine, it is the responsibility of the user to make sure that the Resource can access the files needed by the task (using a distributed File system). Nevertheless, if

it is not possible to provide such a system, Condor provides a **File Transfer Mechanism**. With this mechanism, both the **executable** and the needed files can be sent to the resource.

### *Flocking*

Condor provides the ability of building Grid-style computing environments that cross administrative boundaries. The technology that allows this is referred as flocking. A group of agents, resources and a matchmaker can be thought as a condor **pool,** where the matchmaker is the central point of coordination. A pool is normally controlled by one organization, that will configure the matchmaker to satisfy its own needs.

Many times two or more organizations will want to share its computing power. However, each organization will want to do it in a controlled way, keeping control of their own pool of computers. For this reason connecting all the agents and resources to the same matchmaker is not a solution.

Previous versions of condor solved the problem introducing the idea of **gateway flocking.** The main idea was to create connections only between matchmakers (federation of matchmakers). Let us suppose that matchmakers A and B are connected. The main idea is that if an agent sends a request to A, and it cannot match it to a resource, it will forward the request to B. Although this method improved the scalability of the system and it simplified the administration of the system, the main problem was that it was not possible to share subgroups of machines. Gateway flocking only allowed sharing at the organization level. A different strategy was proposed: **direct flocking.** With it, agents may simply report themselves to multiple matchmakers. However, the number of total open connections between nodes grows faster, making the system less scalable.

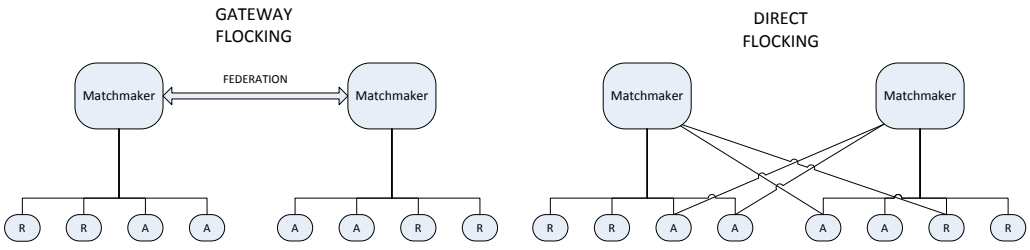



Figure 21 Flocking types in Condor

### 4.1.3 BOINC

The BOINC project helps researchers to use computing power of home PCs.

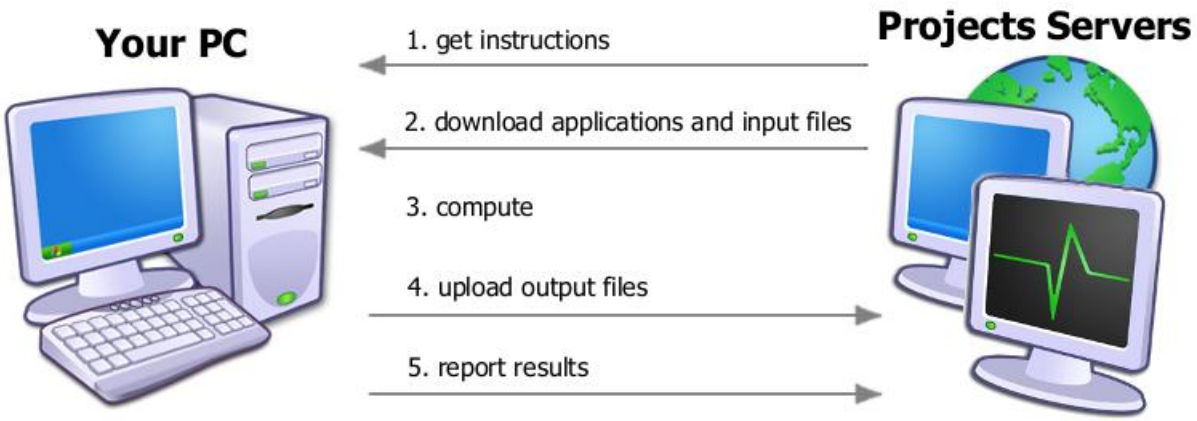



Figure 22 BOINC

BOINC is designed to work in non-trusted environments. Their idea is to have a BIG problem, divide it in smaller tasks, and send each task to a different desktop user. Since it is not possible to trust 100% those users, the same task is sent to multiple users. When some results are received, they are compared, and if they match the task output is considered valid.

This model does not match at all what we want to achieve, so here concludes the section. We wanted to at least mention the project, since it is helping a lot of researchers around the world.

## 4.2 Data Parallel Execution Engines

New distributed programming models and frameworks arise from the need of providing transparencies to the programmer. New abstractions allow programmers to express simple computations without having to deal with parallelization, fault tolerance, data distributions or load balancing[8]. Distributed Resource Management systems try to do it as general as possible for any kind of application. Nevertheless, there are problems that are not easily solved using those systems. For many applications, though, there is a very simple way to achieve scalable performance: exploiting data parallelism.

If we have an enormous amount of unstructured data that we want to analyze, like web pages or logs, which is inherently independent one from another, it seems easy to divide that data into smaller pieces and analyze them in parallel. If one of the processes executing over a small dataset crashes, fault tolerance can be provided by re-executing that process on a different machine. If the pieces were small enough, the re-execution overhead is negligible.

Different engines provide different computation patterns. For example, Dryad [14] combines computational "vertices" with communication "channels" to form a dataflow graph. Due to a lack of time, in this section we are only going to talk about another computational pattern: **Map Reduce.** Nevertheless, we believe it is enough to understand the basic ideas of Data Parallel Execution Engines.

Map Reduce is a programming model (which has different high efficient implementations, like Google Map Reduce or Hadoop) that borrows two widely used functions to process lists from functional programming: **map** and **reduce**, but extending their meaning.

The idea is to think about the unstructured data as a huge list of elements. Those elements can be web pages, lines from different files, images… That data is stored in a special Distributed Files System (which also has two different implementations, Google File System and Hadoop File System), which provides fault tolerance replicating data over at least 3 different hosts in the system.

Each map function is fed with a small subset of those elements. From the programmer's point of view, the map function takes an individual element of the subset, and process it. The input elements are in the form of tuples: {key, value}. For instance, in an application counting the number of words in a text file, the key could be a document file name and the value a line of that file. The output of the

[8] Simplicity is derived from not having to deal with this complex problems.

map function is a collection of tuples with the form {key, values}. That is, a map function could return 0, 1 or multiple outputs from a single input.

The map function executes in parallel on different nodes of the cluster. Since data is very big, the engine will run the function keeping in mind **data locality**: It will try to execute the function in the node that actually contains the subset of data to analyze. If that node is already performing another function, it will try to select a neighbor machine (in the same rack), since the transmission of the data will be faster. Each map function writes its output to a file in the DFS.

When all the map functions have finished, the engine takes all the output files and merge all the tuples with the same key. Producing a tuple in the form of {key, value1 [value2, value3,…]}. Each tuple is fed to a reduce function. The reduce function takes such a tuple and outputs a tuple in the form of {key, value}. Hence, the reduce function aggregates all the values with the same key. All the steps are shown in the following figure, taken from [15]:

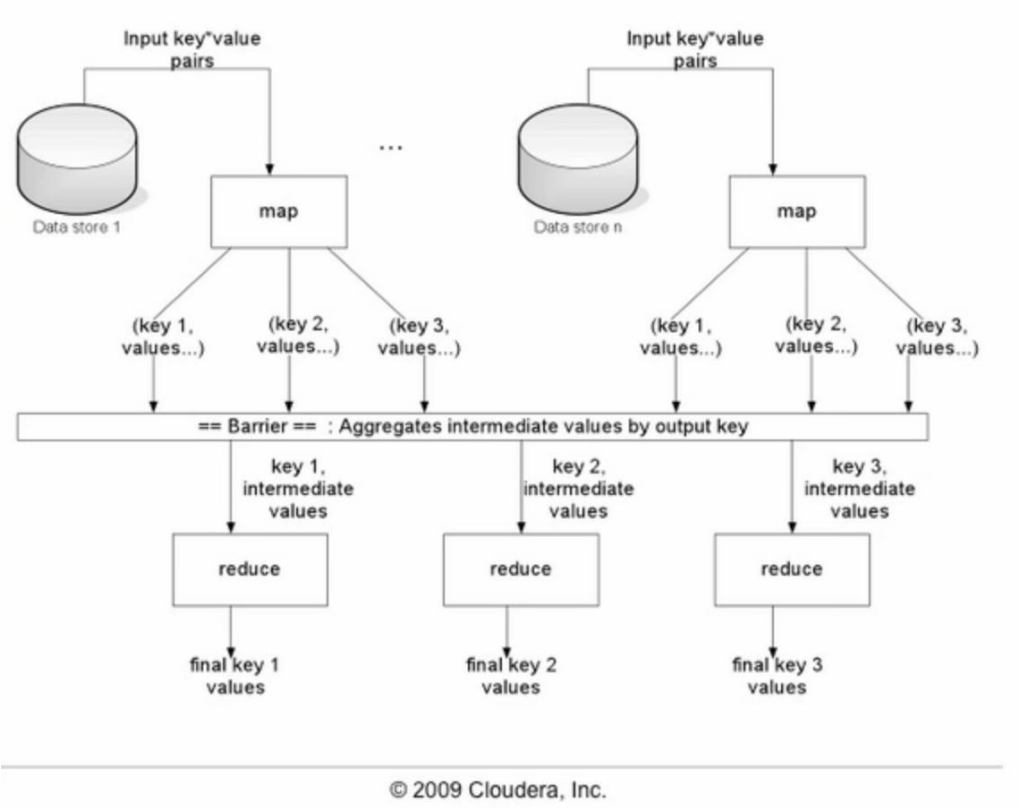



Figure 23 Map-Reduce process

An immediate application of MapReduce in the Quintiq system could be to improve the speed of filtering elements in a list. It would be sufficient to use the

map function, and the filtering could be executed in different machines. Nevertheless, in order to do it, it would be needed to change the entire Quintiq architecture, since right now each TCE, FC or dispatcher contains its own copy of the dataset. In a MapReduce World, all those elements would form part of the Distributed File System, removing the need of a central server. However, how the transactions are handled by that Distributed File System is something completely out of the scope, and who knows how the performance would be.

Nevertheless, map reduce is also applicable in a single machine, so the filtering could be implemented using map reduce on each fat client or FCE in order to use more than one thread.

**Dryad,** developed by Microsoft, tried to go even further, extending the concepts of Map Reduce and letting the programmer think about a task as small subtasks (vertices) which communicate following an arbitrary Directed Acyclic Graph. Nonetheless it is not longer developed and supported, so although it could be interesting to study, it should not be used in a production environment.

To finalize, it is worth to mention that many high level languages have been implemented over the map-reduce programming model. Two of the most important are Sawzall (Google) and Pig Latin (Hadoop), and provide a grammar similar to the one found in SQL statements.

## 4.3 Conclusion

Changing the entire architecture of the Quintiq software and modifying the dataset structure to fit the map reduce model is something that would take years of development, and many parts of the software should be written almost from scratch. From a Master's thesis point of view, it makes no sense to try to do this, since the scope would be too broad. Although these engines and programming models are very appealing, the purpose was to execute simple tasks distributedly, so the Distributed Resource Management Systems are our best bet:

- The algorithm we want to run (CPLEX) can be exported as an executable on its own. Hence, we can use it in both Grid Engine and Condor.
- We will just have (for the moment) one user (the Quintiq Server).
    - In the condor framework the server will contact with a single agent, which will handle all the tasks. The Condor framework provides **problem solvers**, which are high-level structures that use agents as the basis. The Master-Worker problem solver could be used.
    - The Grid Engine system matches perfectly. The Quintiq Server will have to send requests to the master node.

Even though these systems would solve our problem, they are overkilling. We do not need so much complexity: Priority management, execution of **any** kind of program, complex administration of the system… The configuration of the architecture is also expensive and complex. Moreover, **we do not want to finish the thesis so early**[9], so we will try to program our own task distribution framework, which should be simpler to use than the others.

We have also seen a variety of fault tolerance methods: Grid engine uses shadow servers (but need a distributed file system). Condor provided from version 6 a high availability and replication module for the matchmaker [16], and agents provide fault tolerance persisting the work in hard disk. The Condor one relies just on message passing, so it is more general than the one on the grid engine. Nevertheless, replication mechanisms are always very difficult to understand. In a way or another they always use a sort of **atomic broadcast** and **leader election** algorithms. These problems should not be solved every time that we want to implement a distributed system.

In fact, our problem could be easily solved if we had:

- An easy, consistent and reliable way to send messages between nodes: Relying just in message passing techniques is more general, like Condor does. TCP sockets have too many corner-cases.
- Abstractions providing coordination between nodes (consistent state between master nodes). We do not want to reinvent the "fault-tolerance" wheel every time.
- A framework to program the nodes as reactive entities (actors). Why? Read the next section.

In the next section we are going to explain how can we get these facilities. After studying many alternatives (see section 5.2), we have decided to use **Zookeeper** as our "coordination provider", and **ZeroMQ** as our Message Passing and threading library. Before explaining what are Zookeeper and ZeroMQ, we find fundamental to explain all the decisions that a programmer must make in order to program a distributed application.

[9] Yes, that is an important conclusion for us. We want to learn as much as possible from this thesis.

# 5 How to implement a distributed system

In this section we are going to talk how the actor model can help us to build distributed applications. We will explain how it deals with concurrency, asynchronous I/O and how can be implemented in C++ using the ZeroMQ library.

## 5.1 Technological barriers

Implementing a parallel or distributed system is not an easy task. This section tries to explain all the decisions we have had to make in order to progress in the development of our implementation. The text is highly influenced by the whitepaper Multithreaded Magic[17], written by the developers of the ZeroMQ library.

Nowadays regular desktop machines have from two to eight cores, and many times each of them can handle more than one thread of execution. Programming applications to use these facilities is not new. The typical approach is to use multi-threaded applications. Nevertheless a programmer needs to deal with concurrency problems, and many times solving them is painful and time-consuming. As shown in a technical article from Microsoft [18], even an experienced programmer will have to face with race conditions, data corruption, failed synchronization, starvation (maybe due to the usage of too many threads accessing a locked data piece)… Programming using synchronization points, critical sections and locks *does not feel natural*. Nevertheless, since it is the common solution for multithreaded programs, let us suppose that we want to implement our framework using it.

We have to choose in which language we want to program it. Here we face a new problem:

- The well-known languages C and C++, do not offer any support for concurrency. For them, a computer is a single threaded process. Multiple third party libraries add thread support to these languages, but they do not use a standard API and many times they are not platform-independent (the thread mechanism is provided by each OS). Although the new version of C++ (C++0x) solves this problem providing a standard library that deals with threads, not all compilers are compatible with it yet. There exists also the Boost library, which is a compendium of c++ libraries and contains a cross-platform implementation of threads.
- Other languages, like Java, Python, Ruby or C# do support concurrency but in a brute-force manner (in the same brute force-manner than the new C++0x

standard). They provide thread creation, waiting or detaching them and synchronization abilities, from simple locks, monitors or semaphores to more powerful techniques to exploit hardware instructions (like Compare-and-swap).

The problem with this traditional approach is that the multithreaded code is expensive and difficult to write, and even more difficult to maintain. There are just too many things than can go wrong. Moreover, this approach is not scalable in terms of number of threads. Even if the code is multithreaded, it could happen that it does not really benefit from multiple cores, since many times threads block each other continuously. A way to solve this problem is to use lock-free techniques to let the threads access shared data concurrently without blocking, but these techniques are extremely difficult to use, and their implementation depends on the memory model of the processor in which the algorithm is running[10]. However, the most important reason to not to use threads is that although they map perfectly to the computer world, they are just strange entities in the human way of thinking (at least, in the author of this thesis way of thinking). In our nowadays society, human interaction matches the message passing technique. Humans talk, touch, feel… All the relations between humans and between human-environment exist thanks to the 5 (or 6?) senses that we possess. We receive information from our senses, and react upon them. The only way that we can communicate with our environment is through our senses[11] (Figure 24). This is the way we are made and understand the world. And, in fact, this vision of the world matches perfectly with the **actor model** of distributed systems (Figure 25), where actors can communicate among them regardless where actors are located.

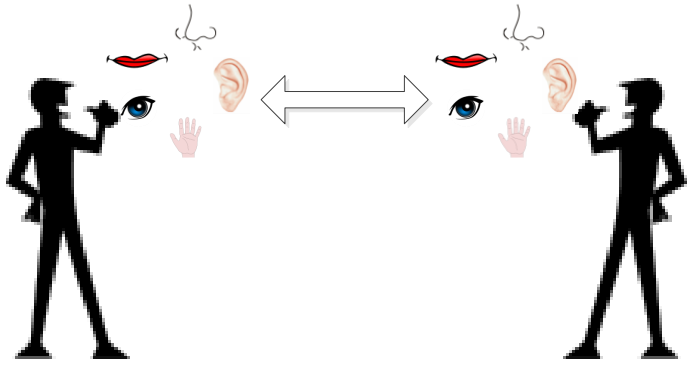


Figure 24 Human interaction

[10] If the reader is interested in lock-free algorithm, we recommend [19] as an introduction.

[11] At least for now we cannot share our brains, right?

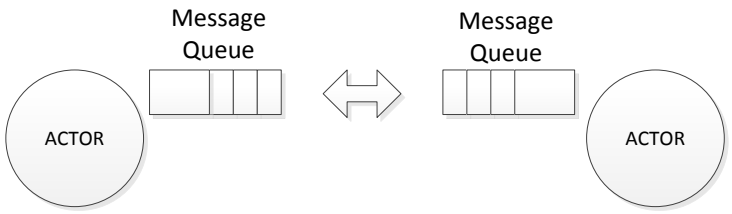



Figure 25 Actor interaction

- Other languages focus more on concurrency issues, and they treat it as a first class component in the language. Those languages, like Erlang[12], Scala (Akka framework) or Clojure, implement the "actor model". Using the actor model to tackle concurrency has shown to be one of the two best ways to do it (the other one being Software Transactional Memory, which relies on shared memory and optimistic locking). Of course, actor implementations use multithreaded techniques (synchronization, locks, lock-free data structures), but they provide a new layer of abstraction, in such a way that concurrent applications can be easy to develop and to argue about.
  The actors are really cheap to create and destruct, and it is possible to have thousands of concurrent actors in one processor. Nevertheless, those languages are not quite popular yet (although they have gained popularity in the few past years).

Nevertheless, not many programmers are used to this programming model. Let us suppose that our programmer does not want to learn how to think in terms of actors and goes back to our multithreaded solution. If a computer was completely synchronous, we could consider this solution good enough. We will see that it is not. Unfortunately, computers have to deal with multiple asynchronous events: signals and IO (input-output) among the most important. With IO we refer to every different way the CPU communicates (hardware in the same machine, communication between threads and processes or with other machines, peripherals… ). In a distributed system we are mainly interested in the communication between processes and other machines (communication between threads is handled using locks or shared memory), and these operations are normally **blocking.** Although send operation can be non-blocking, if a thread wants to receive something, it will block until something is received (that is, someone has sent something over the channel). Hence, the thread is wasting time waiting for a response (unless waiting is considered valuable).

The next step is to allow threads to perform asynchronous operations. The solution is that if a given operation would block the thread, it returns immediately, so the thread can try again later (and perform some operations between the tries).

[12] Nowadays Erlang is the best implementation of actors.

Using this approach, it is the thread the one that has to constantly poll the channel to see if it can perform the operation. A more advanced solution uses OS facilities. It is the OS the one that will poll the channels, being possible to let it poll more than one (and probably in a more efficient way). Nevertheless, each OS provides its own (and incompatible with other OSes) facility: /dev/poll, kqueue, select, epoll…

There exist libraries, like libevents or libev, which support all of these systems, exposing its own event API, and using the best system depending on the OS it is used. However, controlling the application flow tends to be difficult. The next usual step would be to create an event-driven framework, which would allow the users to subscribe handlers (functions) to a determined event. The common approach to implement them is using the Reactor pattern, as seen for example in [20]. Please note that the reactor patterns would provide a "sort-of" actor behavior if we treat events as messages than can be received. However, a very important point is missing. Handlers should be able to have state (that is, remember things between different calls). We will explore this option later on.

Again, there is no standard API for event-driven framework common to all programming languages. Nevertheless, let us suppose that we decide to use one of these frameworks.

Are threads and event-driven frameworks enough to create distributed systems? The answer is, again, no. We have to handle the communication between machines. And this opens an entire new paradigm. The common approach is to use TPC/IP, since it happens to be almost universal. But using plain TCP sockets is hard (We do not want to reinvent the wheel and program all the corner cases in a, for example, normal TCP/IP connection). Additionally, what would happen if we want to change the communication layer? Do we have to implement everything again? Or should we create an abstraction library which handles different communication protocols?

There are multiple implementations of message-passing libraries, and again, their API (and many times wire protocol) are not compatible between programming languages or OSes. Normally each event-driven framework provides also facilities to do asynchronous messaging.

As we have seen, choosing a programming language usually limits us which libraries can we select, and with each library we restrict ourselves more and more. Moreover, It may seem that having chosen programming language, threading library, event-driven framework and message library would be enough. And the answer is, as you may know, that it is not enough. If we recall Figure 4 and Figure 6, we can see that although the architecture is different (it is not the same to

program a worker in a thread than in a different machine), conceptually it is exactly the same. We have two nodes that must communicate between each other. The programmer should not worry (or at least should not be very concerned) about where and how are the nodes deployed. The programmer should be able to treat those nodes as actors. Although there exist actor frameworks, they are again incompatible between languages and each one uses its own no compatible message-passing implementations. So, are we limited to use Erlang or Scala?

One of the main requirements for our task-distribution framework was that it should be programmed in C++, but it should be also easy to port to other languages. Do we have any alternative? One of the goals of this section is to provide an affirmative answer to this question.

## 5.2 Thinking of actors

As we saw in section 4.2, there are frameworks like map-reduce that provide high-level abstractions of distributed systems, allowing the users to perform complex tasks without having to worry about scalability, reliability and fault-tolerance. Nevertheless, someone has to program the map reduce engine. And that someone will face all the problems explained above! Now we face the problem of providing the right tools to implement that engine (from our point of view, our task distribution engine).

There are many middleware libraries that provide services like group messaging, quorum creation, failure detection, state machine replication[13]… Just to name a few: Corosync, Spark, ZeroC, Spread or the Microsoft Asynchronous Agents library. Many of them have support for the mainstream programming languages. Even though they provide solutions for many problems that can arise in a distributed environment; we are most interested in providing tools to implement these kind of middlewares[14]. This is why we will not consider them anymore.

As we have said many times, actors can be the solution. We have talked a lot about actors, and we have given a general definition of them. Now it is time to give a definition that actually shows how actors are implemented. We think that a good definition is given in **[21]** (Chapter6):

> "Actors are **threadless**, **stackless** units of execution that process messages (events) serially […].Actors process incoming messages and encapsulate their state[…]. They always process messages **asynchronously**

[13] All the typical problems that must be solved in a distributed environment

[14] We think that in order to understand how useful the frameworks are, first we have to face the problems their developers suffered.

and may not process messages in the order that they were delivered […]. Messages are delivered to an Actor's **mailbox** that the Actor can currently process. If the actor cannot process any messages currently in the mailbox, the **Actor is suspended** until the state of the mailbox changes. The Actor will process **one message at a time** […]. Messages can show up in the Actor's mailbox while the Actor is processing a message."

The best known actor implementation is Erlang, which actually implements the definition given above. It is important to understand what does the definition mean: We will discover that **we cannot implement such an Actor in a language like C++**. However, we'll be able to implement something similar, sometimes referred as **object-oriented actors**.

### 5.2.1 Erlang actors

All the C++ implementations use the stack to store variables (the ones declared as automatic variables) and to pass arguments to functions. Each time that a function is called, more memory is needed from the stack. This implies that for example, recursive functions can potentially produce a stack overflow. The classical example is the recursive factorial function:

```
int factorial( int n )
{
        if (n==0 ) return 1;
        return n* factorial(n-1);
}
```

It is possible to avoid the increase of the stack using tail-recursion. The following re-implementation of the factorial function uses it:

```
int factorial( int n,int counter )
{
        if (n==0 ) return counter;
        return factorial(n-1,(n-1)*counter);
}
```

Now, if we want to calculate the factorial of N, we have to call factorial(N, 1). See that this function will return the value of factorial(N-1,N-1). Instead of increasing the stack, the compiler can delete the current stack (we do not need it any more) and replace it with the new one for the factorial (N-1, N-1). Nevertheless, not all the compilers are able to do this kind of optimization (called Tail Call Optimization), and if they can they will not always optimize it, so we cannot rely on this optimization in order to implement actors.

We are sure that the question the reader is thinking on is: Tail recursion seems a good trick, but what it has to do with actors? The answer is that it is one of the core foundation in the Erlang actor implementation. Actors can be seen as a chain of functions, where each function calls another one. Since everything is a function and **variables are immutable**, the only way to store state is through function arguments (it makes no sense to have a global state that cannot change). Since all the state is contained within the function[15], it is very easy to make a snapshot of the actor, pause it and execute a different actor. The paused actor can be later on executed on the same or on a different thread (so they are threadless).

The important part is "**in a different thread"**. Actors can resume their execution on different threads without any kind of synchronization from the programmer's perspective, because there is no shared data. This is a really powerful property, which lets the system scale with the number of CPUs automatically and without any kind of source change. As more CPUs your computer has, as faster your program will run.

Moreover, spawning new actors and deleting old ones is really cheap, given that there is not any kind of contention nor context switch involved. It is so cheap that a common practice is to create an actor to contain (wrap) shared data between other actors. In order to access the data, the actors will have to send messages to the actor that wraps the data. Since actors can only process 1 message at a time, the access to the data is automatically sequentialized, no locks involved!

Nevertheless, the Erlang programming model is too strict. Among others:

- We need to access data in a very efficient way, and then the "wrapper actor" trick is too inefficient[16].
- We have to interface with other libraries which are not compatible with Erlang[17]. Although it is possible to interface with C libraries, it is far from easy.
- It can be difficult to model complex behaviors, since it is difficult to follow the path of execution (there is no any central loop that controls it). However this problem also appears in asynchronous programming. In the following sections we will see that we can handle this problem using state machines.
- Erlang uses its own message serialization protocol, which is not extensible to other languages.

[15] Erlang provides ways to store data outside the "actor", like the Mnesia Database, we do not consider them in our analysis.

[16] Again, Erlang provides some facilities to solve this problem

[17] Although the Erlang standard library is quite complete, there is not a good third party ecosystem around it.

Although these arguments may seem week, the following one is the most important: It is difficult to find experienced Erlang programmers, and even more difficult to find companies that are willing to change their entire code base from their particular programming language to Erlang.

What we want is to create some of the best things that Erlang provides and adapt them to C++. Moreover, the implementation should be able to be easily ported to other programming languages, and different actors running on different programming languages should be able to communicate with each other without any extra code.

### 5.2.2 ZeroMQ

The best way to describe ZeroMQ is to copy the explanation found in ØMQ - The Guide [22]:

"ØMQ (ZeroMQ, 0MQ, zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry whole messages across **various transports like in-process, inter-process, TCP, and multicast**. You can connect sockets N-to-N with patterns like fanout, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its **asynchronous I/O** model gives you **scalable multicore applications**, built as asynchronous message-processing tasks. It has a score of language APIs and **runs on most operating systems**. ØMQ is from iMatix and is LGPL open source."

Asynchronous communication is achieved by performing all the send and receive operations in 0, 1 or N separate thread. Those threads are completely transparent for the ZeroMQ user, and are handled by the **context** object. The context object is the only object thread-safe in ZeroMQ, in the sense that it can be used concurrently from multiple threads.

From the context it is possible to create different kind of **sockets**, which can be tight to different kind of transports: TCP, InfiniBand, PGM or interprocess communication. Each socket has its own mailbox, from where it can read messages. All the transport mechanisms except the interprocess (which does not need any extra thread) one use one or more threads. In these threads different OS sockets are opened and polled automatically. Sockets are not thread-safe in the sense that the only thread that can use a socket is the one that created it. If we want to change the "ownership" of the socket, the transference has to be made using a full memory barrier in order to make sure that the creator thread does not access the socket after the new owner thread has used it.

#### 5.2.2.1 *ZeroMQ sockets*

Although we are not going to explain how ZeroMQ is implemented, it is important to have an abstract idea of how sockets work. If the reader wants to deepen in ZeroMQ, he can read some of the white papers at http://www.ZeroMQ.org/area:whitepapers.

Each socket contains two queues, an inbound and an outbound one. Each queue is implemented using non-blocking algorithms. Since no OS synchronization facilities are used, putting and getting messages from the queues is as fast as possible, without any thread contention. Hence, ZeroMQ sockets provide us with a big chunk of the actor implementation for free!

Moreover, there are different socket types, each one with its own behavior and a particular **incoming** and **outcoming routing strategy** (each socket can be connected to more than one socket). These different behaviors allow us to easily implement different node topologies. The most important routing strategies are depicted in Figure 26. There are two other routing strategies: last-peer (sends the message to the peer that sent us the last message we have received), used in the request-reply pattern, and a special routing strategy for the ROUTER socket. In-process communication does not use any thread (it uses lock-free techniques over shared data structures). The other transports will normally use 1 thread, although applications with a high throughput requirement can use more than one.

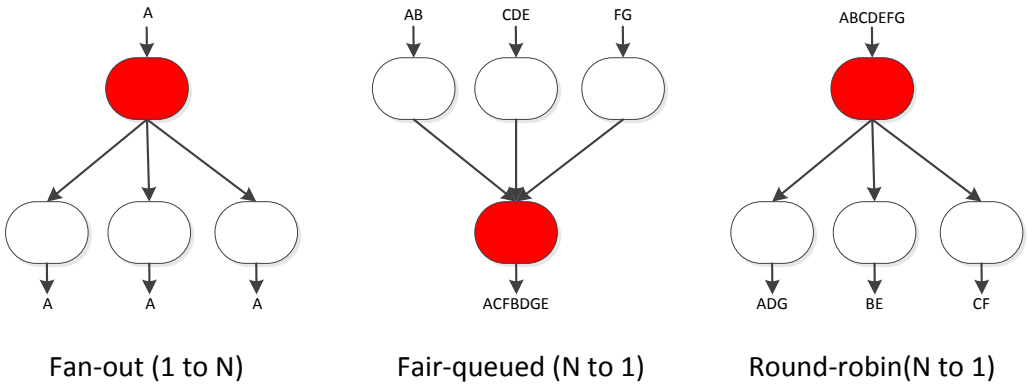



Figure 26 0MQ socket's routing strategy

For instance, the request-reply pattern emulates the typical architecture of server-client used with TCP. Nevertheless, 0MQ sockets are more powerful: Messages **are not streams but blocks of data**. When the programmer sends x bytes in a message, the receiver will receive exactly those x bytes when it executes the receive operation.

The order of connection is not important. We can connect first the client socket, and bind later on the server socket. Everything will work fine! There is no need to restart connections. All is handled automatically by ZeroMQ.

This *magic* behavior lets the programmer think about the architecture he wants to build, and not how to build it. For example, let us suppose that we want to send tasks from a central server to processes following a round-robin assignation. The only thing that we have to do is create the connections showed in Figure 27.

Programming the Server is as easy as sending messages through the PUSH socket. Programming the Worker is as easy as receiving messages, processing them ands executing the task described in the message. PUSH sockets have a round-robin outgoing policy. We could extend the architecture and have different servers sending messages to the same pool of workers. The great thing is that we do not have to change any code. Just start a new server process and connect it to the pool of workers! PULL sockets have a fair-queued incoming policy, so there will be no starvation (see Figure 28).

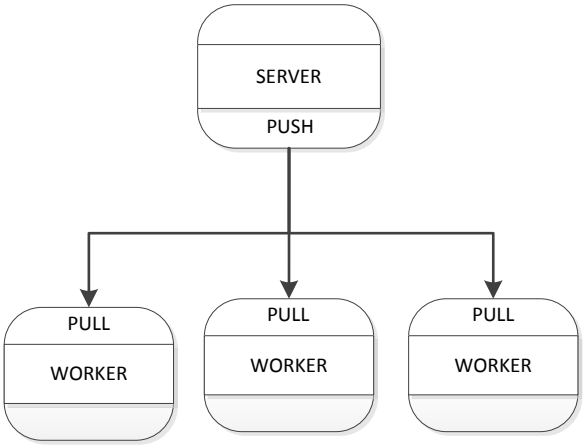



**Figure 27 Easy task distribution using 0MQ**

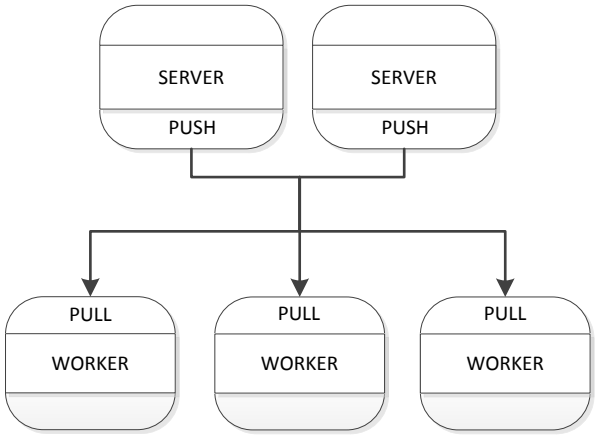



Figure 28 Adding a new server with any additional configuration

This dynamicity is perfect for prototyping, but also for production. Even better, working with 0MQ does not limit us to think about processes. Please remember Figure 1. The system depicted in Figure 27 uses *nodes*. That means that we can create a multithreaded application using the same architecture, where each node will be a thread. The only thing that the programmer has to change is how the sockets are configured (besides creating the threads, of course). The rest of the code will remain the same. **Hence, we do not have to differentiate between networked applications and multithreaded applications. They are all the same.**

In the next subsection we are going to exploit this idea to define our own programming model.

### 5.2.3 From threads to actors

As we saw in section 5.1, it should not feel awkward to treat communication between concurrency entities as human communications. It may seem shocking at first, but it should fit naturally into our brains. Our first approach is to try to transform threads into actors. All the communication problems are solved by using 0MQ sockets. Nevertheless, there are many ways of using them. After reading and studying the 0MQ Guide, we have seen that the typical way of programming a single threaded 0MQ program is as follows:

1. Create 0MQ socket and all the sockets needed for communication.
2. Configure the sockets (connect/bind to endpoints, subscriptions…)
3. Create a loop that has hardcoded the application behavior:

```
while(1)
{
```

```
  create request
  write in one socket
  wait for response (BLOCKS)
  read from the socket
}
```

The thread will block until there is some message to be read from the socket. The time in which the thread is blocked is wasted time. Nevertheless, the reader should be surprised by the simplicity of creating a networked application. There are no acceptor sockets, there is no need to create any threads that handle the connection. Everything is done by ZeroMQ.

However, it is too inefficient. We would like to do other things while waiting. ZeroMQ provides a **poll** function that can work with its sockets. Each socket can have three different events, but the interesting ones are[18]:

- **Receive event:** It is possible to read a message from the socket without blocking.
- **Send event:** It is possible to send a message without blocking (due to overloaded buffers).

Poll uses internally the best poll mechanism provided by the underlying OS (similar to what libevent does).

In order to use the poller, the programmer has to create an array of polling elements to be polled, and pass it to the poll function. Each polling element is a tuple of {socket, received/sent event}. A timeout can be also specified, so that if no events have been produced within the timeout, the execution continues.

Then, a more sophisticated solution can be used to create programs:

1. Create 0MQ socket and all the sockets needed for communication.
2. Configuration of the sockets (connect/bind to endpoints, subscriptions…)
3. Subscribe events into the poller
4. Use the same loop, but now we can do different things on different sockets

```
while(1){
  poll();
  if (read event in socket A)
```

[18] The third one is an error event, but we will not consider it.

```
            do something
    if(read event in socket B)
            do something
    if(send event in socket A)
            do something
}
```

This approach starts to look nicer. The program communicates with the exterior using sockets, and if we receive a stimulus from the outside world (a read event in socket x), then we can program a reaction. Nonetheless, it is not convenient enough:

- It is not easy to dynamically modify the group of events polled. The behavior of the actor is mixed with controlling structures (chain of **if** statements).
- It is not easy to subscribe events.
- There is no state encapsulation.
- Many times actors will have to execute some behavior in a timely manner. It is not clear how to do it.

A **first step** to improve this is to be able to connect events with functions. Instead of having to check with the **if** statements if some events have happened, the poll function should automatically execute the "Do something" part when an event happens. Hence, the code would be similar to:

1 Create 0MQ socket and all the sockets needed for communication.
2 Configuration of the sockets (connect/bind to endpoints, subscriptions…)
3 Subscribe elements into the poller. Now an element is {socket, event, function to be executed}
4 The following code would be needed:

```
while(1)
{
    poll();
}
```

With this structure we have successfully separated the behavior of the actor and the code that actually executes it (see Figure 29). The "function to be executed" is usually referred as a callback. Many C++ event-driven frameworks require the callback to be a pointer to a function with a given signature. However, this is very limiting. The main concern with this approach is how to maintain state between different calls. There cannot be any kind of encapsulation, and the context has to be a global variable.

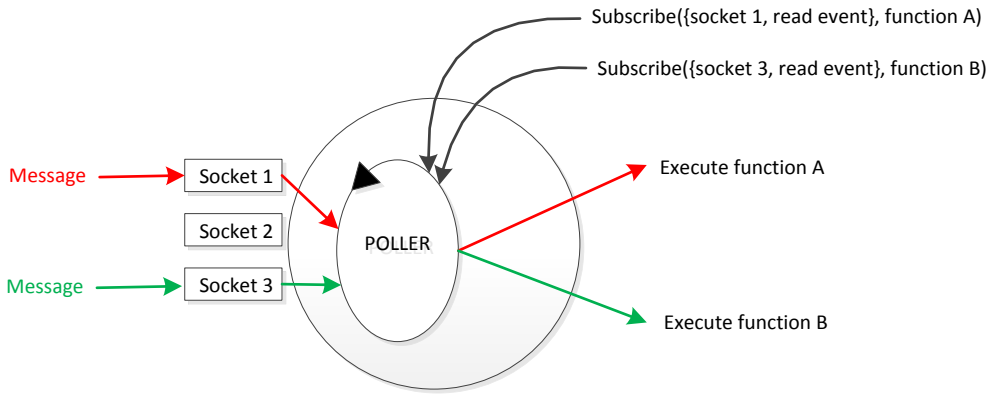



Figure 29 Reactor pattern

We would like to be able to encapsulate state into objects. Those objects could **maintain state** between different events. Classes are the Object-Oriented way to solve this problem. They encapsulate behavior and state. Nevertheless, in C++ pointer to object methods are not the same as pointers to functions, so we should provide two function signatures: One to subscribe object methods as callbacks and another to subscribe regular functions as callbacks. But, what happens if, for a different reason, we would like to use objects implementing the **() operator** as callbacks (the so called functors)? And what if we want to use lambda functions from the new C++ standard? All these problems would not appear in languages like python, but in C++ they are further than trivial. The problem is solved via the **std::function** class template.

A **second step** would be to integrate timers into the poller engine. Functions should be able to program other function to execute at a given time in the future.

With the idea of event handlers we have lost flexibility. In a **third step** we would like to be able to execute handlers not only if some event has happened, but also if a condition over those events has happened. Many times we would want to execute something if we have received a message from socket A AND a message from socket B.

We can improve this even more with a small amount of effort. Borrowing ideas from Aspect Oriented Programming Languages, we would like to add new functionalities to our actors without having to modify any function implementing its behavior. For instance, we would like to easily add a logger to log all the messages received, or we would like to modify the messages passed to the handlers (for instance, to implement a security layer). To do that, what we want is to be able to connect more than one handler to the same event (or condition of events). Not only that, but we would like to be able to add and remove dynamically handlers and also control the order in which they are called. This could let us modify the

actor behavior programatically, from a configuration file or even remotely. We could add new functionality without changing the source code. Again, using the right tool this is easy to implement: Boost.Signals.

Being able to encapsulate both behavior and state in the same object will give us an object-oriented actor. An application can be composed of different actors communicating with each others. In order to explain this, let us continue with the human analogy.

When a human body receives a stimulus from the outside world, it is received via one of the 5 senses (sockets). Then that stimulus is sent to different organs (components or object-oriented actors). Organs react upon events, and send back responses to those events. The responses depend on both the event received and the state of the component. For instance, if I see a pretty girl I will start to sweat. Depending on my behavior, I will walk and talk to her (send message to the leg actor). It may seem that we can do this with what we have right now. In fact, different functions can react upon the same event. If we encapsulate these functions in different objects, then we would have the organs. Each function could create a response to that event. However, we are forgetting a really important thing: **Organs communicate with each other internally**. A possible solution could be to execute methods of other objects. Nevertheless, that solution could produce recursive operations and hence, starvation of our actor. The communication between organs should be asynchronous, and all the organs should work on the same thread (body)[19]. We do not have to think in complicated mechanisms, we already have all the tools. The internal communication should be as easy as the external communication, and in fact, we can use again ZeroMQ: inter-process communication fits perfectly to our needs (see Figure 30)

[19] Although this is not compulsory, it allows us to share all the variables without synchronization.

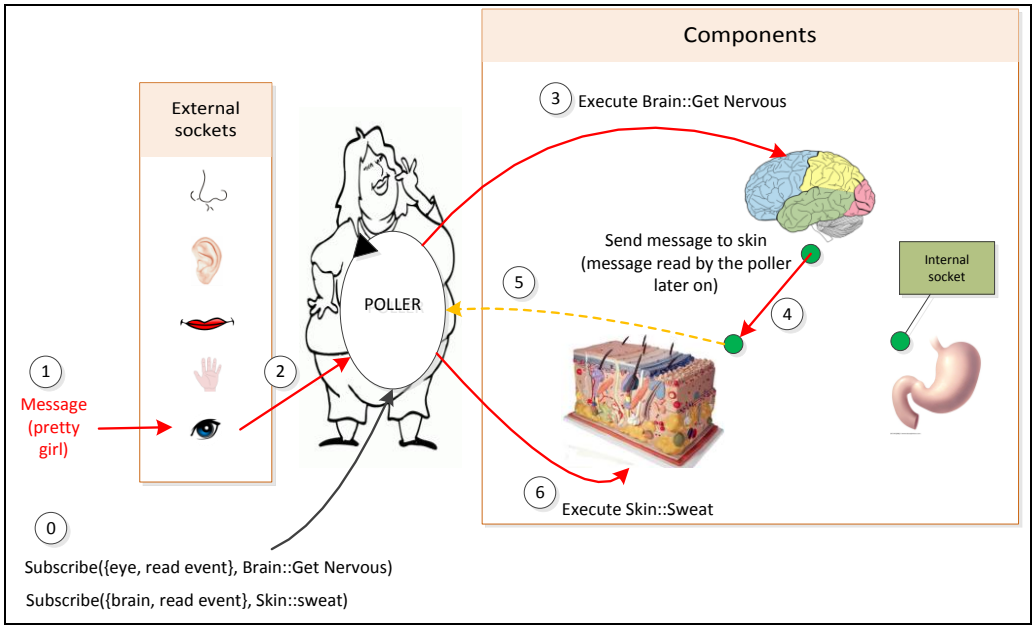



Figure 30 Human body as an actor

Let us recapitulate. The human body will be a process. Humans can communicate with the exterior using its 5 senses (ZeroMQ sockets using TCP, intra-process or multicast transport). The human body is composed of different organs (components). Each organ is completely defined by input events (which can come from the 5 senses or from other organs), output events (which can go directed towards one of the 5 senses or towards a group of organs), the behavior triggered by input events and which may generate output events. The behavior of a human can be very complex, but each organ perform a limited set of actions. It is the connection of all the organs what creates the complex behavior. This components are part of the program, and it does not matter where are executed. Moreover, organs can be exchanged. Let us suppose that we live in a world where it is possible to create hearts that pump more blood and let us run faster. That heart will have the same input and outputs, and hence the person just needs a simple "heart operation" to change the old one with the new heart. That new heart can be cloned and reused in different bodies.

We would like to create actors interconnecting components in the same way organs are interconnected. Moreover, when a component is written, we would like to reuse it in different actors. Of course allowing this is not easy. Connections between components can be performed using different kinds of ZeroMQ sockets. Each of them can use also different transport, and although we said that the API is the same, some considerations must be taken depending on the transport (specifically, between inprocess transport and the rest). We are not restricted to have all the components in the same thread, nor in the same process nor machine.

The next section introduces QZMQ, a library that tries to implement some of the concepts that we have talked in this section. Although it is still a prototype, it solves many of the problems discussed. Moreover, if the reader is interested in a really good C++ actor library, we refer to libcppa [23]. However, the libcppa library is also at its beginning, and is not integrated with ZeroMQ.

# 6 QZMQ: Quintiq ZeroMQ Binding

Gathering all the ideas from previous sections we have developed qzmq, a ZeroMQ binding for C++ which tries to implement a flexible system that lets us implement different kinds of actors.

## 6.1 What is qzmq?

Qzmq solves different problems in the same library:

- ZeroMQ is a C library. At the time of writing there are 3 versions: 2.2, 3.0 and 3.2.
    - 2.2: There is a header only C++ binding. It is a very simple binding that only provides the Resource Acquisition Is Initialization for zmq objects.
    - 3.0: A "high-level" C++ binding can be found in: https://github.com/benjamg/zmqpp. Although it is a good binding, it supports zmq 2.2 and 3.0. The 3.0 implementation changes the structure of the messages on the wire, and hence it is not backwards compatible with the 2.2.
    - 3.2: This versions recovers the message structure from version 2.2, making it compatible again with older versions. There exists no binding for this version. Since it was the last stable versions at the time of writing the thesis, we preferred to use it. Hence, **qzmq is a high level binding for zmq 3.2.**
        - Handles the life-cycle of context, sockets and messages.
        - Supports multipart messages
        - Google Protobuffer integration
- It is not convenient to use zmq with its poll function, as we've seen in section 5.2.3. QZMQ provides the **zpoller** object, which lets us implement an event-driven framework. Callbacks can be anything: pointers to function, pointers to a class method, functors[20] and lambda functions. It also provides the **ztimer** object, which lets the programmer to execute functions in a timely manner. The ztimer plays well with the zpoller, so all the functions are executed in the thread where the zpoller lives.
- Qzmq provides different actor implementations using zpoller. Each different actor is a base class which can be derived by the programmer, who can override some of the methods. It also provides some utility classes.

[20] Class implementing operator().

- Zproto: Lets us implement RPC clients and servers using Google Protobuffers.
- Zactor: Provides a really easy interface to configure sockets and subscribe events. It is the base class used to implement the task distribution framework.
- Zstore: Let the user to easily store global actor state on disk, using protobuffer definitions.
- OsSignalHandler: Provide support for Os signals (like Ctrl+c).
- Patterns: Implements some patterns in the form of templates. Right now there is just a thread-safe lazy Singleton class, which uses a lock-free algorithm to create the instance (we seek to not to use any kind of OS synchronization).

## 6.2 Using qzmq

The qzmq source code is heavily documented using doxygen. The source code contains a doc folder with the automatically generated documentation. It would be infeasible to try to explain all the components of qzmq in this document. Instead, we are going to gradually develop a service that receives requests from a router socket ("inproc://generate_number") and publishes random strings through a publisher socket ("inproc://random_number").

Before reading the following, we recommend the reader to read some parts of the official ZeroMQ guide [22] in order to see how to use the C API and understand the benefits of qzmq.

We will suppose that the following code is included in all the code snippets:

```
#include <qzmq/qzmq.hpp>
#include <string>
#include <randomString> // Let us suppose that this generates random strings


using namespace std;
using namespace qzmq;
```

### 6.2.1 Implementation 1

The first implementation will be used to see how qzmq simplifies the usage of ZeroMQ.

```
int main()
{
        random_gen random;
        zcontext ctx;
        zsocket sock_rout(ctx,socket_type::router); //Create the socket from the ctx
        zsocket sock_pub(ctx,socket_type::publish);
        sock_rout.bind("inproc://generate_number");//Bind the sockets. We are using interprocess com
                                                   //munication. Requests
        sock_pub.bind("inproc://random_number");
        zmessage msg;//Message container, used to receive and send.
        while(1)
        {
            sock_rout.receive(msg); //Blocking operation
           //Do something with the message
           msg.rebuild();//After "reading" the message, we rebuild it in order to send a new message
           //zmessage is a deque. We can push back and front.
          //Although ZeroMQ messages only store data, we can parametrize the functions in order to send
          //strings, ints or Google Protoboffuer objects. This method is planned to be extensible.
           msg.push_back<string>(random.gen());
           sock_pub.send(msg);
           assert(msg.size() ==0); //After sending, the code cannot access the message. It has been flushed.
        }
}
```

### 6.2.2 Implementation 2: Using the zpoller

Now let us suppose that we want to publish the random message when we receive a request from the endpoint "inproc://generate_number1" or from the endpoint "inproc://generate_number2". We cannot execute sock_rout.receive(msg), since it is a blocking operation. The solution is to use zpoller.

```
int main()
{
        random_gen random;
        zcontext ctx;
        zsocket sock_rout1(ctx,socket_type::router);
        zsocket sock_rout2(ctx,socket_type::router);
        zsocket sock_pub(ctx,socket_type::publish);
        sock_rout1.bind("inproc://generate_number1");
```

```
        sock_rout2.bind("inproc://generate_number2");
        sock_pub.bind("inproc://random_number");
        zmessage msg;
        zpoller poller;
        poller.add(sock_rout1,zpoller::POLL_IN);
        poller.add(sock_rout2,zpoller::POLL_IN);
        while(1)
        {
                poller.poll(zpoller::WAIT_FOREVER); //We can specify how much do we want to wait
                if(poller.has_input(sock_rout1))
                        //Send the message
                if(poller.has_input(sock_rout2))
                        //Send the message
        }
}
```

The zpoller object let us poll both sockets for a given event. In this example, the poll function is executed with the WAIT_FOREVER mode. This means that the thread will block until one of the events subscribed happen. In this case, we were interested in the POLL_IN event (that is, we can call the receive operation without blocking). However, we have the intuition that the code inside the while can get complicated. What we really want is to be able to link the event and the code that reacts in the same place.

### 6.2.3 Implementation 3: Using the zpollerm

To make things more interesting, we will suppose that when we receive a message from sock_rout1 we have to execute the function service1(), and when we receive a message from sock_rout2, the methods service2() from the following class:

```
struct MyService2
{
        void service2();
}
```

The zpollerm makes this very easy:

```
int main()
{
        zcontext ctx;
        zsocket sock_rout1(ctx,socket_type::router);
        zsocket sock_rout2(ctx,socket_type::router);
        sock_rout1.bind("inproc://generate_number1");
```

```
        sock_rout2.bind("inproc://generate_number2");
        zmessage msg;
        zpollerm poller;
        poller.subscribe(sock_rout1,zpollerm::POLL_IN,[]{service1();}); //Using lambda functions
        MyService2 s2;
        poller.subscribe(sock_rout2,zpollerm::POLL_IN,std::bind(&MyService2::service2,&s2)); //Using
functors
        poller.start();
}
```

The subscribe function is more powerful than it seems. We can even specify boolean conditions of events of different sockets. For instance, let us suppose that we want to execute the function service1_2() when we can read from the two sockets. We can use the macros QZMQ_COND_STATEMENT and QZMQ_COND to make this:

```
        zpollerm poller;
        poller.subscribe(QZMQ_COND_STATEMENT(
                QZMQ_COND(sock_rout1,zpollerm::POLL_IN) &&
                QZMQ_COND(sock_rout2,zpollerm::POLL_IN)
                ),//Conditions: Can be any boolean expression of QZMQ_COND
                &service1_and_2);//Callback
```

### 6.2.4 Implementation 4: Using the ztimer

Let us suppose that we also want to execute service3() every 1 second and service4() every 100 seconds but just 100 times. Doing this with just ZeroMQ would be a mess. However, the zpoller contains a timer inside.

```
        poller.timer.schedulePeriodic(ztimer::milliseconds(1000),&service3);
        poller.timer.scheduleNth(ztimer::seconds(100),100,&service4);
        poller.start();
```

### 6.2.5 Implementation 5: Abstracting everything into an actor class

The zpollerm's life-cycle is shown in Figure 31. The init, before_polling, after_calling and shutdown states are implemented using BOOST signals. Subscribing the proper functions to the proper signals makes it possible to create different abstractions, like a server that executes Google Protobuffer RPC requests (implemented in the zproto class) or an easy to use actor object, like the zactor. In this example we will create an actor that handles requests from a TCP socket. The implementation of the actor is as follows:

```
#include <qzmq/actor/zactor.hpp>
```

```
struct MyActor: public qzmq::actor::zactor
{


        MyActor(zpollerm& poller, zedge& edge):zactor(poller,edge){};
        virtual void subscribe_handlers()
        {
                subscribe("MyActor","service1",zpollerm::POLL_IN,std::bind(&MyActor::service1,this));
                get_timer().schedulePeriodic(ztimer::milliseconds(1000),[]{/*Do sth periodically*/});
        }
        virtual ~MyActor(){}
private:
        void service1();
```

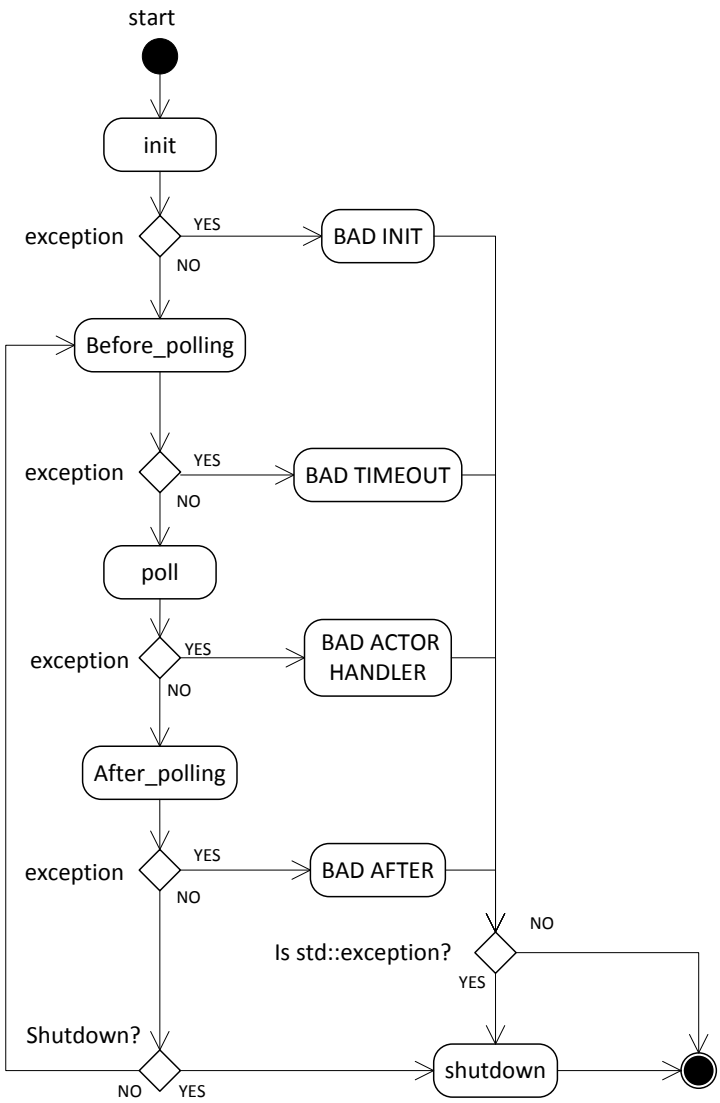



Figure 31 zpollerm life-cycle

The great thing about an actor is that the socket configuration is not hardcoded and can be configured at runtime, using the zedge and zpoint classes (which are derived from the zconfiguration template class). In fact, the zactor and its configuration is highly inspired by the **Cloudless project[24].**

```
int main()
{
        zpollerm poller;
        zedge edge;
        //It is not necessary to specify the zcontext. A default instance is used.
        edge.add("MyActor",

        zedgepoint().add("service1",zpoint().withType(socket_type::router).bindTo("inproc://service1"))
                .add("service2",zpoint().withType(socket_type::router).bindTo("inproc://service2"))
                );
        //Now the sockets are created when the actor is started by the poller, and not before. It is the actor
        ///the one that owns the sockets.
        MyActor actor(poller,edge);
        poller.start();
}
```

Now that everything is encapsulated in the MyActor class, it is really easy to execute more than one actor in the same thread. The only thing that we must take care of is that the model of execution that the callbacks use is a **run to completion** model (RTC). That is, if the callbacks execute a lengthy operation, they will not let the other actors to execute (nor the timer). Other execution models (like the ones used in Erlang, in which each actor can be stopped at any time) are really complex to implement, since these abilities must be provided by the kernel (we will not even consider this option, at least in this thesis). Another great thing about encapsulation is that now the actor can store its own context in private variables, and that context is maintained from different callback executions. Since everything is executed in the same thread, there is not need of synchronization, and it is really easy to program complex services with little effort.

### 6.2.6 Real life example: Handling OS signals from different threads

Although thinking about messages may seem difficult at the beginning, as soon as you start programming with this model many complex things can be implemented in a very fast way. The different routing characteristics that ZeroMQ provides make it possible to quickly prototype different solutions for the same problem.

One of these problems could be how to catch OS signals and shut down the application gracefully. Although each OS provides its own API to catch

signals, they follow the same convention: The application has to provide a callback that will be executed in an OS thread. Depending on the return value of the callback, the application will be stopped (what a typical Ctrl+C does) or will continue (in order to shut down the application gracefully). The callback can only access global variables.

A typical solution is to change atomically a shared Boolean variable "stop" from false to true. Then, the application thread can check this variable periodically, and if its value is true, shut down the application. However, what happens if there is more than one thread? What happens if that stop message has to be sent to a different process on the same computer? Can we do everything in the same step?

The solution that we have followed in qzmq is to use a publish socket in that function. Whenever an OS signal is caught, we publish it through the socket. These allow very flexible designs, like:

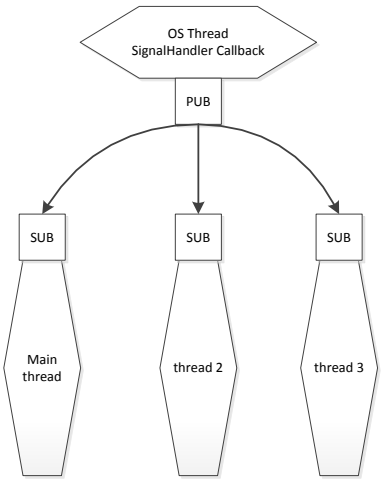



Figure 32 Signal handling 1

The Signal handler broadcasts that signal to all the threads subscribed to it. Then each thread can perform its own shutdown, and then synchronize between them.

Another possible option is to send the message just to the main thread, and then let it shut down one thread at a time, using other sockets like request-response or router-dealer.

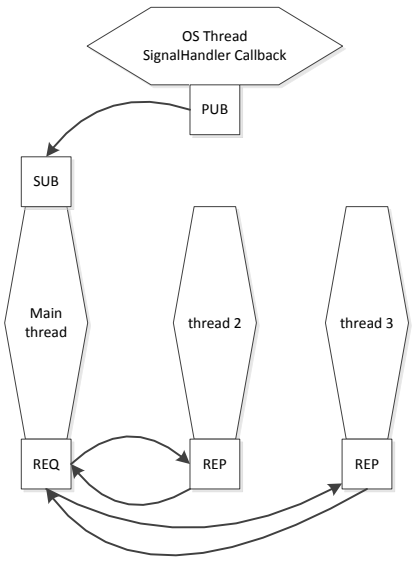



Figure 33 Signal handling 2

Using publish-subscribe pattern is dangerous in the sense that:

- If TCP is used as the main transport, then it might happen that the first messages publishes are lost ( when a socket subscribes, it takes some time to perform the physical connection between the two sockets).
- If inproc is used, then the publisher socket has to be the first to bind, and then and only then the subscriber sockets can connect to it.

However, these are just implementation details that the user of the system should not be aware of. What we have to realize is that this is a very powerful technique: just adding some sockets or changing some configurations we can implement a great variety of communication infrastructures between threads and machines.

# 7 Zookeeper

Although now we have the tools to program using the actor model, there are many problems that must be solved. From the analysis done in section 4, some of the problems that we should solve are:

- Detect failure detection of the worker nodes
- Provide fault-tolerance into the system: Fault-tolerance for the workers can be provided restarting the tasks on different workers. Nevertheless, fault-tolerance for the master node without relying on distributed file systems is not easy. We would need to keep a **consistent state** among the masters. To do this, **group-membership** would be needed. To implement it, we would need to implement an **atomic broadcast** before.

Each of these problems is on its own a very complex matter, and probably properly solving them would give us enough material to write more than one thesis. We mentioned in section 5.2 that there are many middleware services that could be completely valid to solve, for instance, group membership (Jgroups, ZeroC, Spread…). However, after analyzing them, we saw that these middleware solved too many problems, and we could advance with something simpler.

In our short study of Map Reduce implementations (see section 4.2), we discovered **Zookeeper**, a service for coordinating processes of distributed applications, which is used in the core of Hadoop. Although the Zookeeper infrastructure does not implement any specific coordination primitive, it exports an API that enables application developers to implement their own coordination services, like:

- Static and dynamic configuration
- Group membership
- Leader election
- Distributed locks
- Process barriers
- Failure detection

The API manipulates wait-free data objects (referred as **znodes**) organized hierarchically as in file systems (more particularly, as a UNIX file system)[21], as shown in Figure 34.

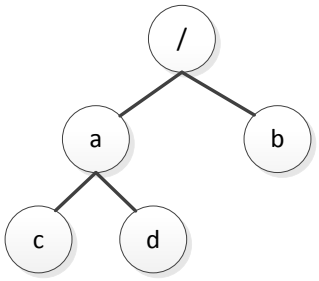



Figure 34 Zookeeper Znodes

Each znode is composed by:

- **Name (or Path):** A znode is accessed by its absolute path from the root ("/") znode.
- **Data:** Acts as a container of bytes. It stores unstructured data, so the application is the one that has to serialize the data. (In a standard configuration it can store up to 1 MB)
- **Timestamp** (When was it created, when was it modified…)
- **Version counters** (Every time the data is modified, the version is incremented by one)
- **ACL**: Access Control List, which identifies the users that can access the znode. **In this thesis we will no longer consider the ACL nor all the API functions involved.**

There are two different types of znodes:

- **Regular:** Clients explicitly create and delete them. They **can have children**.
- **Ephemeral:** Clients either delete them explicitly, or let the Zookeeper system delete them when the client that that created them terminates (deliberately or due to a failure). They **cannot have children.**

[21] The Windows filesystem has multiple root nodes

The basic operations that a client can perform over the zookeeper znodes are:

- Create
- Delete
- Check the existence of a znode
- Get the data (and the state of the znode, which includes timestamps and version counters)
- Set the data
- Get the names of the children of a znode (for instance, in Figure 34: getChildren(a)={c,d} )

The state of znodes is **consistently** maintained within a group of machines that run the Zookeeper server program. Those machines are interconnected with each other creating a quorum of nodes in order to provide **durability liveness** properties:

- If a majority of zookeeper servers is active and communicating, then the service will be available
- If Zookeeper responds successfully to a change request, then the change persists across failures as long as a quorum of servers is eventually able to recover.

Clients of the Zookeeper service can connect to whichever node that comprises the quorum. Inside this quorum, one of the machines is elected as the **leader** of the group. The other machines are referred as **followers.** When a client performs a read operation, it does it directly from the server it is connected to (**fast reads)**. However, write operations are routed to the leader node, which atomically broadcasts it to all the nodes (including itself), as seen in Figure 35. **Client sessions and watches, which will be discussed later, are also controlled by the leader.**

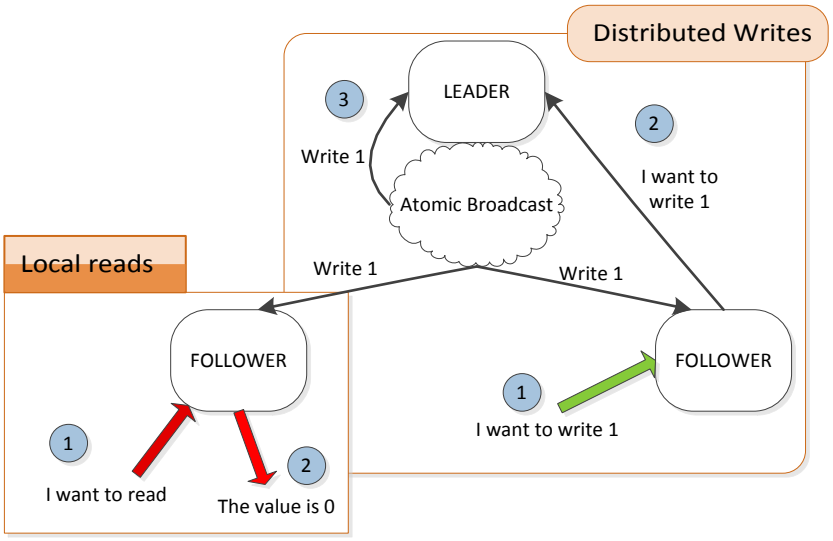



Figure 35 Reads and writes in Zookeeper

Zookeeper guarantees **linearizable writes and FIFO client order.** What does this mean and which are the implications of these guarantees? How much consistency this guarantees provide? In order to proceed, the reader must understand the concept of **linearizability.**

## 7.1 Linearizability

Linearizability is described by Herlihy [25] as a "a correctness condition for concurrent data structures […] which provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response".

We will say that an execution is linearizable if the operations of all the processes appear in some sequential order and the operations obey the global time order of operation.

We are going to try to give just an intuition of linearizability in order to progress in our explanations. Let us suppose that each client can execute just one operation at a time. Each operation has an invocation and a response. Operations are represented as shown in Figure 36.

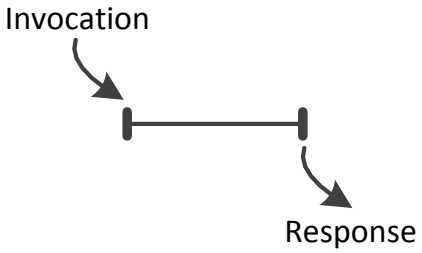



Figure 36 Invocation-Response of a concurrent operation

Now, let us suppose that we have a concurrent data structure that implements a register (which stores an integer value). A register has 2 operations: **W(Value)** writes the value into the register. A read operation reads the value stored in the register **R(value read).** The global time is represented as:

Figure 37 Global time representation

The executions in Figure 38 are linearizable. Nevertheless, the execution shown in Figure 39 is non linearizable. We cannot put points on the global timeline! Operations that fail (that is, they have an invocation but not a response) are treated as if they were completed at every node or never occurred. In Figure 40 we can see an example of this. Figure 41 shows a failed operation that appears as completed in one node, but never occurred in another one. Consequently, the execution is not linearizable.

This linearizability concept limits us to one operation per client at the same time (and hence, a synchronous operation). Zookeeper extends the idea of linearizability in order to provide **A-linearizability** (Asynchronous linearizability).

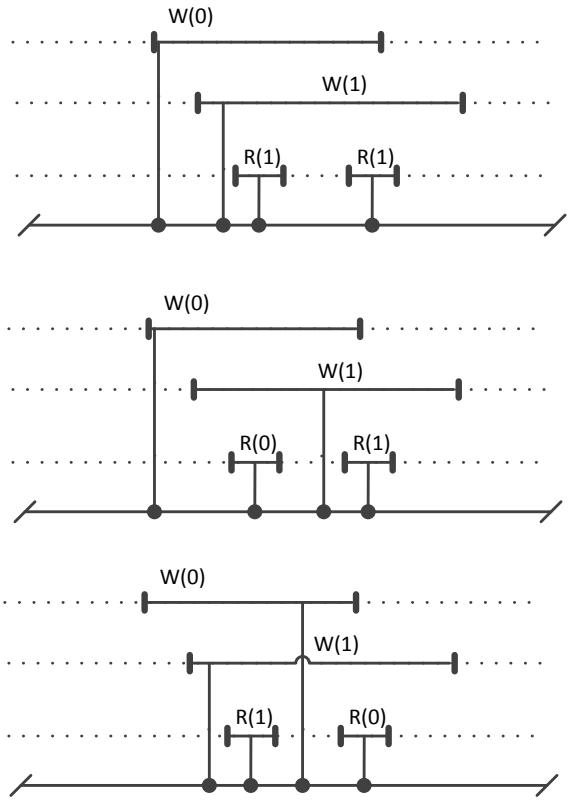



Figure 38 Linearizable executions

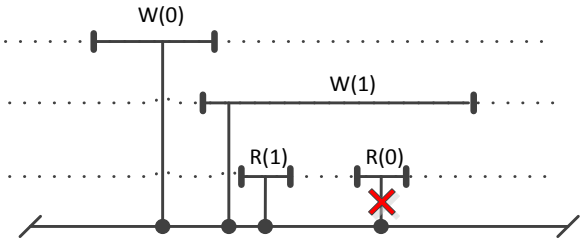



Figure 39 Non linearizable execution

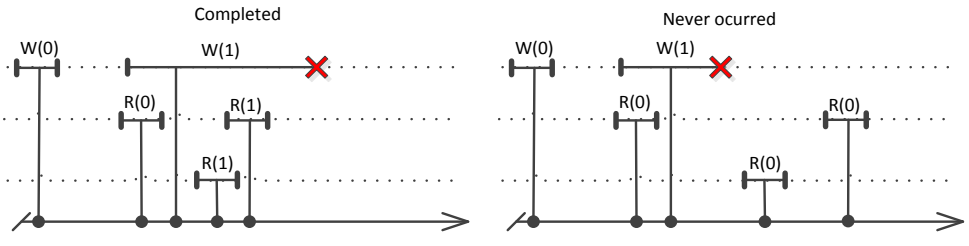



Figure 40 Linearizable executions with failed operation

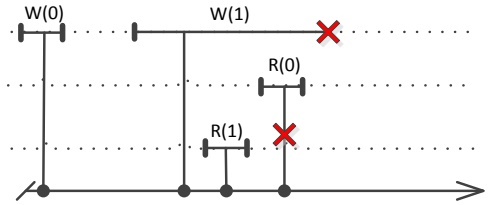



Figure 41 Non linearizable execution with failed operation

## 7.2 Asynchronous linearizability

One of the main key points of zookeeper is its performance. In order to achieve it, it lets the clients perform asynchronous operations (clients do not have to wait for the response of the first operation in order to invoke a new one, see Figure 42 ).

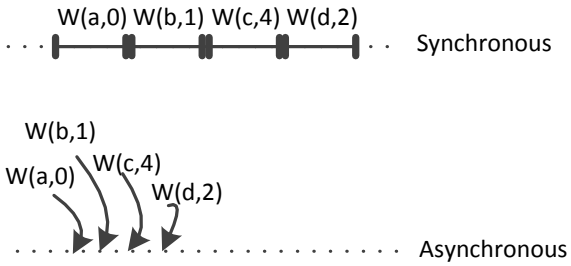



Figure 42 Asynchronous operations in Zookeeper. W(x,y) means write value y in register x

Nevertheless, asynchronous operations are always difficult to understand and implement. Algorithms using asynchronous operations tend to be very difficult to design and follow, since an operation request is not immediately followed by its response, and algorithms must be divided into small pieces. We will talk about how to implement asynchronous algorithms in the following sections. Before doing that, we need to fully understand the guarantees zookeeper provides:

- **FIFO order of client operations:** Although operations are asynchronous, they are sent one by one. Zookeeper guarantees that it will apply each operation in the order they were sent. Nevertheless, this just applies to a single client. If two clients send operations to zookeeper, **they will not follow the global time order**. This means that although client A may have sent an operation before client B, zookeeper does not promise that it will apply first A's operation and then B's operation. In Figure 43 we can see how although client

operations are interleaved, the order in which clients submitted the operations is maintained.

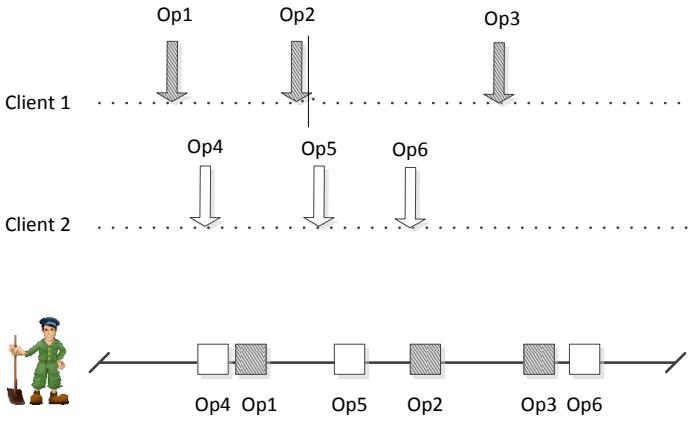



Figure 43 FIFO order of client operation in Zookeeper

- **Reads are performed locally.** This means that the system does not guarantee precedence order for read operations. Read operations may return the old values of a znode even though a more recent update to the same znode has been committed. If we have different clients connected to different zookeeper nodes, the following scenario could happen:

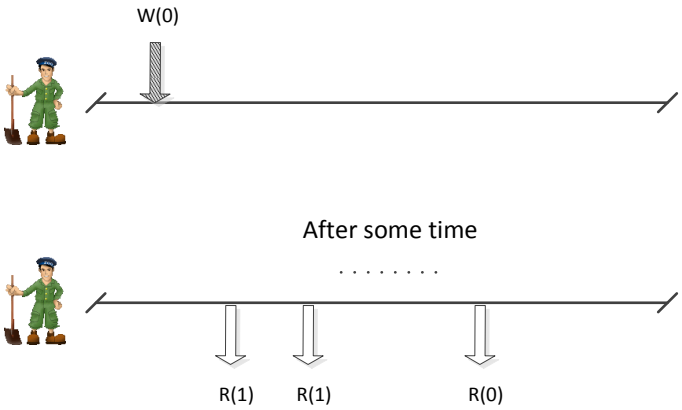



Figure 44 Consequences of fast reads in Zookeeper

Write operations are slow, since atomic broadcasts need to propagate it to a quorum of servers.

In fact, the problem could be even worse, since the same client writing to the same zookeeper node could read old values, as seen in Figure 45.

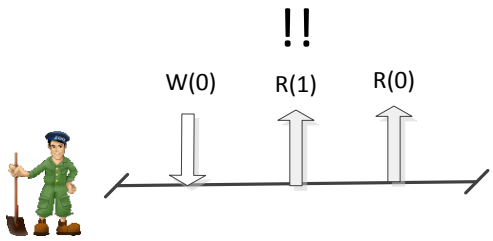



Figure 45 No sequential ordering

Although the behavior presented in Figure 44 is reasonable, the one in Figure 45 is not. A client should see at least a **local consistency** (otherwise it would be extremely difficult to program any kind of algorithm). If a client writes the value 0 in one znode, and if we suppose that no other client has changed its value, then the reasonable behavior is to read 0 from that znode. This follows the FIFO order of client operations. Zookeeper solves this problem using a transaction number **zxid** in each operation. The local state of the znodes in each zookeeper server also contains a transaction number. When a zookeeper node receives a read transaction with zxid greater than the state of its znodes, it waits until its local znode state is at least equal to the zxid of the read operation. Then, a possible scenario is the one seen below:

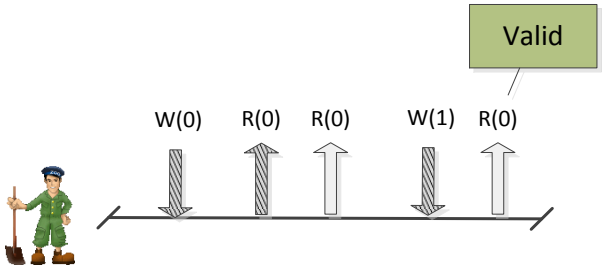



Figure 46 Sequential ordering in Zookeeper

The client maintains always a consistent view of the system. It W(0), and then R(0), updating the state of the znodes to the last version. Hence, a read of a different client will return also a 0 value. Nevertheless, after the client W(1), suddenly the client R(0)! This behavior is completely **valid**. The read operation from the client does not contain the newest transaction number, so the server is not required to synch with the leader. The W(1) operation has not been atomically broadcasted to the servers yet, so the client reads an old value. However, this client still has a consistent view of the znodes! The conclusion is that **FIFO order is not preserved among different clients**. Most importantly, it is not preserved even if the clients are

connected to the same server. Not having this property allows zookeeper to provide **fast reads**.

From the last example, we can see that if a client wants to have an updated view of the znodes's state, it has to first perform a write operation and then a read operation. The transaction number will do the rest. In order to improve this, zookeeper provides a special operation: **sync.** The sync operation forces the zookeeper node that receives it to apply all the possible updates to the system. Hence, a possible way to read always updated versions of the znodes values is to do a synch followed by a read operation (Figure 47). This combination provides the same properties than the ones given by Herlihy's linearizability[22].

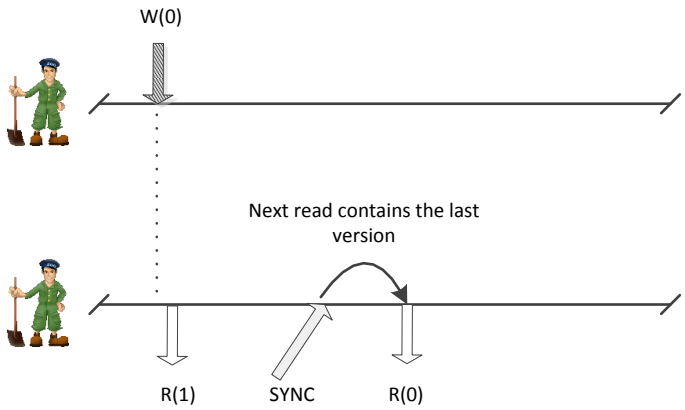



Figure 47 Synch + Read combination

Nevertheless, this method forces the system to do synchronization even though maybe the value of the znode the client is interested in has not changed its value. In order to solve this problem, Zookeeper provides the ability to set a **watch** over a particular znode. The operations **exist, get** and **getChildren** provide the ability to specify a **watch callback**. When a watch is set in a znode Z by a client C specifying a callback CB, the Zookeeper server will notify C when Z changes its value, is deleted or something has happened to its children. After the notification, C will execute the callback CB[23].

Watches are **one-shot triggers**[24]. If the client wants to be notified again, he has to set a new watch. Watches notify the type of change, but they do not include more information than that. In the following figures we are going to explore how to use watches.

[22] Well, more or less. There might be a znode modification between the sync and the read operation. This can be solved using transactions (which zookeeper provides).

[23] More about callbacks in next sections

[24] Not entirely true, we will revisit this concept later.

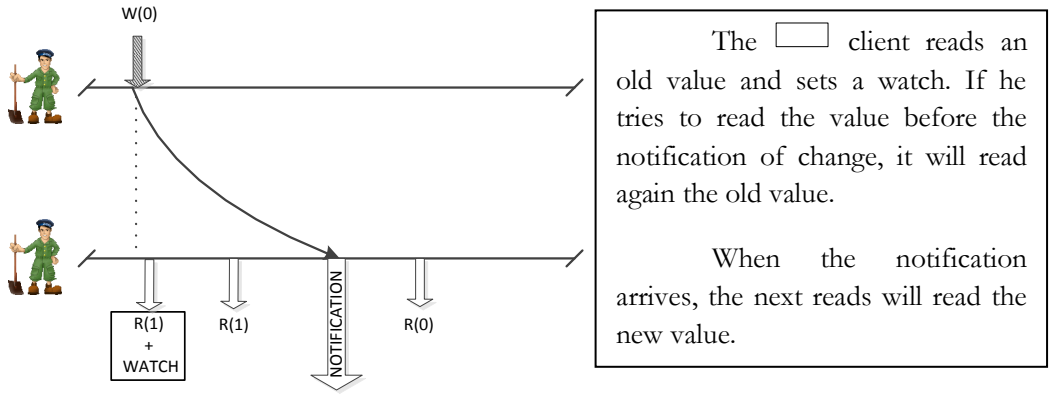



Figure 48 Setting watches in Zookeeper

This is referred as **order guarantee for the notifications** in the Zookeeper paper:

> "If a client is watching for a change, the client will see the notification event before it sees the new state in the system after the change is made. "

Hence, after the notification, the client is guaranteed to see the new state. He is also guaranteed that if he performs a read operation before the watch, he will read the old value**.**

**Watch notification should be the only way of synchronization between different clients**. Let us suppose that the client writes 0, and notifies thorough a communication channel to the client that the value has changed. If that notification arrives before the watch notification, then the client could read an old value (and interpret it as the new value).

To finalize this section, we consider important to understand why watches are single-shot notifications[25] that do not include the data changed. The only purpose of watches is to notify that a change has happened. The client will do what it finds convenient with this information. It could happen that two different updates to the same node are produced at almost the same time. If notifications were not single shot, two messages would be sent to all the clients interested. **If notifications carried the data changes, messages would be heavy to send.** Many times clients probably will not want to have the two updates, but just the last one, or even none of them. Zookeeper watches allow to do something like this:

[25] Again, not completely true

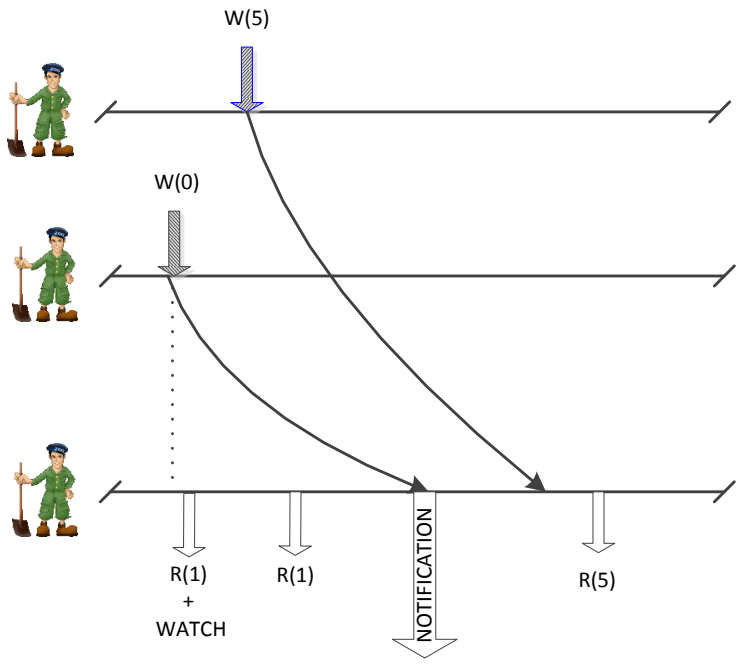



Figure 49 In zookeeper watches produce single-shot notifications

Using just one message (notification), the ▭ client will read the last version of the value, which is probably what he wanted in the first place! Let us suppose that a client writes a znode, and the transaction number is 1. Zookeeper promises that all the clients that read that znode after having received a watch will read **at least** the version from transaction 1 (but they might read newer values). In this example, the ▭ client is reading the last value written (5), which was wrote by ▨.

### 7.2.1 Final thoughts about Zookeeper

Now that we understand what Zookeeper promises, we will try to unleash all its power. Programming distributed applications with zookeeper might seem trivial: anyone without experience might argue that Zookeeper is just a file system with some strange properties. However, not many people are used to thinking of a distributed system as a big file system. This lack of experience made the zookeeper authors to propose what they call **recipes:** small algorithms that implement services like failure detection or leader election. For instance, a failure detector can be created using ephemeral nodes and watches over them:

- When a client connects, it creates an ephemeral znode.
- If someone wants to be notified about the failure of that client, it creates a watch over that ephemeral znode.

- When the client fails, the ephemeral node is automatically deleted by the zookeeper system, and all the other clients will be notified about the deletion (and hence the failure of the monitored client).

The great thing is that any kind of application can benefit from such a service without having to implement heartbeats, timers or whatever algorithm is used in a typical implementation of a failure detection service. Everything is handled automatically by Zookeeper.

But, the recipes proposed are far away from perfection. They do not deal with almost any kind of error, exception or corner case than can arise due to connection, server or client failures. Depending on when or how a failure is recovered, the client may lose the consistent view of the system. How can someone implement an algorithm where all the operations can fail in multiple different ways and with multiple different consequences? One could say that computers and connection almost never fail. However, if we say so, why do we want to use Zookeeper? We want to use it to cover those strange moments in which something fails.

The big problem that we faced while trying to implement an algorithm using Zookeeper was that although the library documentation mentions all the possible errors, there is no guideline explaining what to do upon each error, or explaining best practices.

In fact, as we will see in the following section, the Zookeeper API is not as easy to use as it seemed in the first place.

## 7.3 Zookeeper API

Now that we understand what is Zookeeper, it is time to keep the animals. As we will see, from the theory to the practice there is a big gap. The zookeeper asynchronousity added to the error unpredictability will make recipes different to model and to implement.

Even though the zookeeper server must run on a Java Virtual Machine, there are client bindings for most of the common programming languages. The officially supported clients are the Java and the C client.

The Apache Mesos, a Dynamic Resource Sharing for Clusters [26] use internally a Zookeeper C++ binding. However, it is just a simple wrapper and does not solve any of the complex things about zookeeper.

There is a very interesting project, the Netflix Curator [27], which provides a high level API that greatly simplifies using Zookeeper. It also provides correct implementations of many recipes. The main problem is that it is a Java library. Nevertheless, the discovery of Curator was very beneficial for us, we have learned a lot from it while implementing our own C++ zookeeper library: **QZK**.

The main difference between Curator and QZK is the environment in which they are executed. Curator is thought to be used in a multi-threaded application, whereas QZK is thought to be used by QZMQ actors.

To begin with, we will explain the main points to use the C client Zookeeper library. While doing it, we will delve into some more complex concepts of Zookeeper.

### 7.3.1 Establishing a connection with the server

The connection with the servers is handled by the **zhandle_t** object[26]. Zookeeper stores all the global state of the connection inside **zhandle_t**, which is explicitly instantiated and initialized (using **zookeeper_init**) by the application. In order to initialize it, the library needs two important objects: The **clientid_t** and a global watcher function **watcher_fun**.

#### *Client session*

Servers identify each client via a unique **clientid_t**. Of course, the first time that a client tries to connect to the system it will not have any clientid_t yet. In that case, the Zookeeper server will generate one automatically. In reality, each clientid_t uniquely identifies a **client session**. Each session represents an "alive" connection between a server and a client.

Servers and clients monitor each other using a failure detection mechanism, which can be further explored in [28]. If the server has not detected the client to be alive before the session **times out**, then it declares the session as expired (and hence, considers that the client has crashed). When a session is expired, all the **ephemeral znodes** created by it are automatically deleted.

#### *Session watcher*

Until now we have treated watches as asynchronous "notifications". The Zookeeper library implements these notifications using callbacks. A callback is a function that is executed by the library when a given event happens. The

[26] The C language does not have the concept of an object, but we will use it anyway. It makes the explanations easier to understand.

Zookeeper library proposes two different ways of dealing with watches. The first one let us specify a global function watch that will be executed whenever an event happens. There are two types of events:

- **Session event**: Executed when the state of the zhandle_t changes. It lets us know when the connection has been established, when it is lost or what happens to the session (recovered or lost).
- **Watch event**: It informs that something has happened in a given znode: It has been created, deleted, changed, a change has occurred in its list of children or that the znode is no longer watched (due to a resource constraint in the Zookeeper server).

These types of events are orthogonally different, so it makes no sense to handle everything in one function. The second way of dealing with watches solves this problem: The **watcher_fn** specified in **zookeeper_init** can be used just to deal with session events. Then, it is possible to specify a callback for each zookeeper operation. Each callback can be accompanied with a void pointer, referred as the **callback context**, which can point to anything that the programmer wants (it is needed if we want to let the callback access non global variables). The callback is executed in a different thread, which means that synchronization between threads is required.

### *Session state*

Understanding the life cycle of the session is crucial. In the zookeeper documentation we can find the following figure, which shows all the possible state transitions:

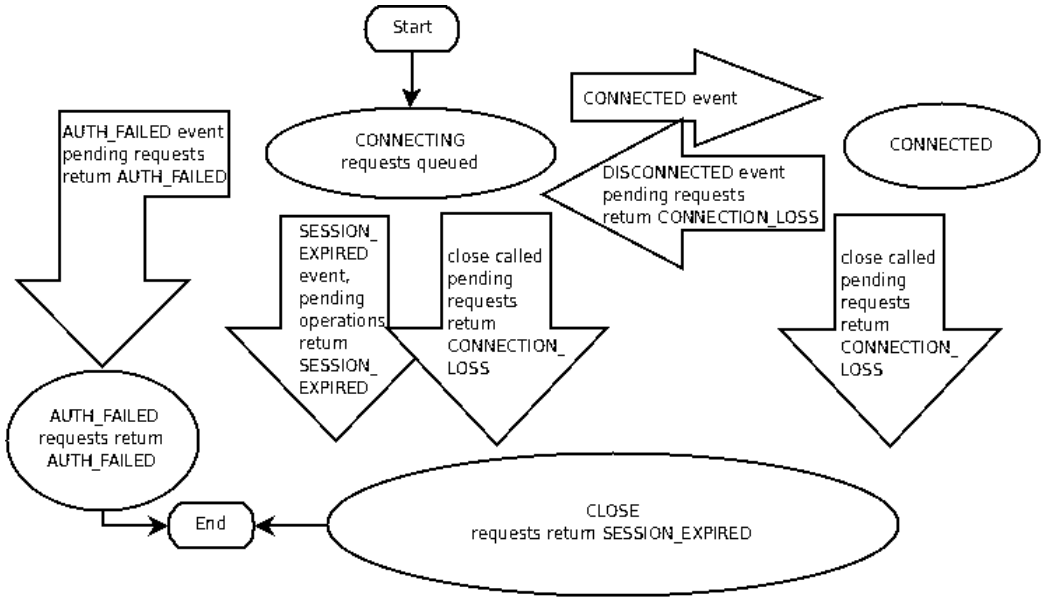



Figure 50 Zookeeper session state transitions

Zookeeper comes with two possible flavors: single-threaded (asynchronous) and multithreaded (synchronous + asynchronous). For simplicity we will continue the explanation using the multithreaded flavor. The Zookeeper developers recommend not to use the asynchronous API unless you want to integrate zookeeper in some kind of event loop, which is exactly what we will want to do: Integrate it within the zpoller in order to be able to use zookeeper within our actor framework (QZMQ).

The **zookeeper_init** function creates a new thread where all the state (connections and reconnections) is handled automatically. The function is non blocking, so the calling thread does not wait until the connection is established with the Zookeeper server (operations after calling zookeeper_init will probably be executed in the connecting state). The **zhandle_t** object is thread-safe, so it is possible to share it with different threads within the same process.

Operations over znodes can be both **asynchronous** and **synchronous** (that is, the block the entire thread until they have completed). Synchronous operations are easy to follow (they let us write an algorithm in the same method). However, asynchronous operations are very difficult to use. In order to call an asynchronous operation, the application has to provide a callback that will be executed when the operation completes. This callback will be executed in a different thread. Hence, if we want to use asynchronous operations we have to write the algorithm in the form of callbacks calling other callbacks (which moreover, will be executed in different threads!). Since the algorithm is divided in callbacks, it can be difficult to understand by other programmers and are a pain to debug.

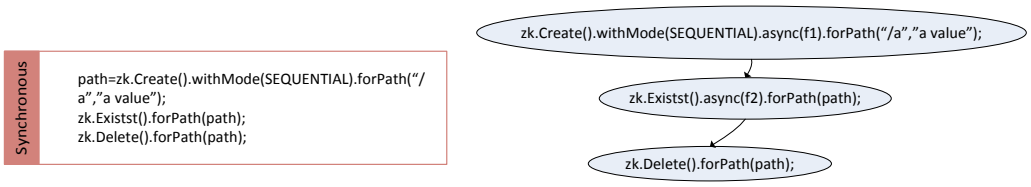



**Figure 51 Synchronous (left) and asynchronous (right) Zookeeper algorithms**

The Zookeeper client will try to recover all the connections automatically. However, there are times in which it is not possible. If the session cannot be recovered, then the session status will be CLOSED, and the application is responsible for closing the zkhandle_t and initialize it again. Session expiration is a fatal error: all the ephemeral znodes are removed. The library cannot know the logic behind the ephemeral znodes, so it is the application the one that has to start a new session and reconstruct or maintain the znode hierarchy.

### *Watches revised*

Watches can be set from the zoo_get, the zoo_exists and the zoo_getchildren functions. Each function lets us specify a pointer to a **watcher_fn** callback that will be executed in a different thread when one watch event happens. Watch semantics can be a little bit confusing. Let us suppose that…

- We call the function exists on node "/a", and that it does not exist. When it is created, we receive a **ZOO_CREATED_EVENT** notification.
- We call the function exists on node "/a", and that it does exist. When it is deleted, we receive a **ZOO_DELETED_EVENT** notification.
- We call the function get on node "/a", and that it does exist. When someone changes its data using set, we receive a **ZOO_CHANGED_EVENT**.
- We call the function getChildren on node "/a", and that it does exist.
    - If someone creates "/a/a1", we receive a **ZOO_CHILD_EVENT** notification.
    - If someone deletes "/a/a2", we receive a **ZOO_CHILD_EVENT** notification.

As we have seen, there is no way to know if a child has been created or deleted with only the information the watcher carries.

We explained in section 0 that watches are single shot events. As we stated, this is not entirely true. Watches also carry session information, and the callback specified to be executed might be executed more than once by the library. This is not mentioned in the documentation, and might lead to incorrect behaviors. The life cycle of a watch is the following one:

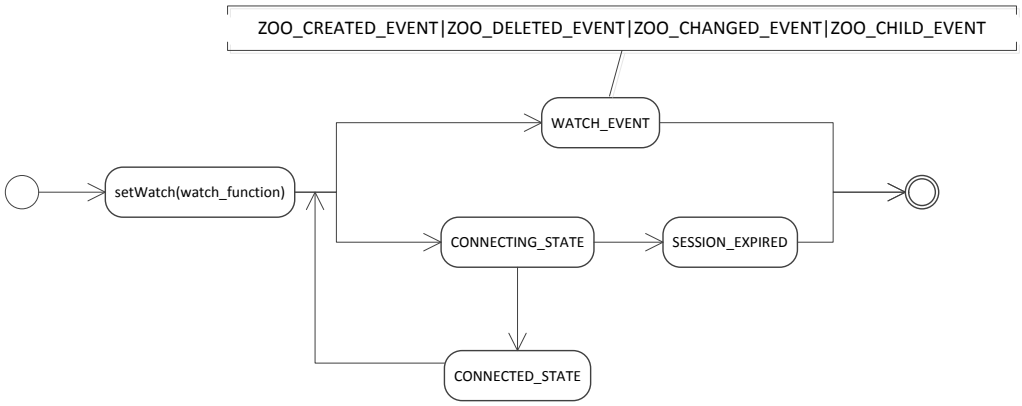



Figure 52 Watch life cycle

Hence, the same callback can be executed several times. The usual path is the upper branch, which is executed when something changes in the znode structure. However, if the connection is lost before the watch is triggered, then the library will try to recover it. If it can do it and the session is maintained, the watch is considered to be still valid. However, if the session is lost (SESSION_EXPIRED) the callback will not be executed again. It is the user the one that knows the functionality of the callback, so he can react upon the SESSION_EXPIRED event in the same callback.

### 7.3.2 Understanding the errors

In a distributed system, many things can go wrong (see section □). Zookeeper is not an exception. From the Hadoop Wiki [20] we see that we can identify 3 possible kind of errors:

- **Normal state errors**, like deleting a node that does not exist or creating a node that already exists
- **Recoverable errors,** like disconnected event, time-out operations or connection loss
- **Fatal errors,** like session expirations or disconnections.

The normal state errors are not complex to deal with, and it is the application the one that has to understand them and react. However recoverable and fatal errors are intrinsically complex for two reasons:

- They do not depend on the logic of the program and can happen at any time. The programmer has to be always aware of them.
- It is possible to get information about these errors from 4 different places, and dealing with them can be tricky if there is not a proper error handling strategy.

Specifically, recoverable and fatal errors can be detected from:

- Return code of a znode function call:
  - Connection Lost
  - Session Expired
  - Operation Timeout
  - Session Moved
- State notification in a watch callback and in the global callback (see Figure 50)
  - Expired session
  - Connecting state
  - Connected state
- It is also possible to know the state of the connection using **zoo_state** (see Figure 50)

### *Recoverable errors*

When a recoverable error happens, the programmer does not need to worry about the validity of the zookeeper handle object. The library will automatically try to reestablish the connection. However, it is important to understand the implications of each error in order to react properly:

- **Connection Lost:** The connection with the server is lost. It can be due to multiple reasons: the server we were connected to has crashed, there is congestion in the network… Nonetheless, the programmer has to be very prudent. If the connection is lost while we are doing an operation, two things may have happened, as we can see in the figure below

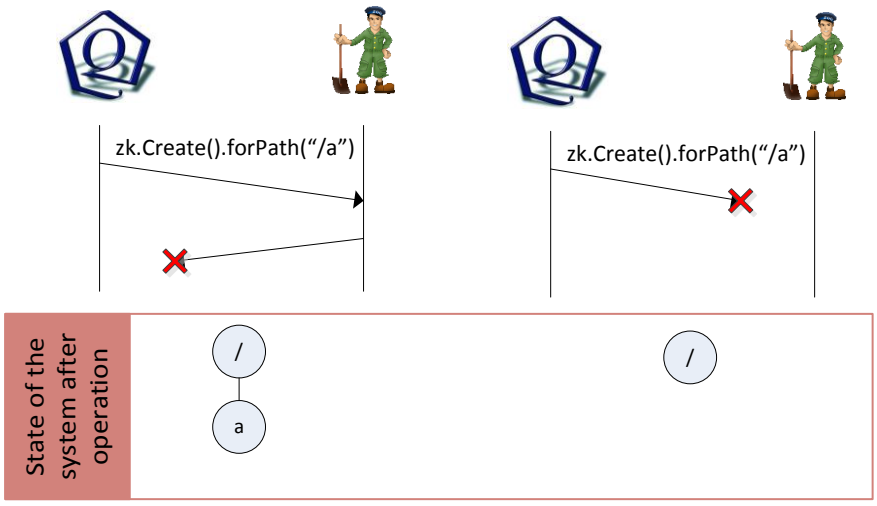



Figure 53 Connection loss error in zookeeper

The zookeeper library cannot decide what to do after a connection loss, since it is impossible[27] to find out which of the scenarios have happened.

In this example, the create operation is **non-idempotent.** Hence, it is the application logic the one that has to deal with these problems, trying to discover what happened and re-executing (or not) the operation. Other operations, like getData are **idempotent** (they do not modify anything). However, Zookeeper does not try to retry these operations, since this may violate ordering guarantees**. How to deal with non-idempotent operations will be one of the main points in our task-distribution algorithm**. Another problem that is left to the programmer is **when to retry the operation**. The Curator library proposes a Retry Loop mechanism. Operations executed within a retry loop will be re-executed each following different retry policies (uniform timeout, exponential back-offs). However, this is not an optimal solution. In section 7.7 we will propose an optimal solution that makes use of the information from the global state watch.

- **Operation timeout:** The client has not received any response from the server within 2/3 of the session timeout. This can be understood as another form of

[27] It would not be impossible, but expensive. Moreover, making requests re-issuable made the read requests become problematic (see [29]).

connection loss, so the programmer has to take the same precautions. If Zookeeper is configured to form a quorum of servers, the client library will try to connect to a different server after 2/3 of the session timeout (If the timeout is due to the crash of the server we were connected to, then we can still try to maintain the session connecting to a different server).

- **Connecting state:** We will get this information from both the global context callback and from ALL the watches set. This might mean that we can get hundreds of connecting state notifications at the same time. Hence, using this information is difficult. The fact that callbacks are executed in different threads, and that the same event can be notified from many watch callbacks at the same time makes it difficult to correctly react upon it: Who should react? The code in the callback? Code in the main thread? How do we synchronize them?
- **Session moved:** The session moved error is a hard to see. We can find the explanation in [30]

> "This exception occurs because a request was received on a connection for a session which has been reestablished on a different server."

In this case, the zookeeper cluster has detected that the session has been moved, **so the operation is NOT executed**. Hence, the only thing that the programmer has to do is to re-execute it again.

#### *Unrecoverable errors*

The **zhandle_t** object becomes invalid when it is explicitly closed (using **zookeeper_close**) or when a fatal error succeeds (which is always seen as a **session expired error**). When a session is expired, all the ephemeral nodes are automatically deleted. It is the application which knows the exact morphology of the znode structure, so it is the application the responsible for initializing the handler again (using zookeeper_init without specifying a clientid_t), and recreate again the ephemeral znode structure. However, doing this in multithreaded applications is difficult. One of the reasons is, again, the vast number of sources from where we can detect that the session has been expired. This makes hard to express a zookeeper algorithm.

### 7.3.3 Specifying Zookeeper recipes

Different Zookeeper recipes can be found in the official documentation [31]. We consider these recipes incomplete in the sense that:

- They do not consider any recoverable error, and hence it is not defined how to recover from them (is there a retrial mechanism?).
- Use a high-level description written in prose, which makes it difficult to make a direct translation from the description of the final implementation.
- Session expiration not fully considered (a client crashing can be considered a form of session expiration, however, what happens if the session expiration succeeds due to a connection loss?)

Although the curator library solves some of these problems (provides a to securely initiate connections and treat recoverable errors), it does not provide a simple way to express the recipes.

After experiencing the difficulties of designing new recipes, and after developing a zookeeper library which works with actors, we have realized that a powerful tool to express and implement zookeeper algorithms are state machine diagrams and hierarchical state machines.

In this section we will propose a novel strategy to program zookeeper recipes which highly simplifies how to think and develop them.

Writing zookeeper recipes is easy if we do not consider the connection and session errors. However, expressing a zookeeper algorithm that has to handle all the possible execution paths is really complex, and there is no standardized form to do it. Implementing the recipes is error prone, since it is very easy to not to handle all the possible errors. Even if we want to handle everything, it is difficult to implement a system that does it.

After dealing with many zookeeper recipes and their implementation (in frameworks like Curator), we noticed that they were really difficult to understand: there is not a common framework to express the recipes. **We realized that state machines are powerful tools that can facilitate the design and implementation of zookeeper algorithms**.

Nevertheless, in order to use state machines it is not enough to have an event-driven framework like QZMQ. We also need to build the machinery that models the state machine, which is not trivial. The book Practical UML Statecharts in C/C++ [32] describes how a complete state machine framework should look like.

## 7.4 What do we have and what we do not

Let us recapitulate for a moment. During the thesis we have developed our own event-driven framework (QZMQ) and we have created an ecosystem of actors that lives within it.

In the next section we will present a Zookeeper C++ binding which works with QZMQ actors. However, although we thought that was enough to implement Zookeeper algorithms comfortably, after trying to implement some recipes we realized that these tools (QZMQ+actors+QZK) were not powerful enough. The problem was not only in the tools, but also in our own ability to describe the recipes. After trying some approaches (from prose to znode draws explaining each step of the recipe), we realized that statecharts might be the proper tools. We immediately started looking for state machine implementations and discovered the Qt C++ framework, which provides a State Machine Framework. However, it was a surprise to learn about its signal-slot mechanism, which shared many aspects that QZMQ offers, but with 17 years of development behind. The signal-slot model is not exactly the same, though:

- Each Qt thread may contain a single queue, which processes objects of type QEvent. The queue is read by a loop, which redirects each QEvent to the appropriate QObject (the receiver object). The signal-slot model works in a multithreaded application (objects can send QEvents to objects living in different threads if that thread is running its own event loop with its own queue), but it cannot be used between processes or machines. QObjects do not need to be subscribed to the event loop. Each QObject has a thread affinity with the thread where it was created. This affinity matches the QObject with the event loop. We would like to learn more about how is this implemented (for sure, it uses thread-local storage, a block of data which is linked with each thread, and that can only be accessed from inside the thread).
- In QZMQ, each open socket represents a queue. Each active object (actor) can have multiple queues. We could think that each actor is a QObject, and each socket is a slot. Instead of sending QEvent objects, QZMQ sends raw data (which has to be marshaled and unmarshaled). The biggest advantage of QZMQ over Qt is that the same approach can be used to communicate between processes and machines.

The great thing with the Qt State Machine Framework is that it is highly integrated with the signal-slot mechanism [33]. After studying, we decided to build our own *theoretical* zookeeper state machine framework based on the abilities of the Qt State Machine Framework.

First of all we will describe QZK and after that we will slowly introduce state machines and how to use them with Zookeeper (which again, are not implemented). At the same time we will explain the common pitfalls while implementing recipes through examples that use the concepts explained.

## 7.5 QZK: C++ Zookeeper API

The API exposed by the C library is not easy to work with. In this section we will introduce our own Zookeeper C++ API, which is partially based on the API exposed by the Netflix Curator Zookeeper client [27].

QZK is divided in two objects: The qzk_server and the qzk_client (see Figure 54).

- The **qzk_server** actively uses the Asynchronous C Zookeeper API, and is highly integrated with the QZMQ zpoller. It is an actor that automatically handles the connection-reconnection with Zookeeper, but that also

automatically creates a new session if the last one was close. It also handles session recovery after a crash.

- The **qzk_client** is an actor that communicates with the qzk_server, receiving the events triggered by it and exposing an interface to be able to execute operations and set watches.

Whereas the C API uses callbacks to execute watches or to execute asynchronous operations, qzk_server uses messages defined by Google Protobuffer definitions, transforming Zookeeper calls to asynchronous messages. Hence, multiple qzk_client can connect to the same qzk_server actor, and trigger operations and set watches. There are up to four communication flows:

- Session watches: qzk_server publishes each session state change. If qzk_clients are interested in receiving updates, they just need to subscribe to the publisher socket (PUB).
- Synchronous operations (znode operations): qzk_clients can send a synchronous operation message to the ROUTER 1 socket. The response will be sent to DEALER 1 (Figure 55).
- Asynchronous operations: qzk_clients can send an asynchronous operation message to the ROUTER 1 socket. The response will be sent to DEALER 2 (Figure 56).
- Watches: When a watch is triggered, a watch message is sent to DEALER 2.

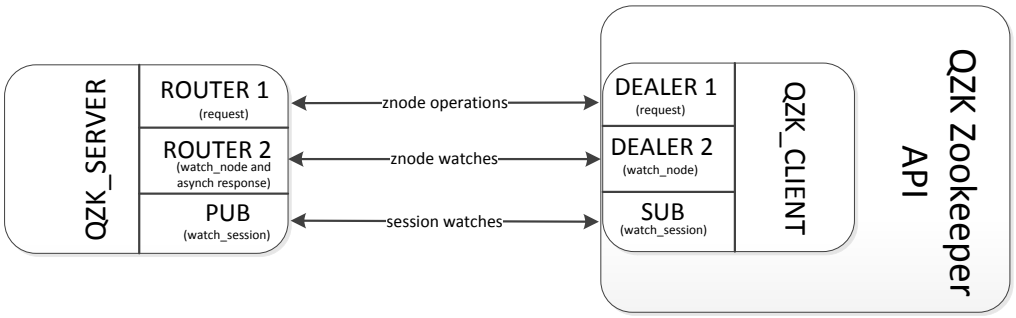



Figure 54 qzk_server and qzk_client implementation

We have used a custom ZeroMQ routing protocol in order to route asynchronous responses or watches from DEALER 1 to DEALER 2. The actual implementation can be explored in the source code attached.

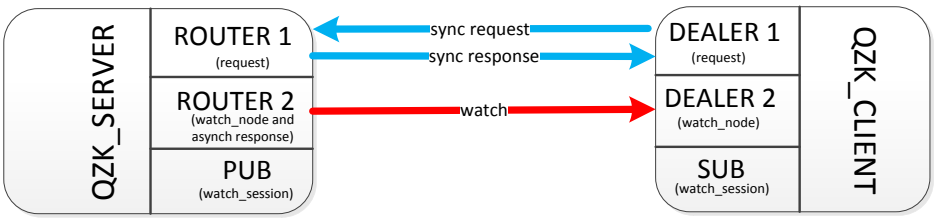



Figure 55 Synchronous data flow

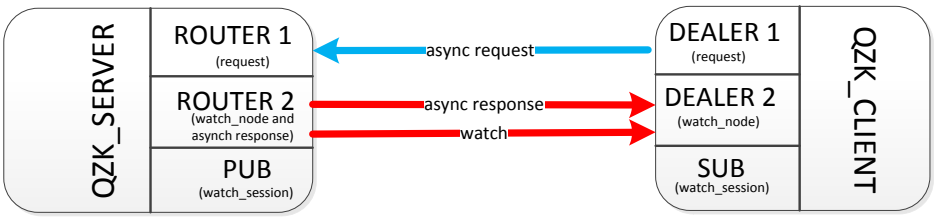



Figure 56 Asynchronous data flow

The qzk_client exposes a fluent-style API which produces a really easy to use and to read code. An example of its usage is:

```
zk.Create().ifNotExists().withMode(qzk::CreationMode::ephemeral).forPath("/node","value");
```

Let us suppose that **zk** is a qzk_client object. From it we can specify the operation that we want to execute and the different options. The last option is always "forPath". After it, the operation is executed.

The complete list of operations is the following:

| | |
|---|---|
| Create() | • **ifNotExists**: Creates the node if it did not exist. (If this option is not used and the node already existed, a NodeExistsException is thrown)<br>• **withMode**: Lets specify the creation mode (Sequential and/or Ephemeral)<br>• **forPath**(path,value) |
| Delete() | • **forPath**(path) |
| Exists() | • **setWatch**: Lets specify the callback watch<br>• **setNamedWatch**: Lets specify a reentrant watch |

| | |
|---|---|
| Get() | • **getState:** Lets specify a reference to a Stat structure, where the information of a znode will be stored.<br>• **setWatch + setNamedWatch** |
| Set() | • **withVersion**: Sets the data only if the znode version is the specified. Otherwise throws a BadVersionException<br>• **getState + forPath** |
| GetChildren() | • **getState:** gets the state of the father znode<br>• **setWatch + setNamedWatch + forPath** |
| Session() | Retrieves the session state |
| Transaction() | Let us execute atomic transactions |
| disableWatch(id) | Cancel the watch specified by id |
| disableAllWatches | Cancel all the watches specified until now. |

Although the qzk_server and qzk_client communication flow allows to implement asynchronous operations, we have not exposed this ability in the API. Hence, all the operations are synchronous (blocking). This means that if there is more than one actor executing in the same thread, special care has to be taken.

The API also offers some possibilities that Zookeeper does not provide, and that we think they make it easy to implement algorithms:

### 7.5.1 Transactions

Although we have not explained it, zookeeper offers the possibility of grouping operations and perform them atomically, that is, all or none of them are executed. However, it is quite inconvenient to use them with the C API. The QZK API provides a fast way to create transactions, as shown in the following example:

```
zk.Transaction()  <<qzk::txCreate().forPath("/a","value1")
                  <<qzk::txDelete().forPath("/b")
                  <<qzk::tx::end;
```

The qzk::tx::end modifier commits the transaction. Not all the operations can be performed in a transaction, just the Create, Set, Exists and Delete.

### 7.5.2 Errors as exceptions

The C API uses error codes to indicate errors in the operations. However, C++ provides a better method to handle errors: Exceptions. Although not everyone thinks that exceptions are the best way to go[34]–[36], the original Java API uses them and we have adopted the same methodology. For us, exceptions are self-explanatory. Moreover, in Zookeeper code exceptions are really important, they should always be handled by the programmer when they occur, since the correctness of the abstraction implemented depends entirely on how do we deal with exceptions.

From our experience developing with Zookeeper, any unhandled exception is a potential design flaw which could derive in a misbehavior of the algorithm.

### 7.5.3 Watches

Using the C API within a C++ program was very limiting, since C does not have the notion of objects. The QZMQ core is very flexible due to the fact that we can execute any kind of std::function as an event handler. In zookeeper watches are callbacks and should be considered as the same kind of events. Hence QZK provides the flexibility of std::function to be used as watch callbacks.

Moreover, the C API executes the callbacks in a different thread. However, in our actor world, everything has to be executed in a single thread. Hence, **watches are executed in the same thread where the actor lives.** This way we do not need to synchronize anything, and many threading problems automatically disappear. In fact, this depends entirely on how qzk_client is implemented. Our qzk_client maintains a map of watch identifiers and std::function. When it receives the watch message from DEALER 2, it looks the map for the identifier and executes the std::function. Of course the implementation does more than this. For instance, if the watch reached its last execution (remember that watches also have a life-cycle: section 0) the entry is removed from the map.

### 7.5.4 Disabling watches

Watches are single-shot events, which **cannot be cancelled**! We would like to cancel watches when, for example, we know that a specific watch will never be triggered again or when we are not longer interested in that event. However, zookeeper does not allow to do it.

In QZK, every watch is uniquely identified (remember that watches are messages in QZK). Using that identifier, the user can call **disableWatch(watch_id)**. QZK will silently ignore the watch when it is triggered by Zookeeper (that is, the callback specified will not be triggered). Without the ability of disabling watches, if the application is no longer interested in the side-effects of the callback executed by that watch, then that information has to be inside the same callback. This complicates things: How can a callback know that it has to do nothing? It is easier to let the main algorithm control these kind of things.

### 7.5.5 Named watches

Zookeeper watches are not reentrant. This means that if an application sets a watch two times on the same znode, the callback will be executed twice. This behavior is problematic (in fact, we do not know how this behavior can be useful). Hence, QZK offers the ability to use named watches using the function **setNamedWatch(std::function callback, std::string name ).**

Using it, if the application sets two watches on the same node, only the first callback will be executed. This makes named watches reentrant.

## 7.6 Introducing State Machines

The algorithm that we are going to show in section 0 has been implemented using the tools explained until now. The implementation is not elegant, and is probably buggy. In fact, it is too difficult to explain how is it implemented. It uses a "sort of state machine" using switch statements, callbacks and enumerations that emulate states (which is the worst implementation of state machines, as it is explained in Chapter 3 of [32]).

We will use a state machine implementation similar to the one found in the Qt State Machine Framework. The QStateMachine in the main object. It is possible to add states, compound states and transitions between states. We can think of an QStateMachine object as an actor with a unique queue where all the events (messages) are stored. A main loop reads each event one by one and triggers the right state transition.

In order to use QZK with a State Machine, we would have to change how messages are sent: All the state notifications, watches and asynchronous operation have to be thought as events sent to the State Machine queue. Moreover, code in a state can **post events** to the queue. A posted event is sent to the state machine queue and is further processed but the event loop (see Figure 57).

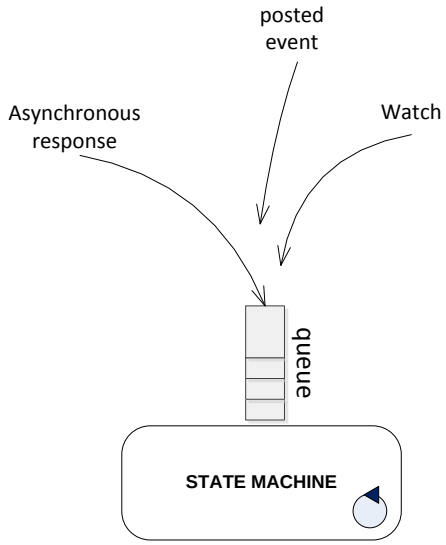



Figure 57 State Machine

Since the state machine is going to run synchronous operations, it should run it its own thread.

Moreover, we would like to give priority to posted events. That is, if a state posts an event and all the other queued events are not posted events, but zookeeper events, the posted event will be the first executed. Although this is not mandatory, it will help us to define watch states in an easier way. Although ZeroMQ does not allow to have message with higher priorities than others, it should not be too difficult to create a priority socket abstraction. The Qt framework, for instance, allows us to give priority to the QEvent objects.

After this short introduction, we will start introducing Zookeeper into the state machine world.

### 7.6.1 zkState

As we have said, the qzk_server is an actor that automatically handles the reconnection and session recovery. It emits updates through a publisher socket. We will suppose that these events are stored in the state machine queue in the form of **zkState** events. These events, which notify the state of the session, can have the following types:

- **CONNECTING**
- **CONNECTED**
- **SESSION_RECOVERED**
- **SESSION_EXPIRED**

The qzk_server has two different startup modes, shown in

- **Try to recover**: If a clientid_t is stored on disk, it will try to recover the session. This is the flow of events that it can originate and its meaning:
- **Not try to recover**: The client does not try to recover. However, it stores the session on disk, since the application might need to recover it in the next execution.

The life cycle of the session is as follows:

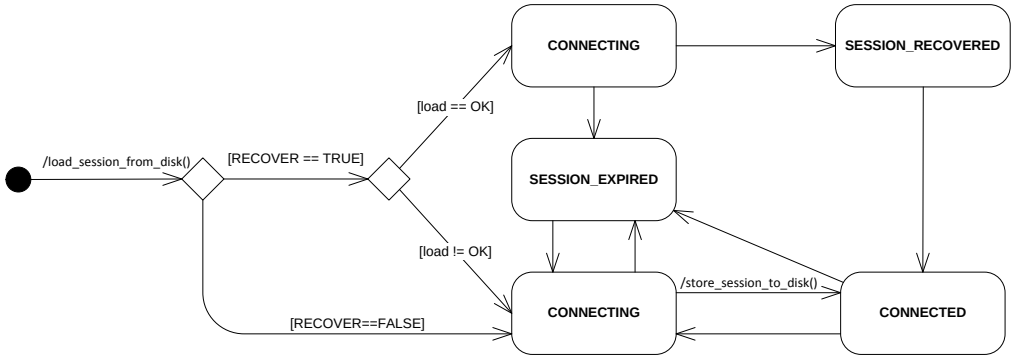



Figure 58 qzk_server session lifecycle

As we see, when an unrecoverable zookeeper error happens (session_expired), the actor is able of restarting a new session. In [29] is stated that

> "Libraries that recover from unrecoverable errors should be used with extreme care, if used at all."

This assertion attacks directly our proposal. However, they assume that the library is going to be used in a typical multithreaded application, where variables are shared between threads. However, in our framework there are no shared objects. All the abstraction that we will construct will use messages (events) to communicate with each other. Remember that messages are received sequentially, and are handled one by one. This will save us a lot of problems!

An important fact is that the life cycle does not consider the CLOSE state that the zookeeper handle has. We will consider that the zookeeper handle is closed when the entire application is shut down, and this is why we do not consider it in the life-cycle. Other components will never see the CLOSE state.[28]

[28] In an event-driven and/or actor framework, it is very important to define the shut down process. In this thesis we are not going to deal with it.

### 7.6.2 zkWatch

As we have seen, watches are executed multiple times, since they also carry session information. However, this information is redundant with the one provided by zkState. When the session is expired, the zookeeper library will execute both the global watch (which emits zkState events) and all the watches. It is enough to process the information once.

Although in zookeeper watches trigger the execution of a callback, we want to see them as events that trigger transitions of the state machine. Hence, watches have to be considered as events that are also sent to the state machine's queue. Since we can have more than one watch on different nodes, we will always name each watch that we set in order to be able to differentiate them. Each named watch (which is reentrant, as explained in section 7.5.5), generates events of type "zkWatch(name).Operation" , where Operation can be: Created, Deleted, Changed, Child and NotWatching.

When using watches in state machines, we do not need to specify a callback. Watches will only post events to our state machine, so the callbacks can directly appear in the state machine diagram.

An example will clarify what do we mean. For example, let us suppose that our application wants to be notified when the znode "/a" exists via a user defined "IT_EXISTS" event (a user defined event can be seen as a message published through a publisher ZeroMQ socket). In order to do this, we can create an actor which state machine is:

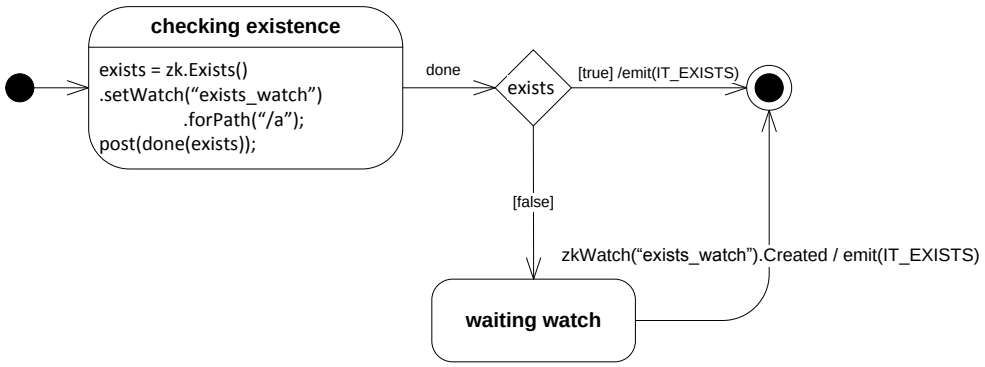


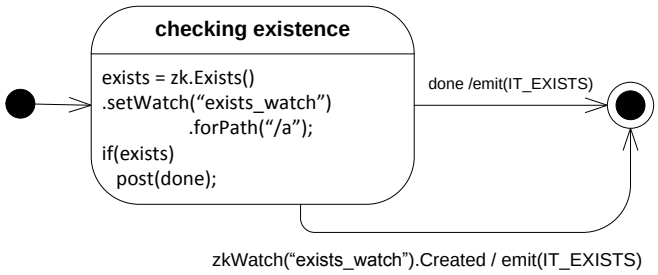

Figure 59 Exists notifier

From this example we can see some of the common problems setting watches:

- We need to set the watch at the same time that we execute an operation. However, many times we are only interested in the watch if a condition is false (in this example, that the node does not exist). If it exists, we are not longer interested in the watch. However, there is no way to cancel watches, so it will be executed in a future (for instance to notify a zkWatch.Deleted event). This is not a problem in a state machine because events received in incorrect states are silently ignored by the machine. Hence, if we receive a zkWatch.Deleted event when we are in the "checking existence" will not trigger any transition.

  **It is also important to remember that posted events have more priority than Zookeeper events**. This is crucial in our model. In a state machine, each transition is triggered by an event. At the end of the "checking existence" state, we post a done event (which will be used many times to force a transition between events). Let us suppose that the watch is triggered between the zk.Exists() operation and the post operation. It would mean that the watch would be executed before the done event, and **the watch would be silently ignored by the state machine**.

  It would be possible to model the same functionality without supposing any kind of priority between events:




  However, with this approach we have lost visual expressiveness. Now everything is compacted in the same state, but it is not so obvious that we are waiting for a watch, so we will prefer the first option.[29]

- We may think that a good solution is to first check if the node exists (without setting a watch), and if it does not exist, set the watch. However, the only way to set the watch is to execute the zk.Exists() operation again, so it makes no sense to execute it twice! Let us suppose that we do it:

[29] Of course, the actual way of expressing Zookeeper recipes will depend on the implementation of both the state machine and the Zookeeper interface.

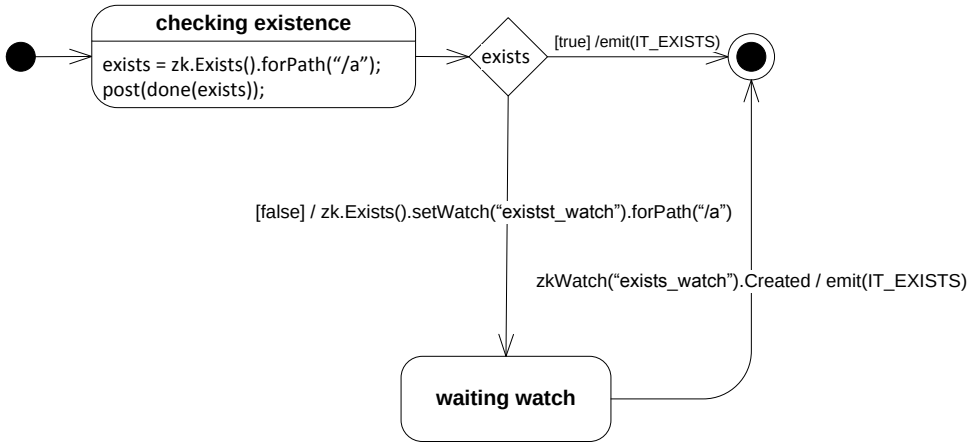



Figure 60 Incorrect usage of watches

Then, the following subtle error could happen: Let us suppose that exists=false. Just before the transition between "checking existence" and "waiting watch", a different client creates the znode "/a". In the transition we set the watch, but we do not realize that now it exists, so we will wait forever in the "waiting watch" state.

Dealing with watches is not easy. For example, in the example above we have forgotten an important event (which almost never happens and that is not considered in the recipes: The zkWatch.NotWatching event. It happens when the zookeeper servers run out of memory, and try to recover it removing some of the watches. Hence, a state machine that considers this event is:

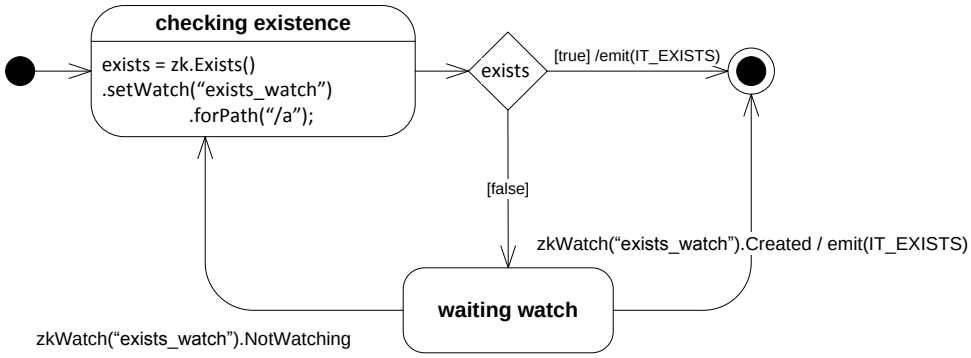



Figure 61 Exists notifier handling all the watch events

### 7.6.3 zkError

In zookeeper, when an operation is executed, it returns an error code. In QZK, operations return exceptions. Exceptions are good because if they remain

uncaught, an upper layer can catch them and handle them. Moreover, exceptions stop the execution path if they are not caught.

If we remember, there were 3 types of exceptions in zookeeper:

- **Normal state exceptions**: These exceptions go with the logic of the application. The user must handle them in the code executed in the state: Some of the common errors that the application must deal with are:
    - zkError.NoNode: The node does not exist (Obtained while trying to delete)
    - zkError.NodeExists: The node already exists (Obtained while creating a znode)
    - zkError.NotEmpty: We were trying to delete a znode that has children. The delete operation is NOT recursive.
- **Recoverable** and **Fatal** exceptions: We want to be able to model them as transitions between states. The main exception that we want to handle are:
    - zkError.ConnectionLost
    - zkError.OperationTimeout
    - zkError.SessionMoved
    - zkError.InvalidState: It means that either the session has expired, or that the client is not authorized to do the operation. Since we are not considering authentication at all, we will consider that zkError.InvalidState as zkError.SessionExpired
    - zkError.Closing: Someone has closed the zookeeper handle. We will not consider this exception, since the qzk_server actor maintains always a valid handle (In Figure 58 there is no CLOSE state)

As a convenience, when we write zkError.ConnectionLost we will refer to both the ConnectionLost and the OperationTimeout, since they have the same semantics (as seen in section 0).

Moreover, **zkError.Recoverable** will group ConnectionLost, OperationTimeout and SessionMoved.

#### 7.6.3.1 *From exceptions to transitions*

In order to understand how to do it, we need to put an example of a real state machine framework. We will use the Qt State Machine Framework. If the reader is not familiar with it, it can find more information in [33].

This framework provides the QState class, which represents a state. This class inherits from QAbstractState, which provides two protected methods that can be implemented:

- onEntry: Execute code when the state is entered
- onExit: Executes code when the state is exited

In our state machine diagrams, we are supposing that the code is executed in the onEntry method. The idea is to subclass QState creating a new class called QStateZk, which provides the virtual methods:

- onEntryZk: Execute code when the state is entered
- onExitZk: Execute code when the state is entered

Then, a possible onEntry implementation can be:

```
virtual void onEntry()
{
        try{
                onEntryZk();
        catch(zkError.ConnectionLost e){
                machine.postEvent(zkError.ConnectionLost)
        }catch(zkError.ConnectionTimeout){
                Machine.postEvent(zkError.ConnectionLost)
        }
        /*other recoverable exceptions...*/
        catch(zkError.SessionExpired){
                //Do nothing
        }
}
```

All the recoverable exceptions are caught and rethrown as events to the state machine. However, the SessionExpired error is not translated into an event! We do not need it, since a zkState.SESSION_EXPIRED will be triggered.

## 7.7 Expressing error transitions: Using hierarchical state machines.

Until now we have not tried to express the error transitions (zkEvent and zkError) in our state machines. If we try to do it, we would realize that each state that executes a zookeeper operation needs to handle ALL the possible errors. This is very inconvenient. Using hierarchical state machines we can group states that have

to deal with an error in the same way. For example, let us suppose that we want to create a state machine that create the znode "/a". We may think that this is enough:

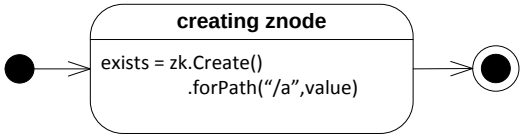



Figure 62 znode creator version 1

However, what happens if the connection is lost? Then we are not sure if the znode has been created or not. We can fix this with the following algorithm:

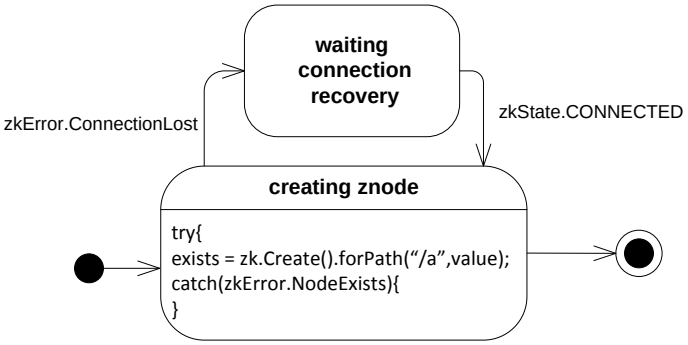



Figure 63 znode creation version 2

Other frameworks like Curator use a retry mechanism. However, why do we want to retry an operation if the connection is lost? The best thing that we can do is to wait until the connection is recovered again!

However, what happens if the session expires? Then we also have to start again. The zk.SESSION_EXPIRED event can happen in both states. Do we have to create transitions for the two states? This is really inconvenient, and the transition complexity would increase with the number of states. Hierarchical state machines let us create groups of states, so we can model the algorithm like this:

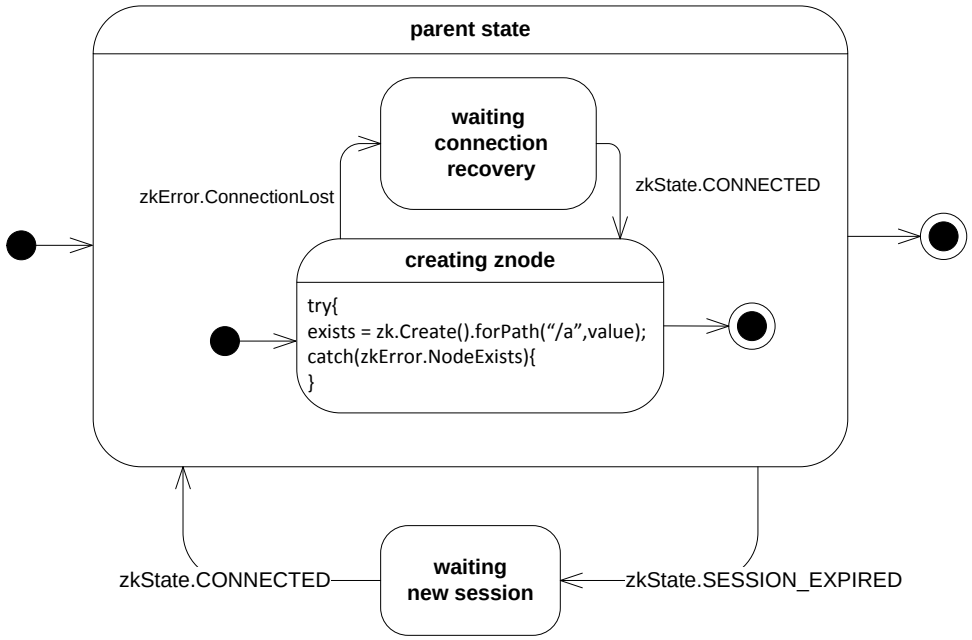



Figure 64 znode creation version 3

The parent state will process all the events that its children cannot process. Hence, we can model transitions in the parent, making the state chart concise and simple. If hierarchical state machines are new to the reader, a good introduction to them can be found in [32].

It is also important to know that a **finished** event is emitted by the parent state when there is a transition to the **final state.**

## 7.8 Recovering the session at runtime: Using history states

In the last section we have solved the zk.ConnectionLost error using a "waiting" state. From this state, when a zkState.CONNECTED event is received we go to the last state (creating znode). This notion of "going to the last state executed" is needed many times. This is usually modeling using **history states**, as shown in Figure 65:

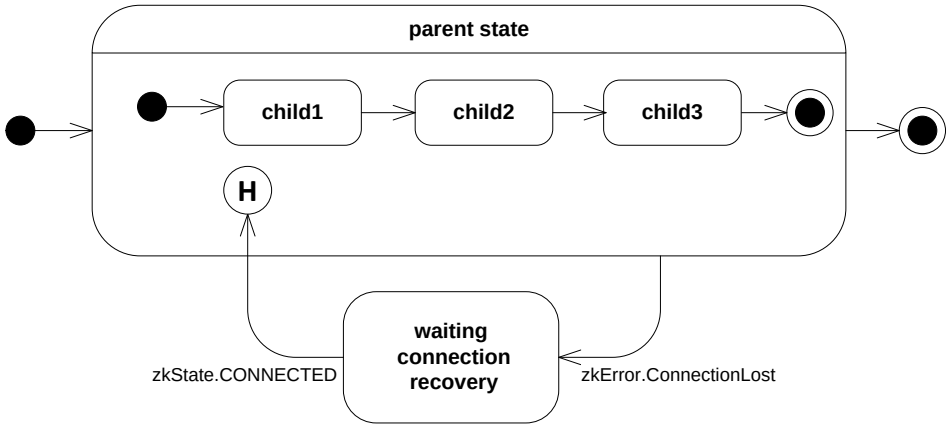



Figure 65 Using a history state

As soon as the state machine gets a little bit complex, there start to be a lot of states. It is possible to describe state machines using composite states. First, you define the composite state (as it is done in Figure 65). Then, you can reuse it in a different state chart using the following representation:


Figure 66 Composite state representation

A history state which remembers the inner state of composite states is referred as a **deep history** and is represented as H*.

## 7.9 Recovering session after a crash

Histories remember the state in a runtime data structure. When the process crashed and is reinitialized, the state machine starts from the beginning. However, as we saw in section 7.6.1 it is possible to let zookeeper try to recover the session. When Zookeeper recovers a session, the Leader server remembers all the watches set by that session, and triggers them sending messages to the Zookeeper clients. The client receives the message and triggers the proper callback. The callbacks are stored in memory, so when the application crashes and is later on initialized, and the zookeeper session is recovered successfully, all those callbacks are lost, which immediately translates into that **all the watches are lost.**

Hence, Zookeeper state machines cannot go always into the last state executed. If that state is waiting for a watch, the state machine will wait forever, since the watch will never be triggered! In section 0 we will explain how do we deal with this problem.

## 7.10 Final thoughts

Now we have all the tools needed to express zookeeper algorithms. The next session can be seen as the proof of concept of everything that we have developed until now. We will present a decentralized task distribution framework using zookeeper.

# 8 Task distribution framework

In this section we will describe a simple task distribution framework in the form of a zookeeper algorithm. The main characteristics of the algorithm are:

- Tasks are distributed in a collaborative way. There is no single master node.
- It is possible to monitor the resources and the tasks (which node is executing a given task)
- Fault tolerant: There is no single point of failure:
    - If a worker crashes and cannot recover within its session timeout, the task is executed by a different worker
    - If a worker crashes and can recover within its session timeout, the task is executed by him again. This lets the task implementer to be able to snapshot the task, and continue from the last snapshot[30].
- It is possible to implement authorization and authentication using Zookeeper ACL lists. However, we will not consider it.
- Tasks follow a FIFO order (they can only be batch tasks without priorities, see 0)
- Each instance of the state machine can be considered as a worker slot. Running multiple instances of the state machine in the same process we can easily get workers with multiple slots.

We will also explain how to solve more advanced problems with zookeeper, like how to reliably create ephemeral and sequential znodes in order to implement a distributed linked list or how to restore the session after a crash.

## 8.1 System architecture

As we saw in the previous sections, the system architecture of a task distribution framework is more or less always the same:

[30] Hoever, we do not provide any API to create the snapshot. It is up to the task.

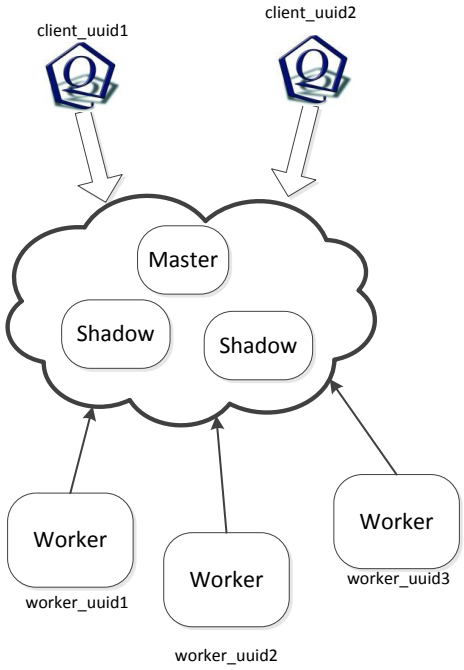



Figure 67 System architetcture proposal 1

In this architecture (Figure 67) the master controls the workers connected into the system, assigns the tasks to them and sends the output of the execution of the tasks back to the clients. In order to provide reliability, some Shadow nodes replicate the state of the Master in case it crashes. Our first approach was to use zookeeper to implement the distributed abstraction needed to implement such a system:

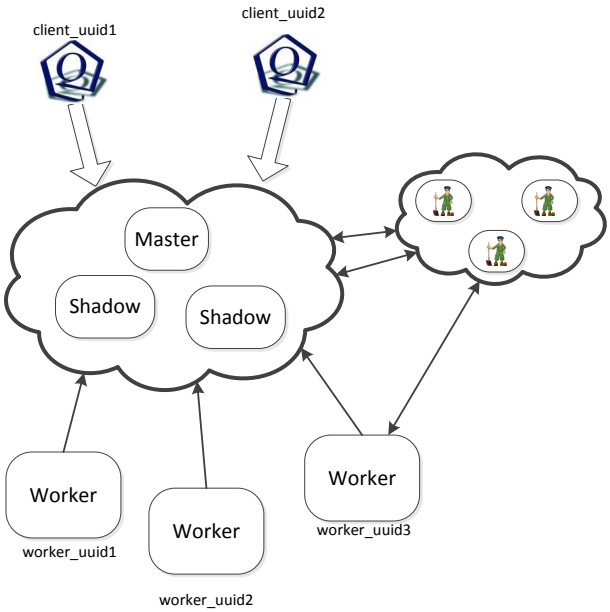



Figure 68 system architecture porposal 2

Failure detection of Workers or Master/Shadow nodes was fairly easy to implement using Ephermeal znodes. However, there was no direct solution to implement state replication between Master and shadows. Suddenly, we realized that we did not need to implement it using Zookeeper, because Zookeeper was itself a Master- shadow architecture, so we decided to change the architecture to the following one:

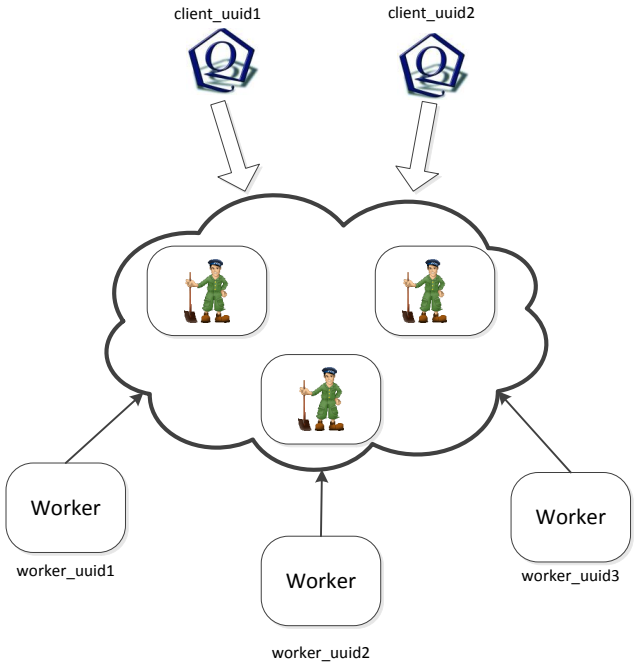



Figure 69 System architecture proposal 3

However, the absence of a master coordinator added a new problem to solve: How tasks are assigned to nodes? As we will see, the Workers help each other in order to assign tasks. Another problem that we had forgotten was the task transmission. CPLEX file descriptions (from now on, problem files) are big (from 10 to 200MB). The solutions of CPLEX algorithms are even bigger (from hundreds of MB to a few GB). However, the Zookeeper service was not thought to carry such amounts of data (remember that writes are slow), so we cannot store them in the znodes. This in turn implies that we need to implement a file transmission protocol between clients and workers. After some consideration on how to implement it using ZeroMQ, we decided not to proceed with this path. This solution would be overkill for our thesis, so we decided to continue in a different direction: Use a regular Network File System to store the task inputs and outputs:

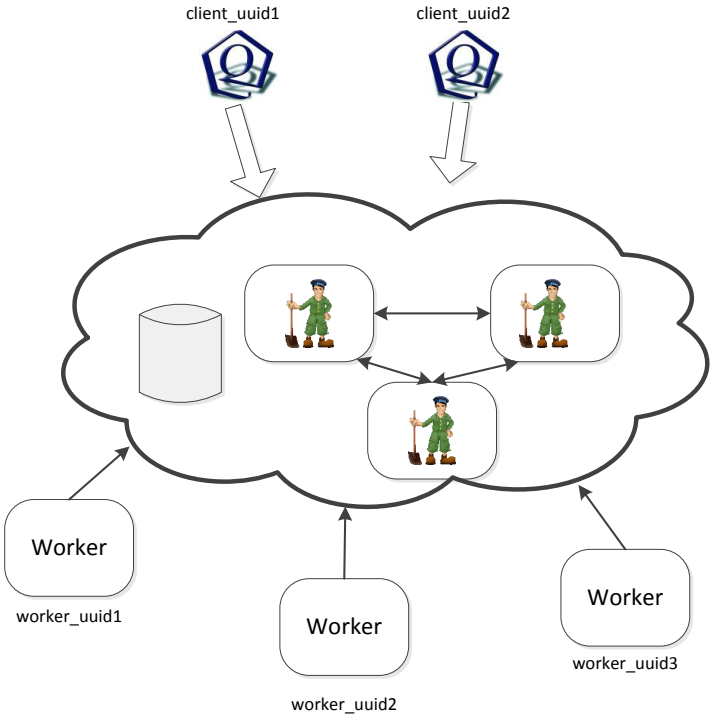



Figure 70 system architecture final proposal

Although this partially solved the problem, it is a patch that we have had to do to be able to finish on time. A proper solution would try to use its own Distributed File System (similar to what MapReduce techniques do, see section 4.2).

## 8.2 Public API

The idea is to simplify as much as possible the work that an application using the system has to do:

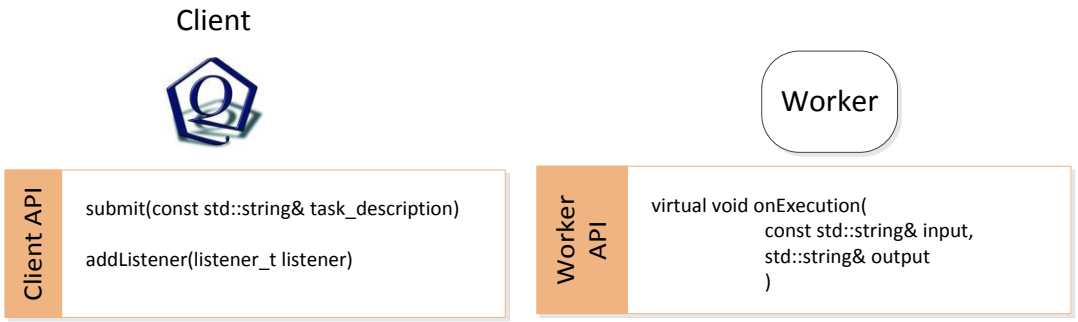



Figure 71 Client and worker API

In order to use a client, we only need to use two functions:

- Submit: submits a task into the system, which will be executed by a Worker
- addListener: Let us specify a callback that will be executed whenever a task has been executed.

The Worker is itself a class which can be inherited from. The onExecution method is executed each time that a new task has to be executed. The application user has to deserialize the task input from "input", do whatever is necessary, and serialize the output of the execution into "output". The problem with inheritance is that the type of work cannot be changed at run time. However, it is easy enough for our thesis purposes.

## 8.3 Znode structure

Developing a zookeeper algorithm implies two things. Thinking about the znode structure and how to modify it. In this section we will explain the first one. We will represent each znode as depicted in Figure 72 Znode representation.The znode hierarchy is depicted in Figure 73.

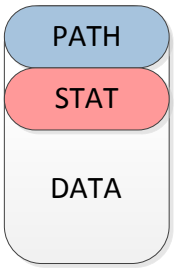



Figure 72 Znode representation

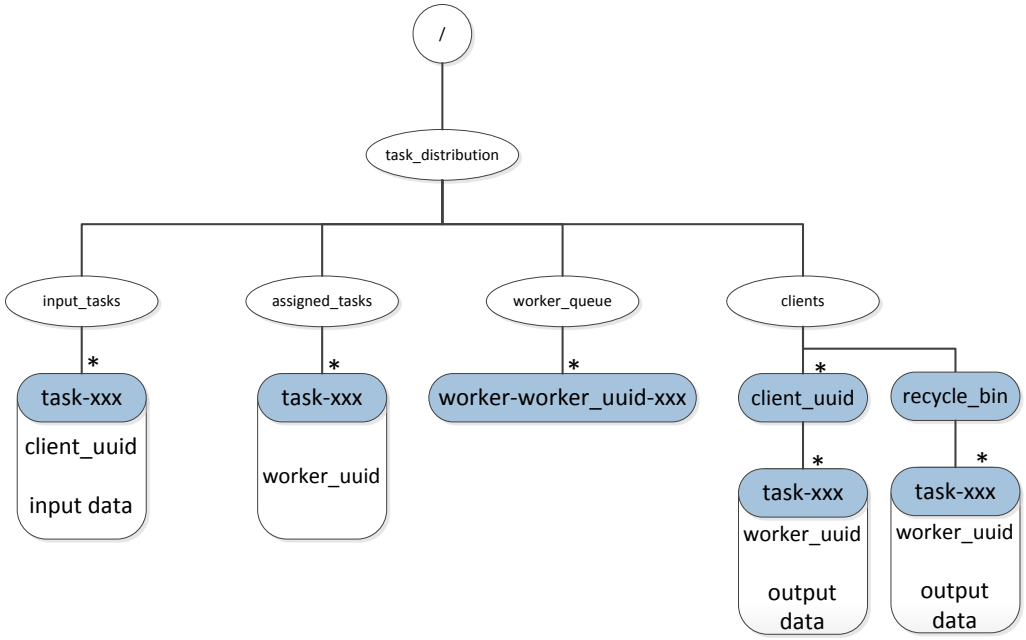



Figure 73 TDF znode structure

### 8.3.1 Representing data in znodes

Data in a znode is unstructured. We can choose whatever method that we want to structure it. We have heavily used Google Protobuffer in QZK and QZMQ to serialize the messages between different actors. There is no reason to not to use it again. The protocol descriptions are really easy:

```
message InputTask{
        optional bytes client_uuid =1;
        optional bytes data=2;
}
message AssignedTask{
        optional bytes worker_uuid =1;
}
message FinishedTask{
        optional bytes worker_uuid=1;
        optional bytes data=2;
}
```

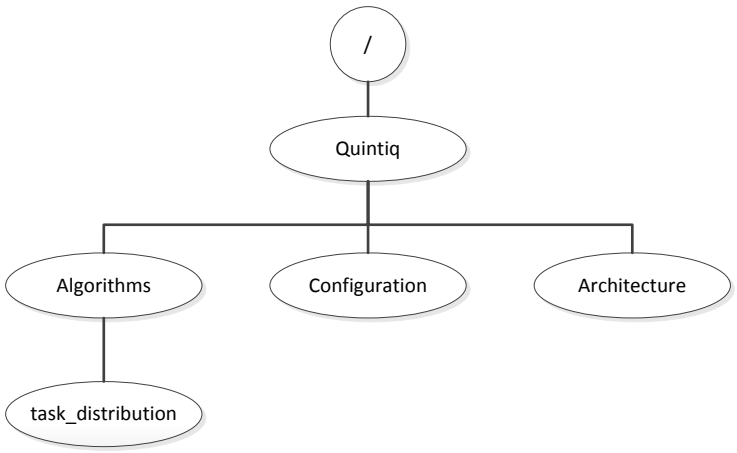
All the fields are optional, following the recommendation that can be found in [37].

- Each client is uniquely identified via its client_uuid. This client_uuid has to be generated from some centralized point, or be randomly generated once and used for the rest of the session.

- Each worker is uniquely identified via its worker_uuid. In our implementation it is a 16 byte number, referred as a Universally Unique Identifier.
- The input and output data is represented as an unstructured array of bytes. The application using the framework is the one that has to deal with its serialization. In our CPLEX example, the data is just the path of the input or output file inside the NFS.

### 8.3.2 The root znode

The root node of the entire hierarchy is **task_distribution**. One of the nicest options of Zookeeper, which was introduced in version 3.2.0, was the ability to run the client commands while interpreting all paths relative to a certain root znode. This way, we could have multiple different applications running on the same hierarchy. The structure of Figure 73 could be lately moved to the following structure without modifying any line of code:




However, we consider that this functionality is broken, at least in the version used in this thesis (v 3.4.3). For instance, if we set the root to be "/Quintiq/Algorithms", and we create "/a", the znode "/Quintiq/Algorithms/a" will be created. However, the return path will be also "/Quintiq/Algorithms/a" (the absolute path from the "/" root), which makes no sense for us.

### 8.3.3 Global explanation

The znode structure is divided into 4 branches:

- **Input_tasks:** Each child of this znode represents a submitted task. The name of each task is task-xxx, where xxx is a monotonically increasing sequence number (generated using zookeeper sequential creation mode). The

monotonicity makes the tasks to be executed in FIFO order. The data stored by the children includes the client_uuid, which stores the id of the client that submitted the task. Although this is not compulsory, it is really convenient for debugging or controlling purposes.

- **Assigned_tasks:** Each child is a task from the input_tasks branch that has been assigned to a worker. The name of these znodes is the same as the ones from the input_taks (in order to easily match them), and they contain the worker_uuid (in order to know which worker is executing the task).
- **Clients:** Each client that wants to receive a task output creates a znode with its client_uuid. Each of these znodes will act as a container of finished tasks. It is the client the one that has to retrieve and delete them. If a task has been executed by a Worker, but the worker cannot find the proper client_uuid znode, it will store the task in the recycle_bin.
- **Worker_queue:** The worker_queue acts as a **distributed linked list** of processes. The list is a FIFO list of Workers that are ready to execute a task (that is, that have empty slots). The first Worker in the list monitors the input_tasks and assigned_tasks branches, and if it is possible, self-assigns a task that is not being executed. After it has the task assigned, it leaves the queue. The second worker wakes up, realizes that it is the first in the queue, and starts monitoring the task branches.

## 8.4 Algorithm step by step

Probably this is the most important section of the thesis. We are going to show how does the task distribution framework works. We will use the tools developed in section 7.6. In order to facilitate even more the process of understanding it, we have created an interactive presentation explaining each part of the algorithm step by step [38]. We encourage the reader to take a look at it if he feels lost.

### 8.4.1 The client

Client operations are so easy that they do not need a complex explanation.

- **Submitting a task:**
  zk.Create().withMode(sequential).forPath("/input_tasks/task-",DATA)
- **Listening for finished tasks:** Let us suppose that our znode is already created. Then the only thing that has to be done is to set a child watch over it. We will be notified every time that a worker finished one of the submitted tasks.

There are some subtleties that we have not mentioned, which can be further explored in the interactive presentation.

### 8.4.2 The worker

Very quickly, the worker is a state machine that automatically connects with zookeeper and self-assigns a task at the correct moment. After the task is assigned, it executes the user provided **onExecution** method.

The worker deals with the following global state variables:

- **my_uuid:** Unique worker identifier. It is a 16 byte integer.
- **queue_id:** Id that the worker gets when entering the worker queue. The queue_id has the form: **worker-<my_uuid>-<sequential_number>**
  sequential_number is the sequence number generated by Zookeeper when creating a znode with sequential mode.
- **current_task:** Task that the worker is executing or is going to execute. It contains two parameters:
  - **task_id:** Uniquely identifies the task in the system. It has the form task-sequential_number
  - **client_id:** Id of the client who submitted the task
- **task_description:** It is the input data of the task (see Figure 73).
- **task_output:** The output of an executed task

In addition, there is a **step** variable that helps us to recover the state machine from a crash. Unluckily, UML state charts do not provide crash-resilient history pseudo states. In fact, defining its semantics would be very difficult. When dealing with Zookeeper, not all the states are recoverable, as we previously mentioned. It is the job of the algorithm designer to detect which are the recoverable states. We have identified the following steps:

- QUEUE_ENTERED: We have created a znode in the worker queue. The next state that has to be executed is "finding predecessor".
- ASSIGNING_TASK: We have detected that we are the first worker in the queue. The next state that has to be executed is "assigning_task".
- I_HAVE_TASK: We have assigned the task (that is, the znode in the assigned_tasks branch is created). However, we have not exited the queue. The next step is "exiting queue"

- TASK_ASSIGNED: Task assigned and exited from the queue. The next step is "executing_task".
- EXECUTED: The task has been executed, but not submitted. This is one of the most important steps. If tasks take long to execute, and after the execution the node crashes, then it would we a waste of time to recover and to have to execute the task again. The next step is "submitting task".
- FINISHED: The task has been submitted, and we can enter the queue again. The next state is "waiting_in_queue".

There are functions that we can call in order to manipulate the variables:

- Store_state(step): It lets us store all the variables on disk, specifying in which step are we.
- Load_state(): It loads all the global variables from disk. If there is no state stored, it is initialized.
- Initialize_state(): It initializes all the global variables. The step is initialized as FINISHED.

#### 8.4.2.1 *Initialization phase*

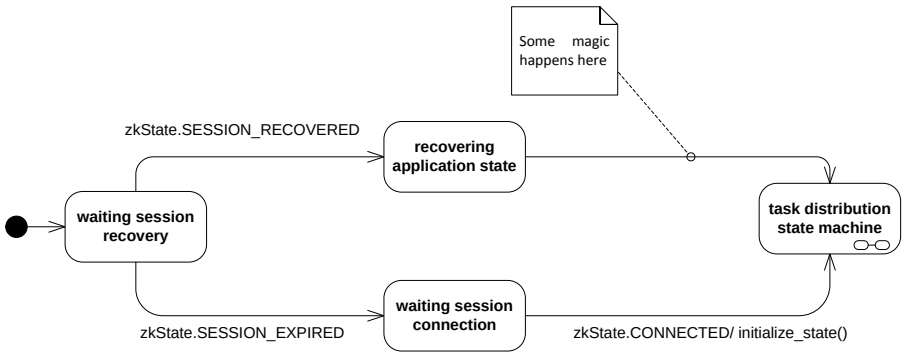



Figure 74 Initialization phase

Here we see that if the session cannot be recovered, then the state is initialized and the task distribution state machine state is entered. However, if the session is recovered, we enter a "magic" recovering application state. This state:

- Loads the state of the variables from disk.
- POSTS to the state machine the step variable.

Although we will not show all the transitions in Figure 74, the following diagrams will show them. All the transitions related to the recovery of the session after a crash will be depicted with a thicker line than the rest.

### 8.4.2.2 *Task distribution state machine. An external view*

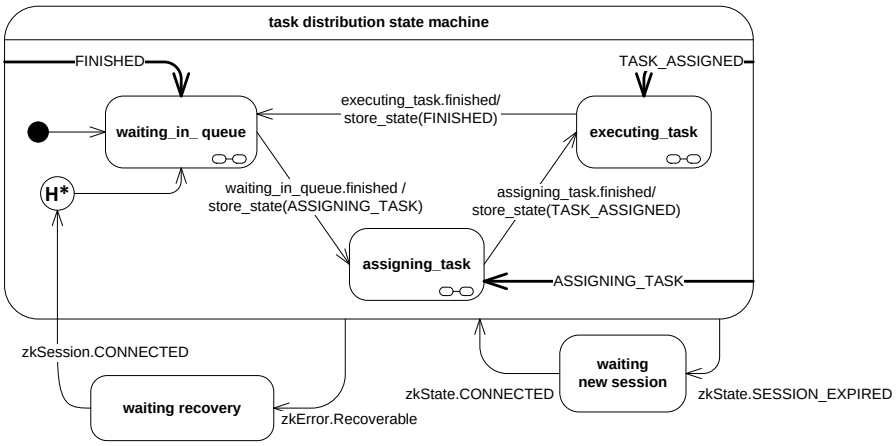



Figure 75 Global view of the algorithm

The figure above shows the task distribution state machine state. This is the root of the state machine. It shows that there are mainly 3 steps in the algorithm: The queue step, the assigning step and the executing step. It also shows how do we handle the session expiration and the recovery phase of recoverable errors using a deep history pseudo state. By default, if there was no previous state executed, a transition to the history creates a transition to the "waiting_in_queue" state.

### 8.4.2.3 *Waiting in queue state*

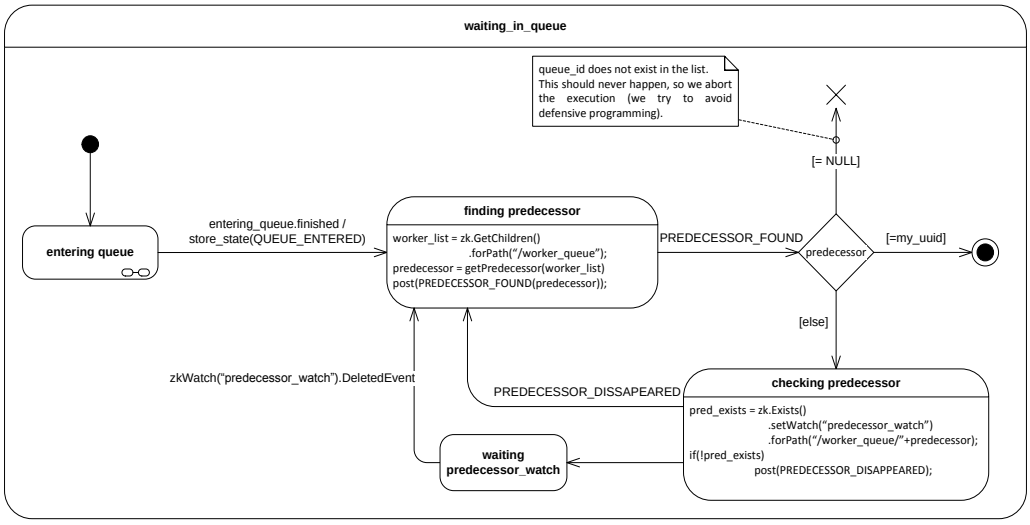



Figure 76 Waiting in queue state

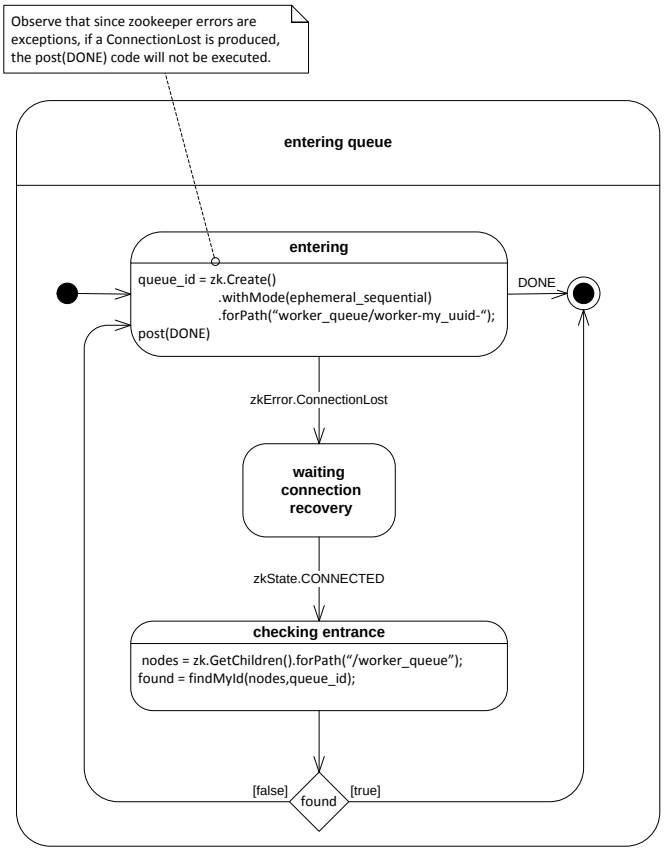



Figure 77 entering queue state

The entering queue state uses a similar method as the one explained in section 7.7. However, in this case we are creating sequential nodes. Since there can exists just one znode per worker in the queue, special care has to be taken (see the description of the function **findMyId** in the following figure):

findMyId(List<String>, String&)

We cannot be sure if the create operation succeeded or not. Hence, we include in the znode name our own id.
Each znode name will have the form
worker-worker_uuid-list_sequence_number
This function has to try to find in the List<String> a String with my_uuid. If it can find it, it will mean that the operation was executed in the zookeeper servers

Figure 78 findMyId function

The worker queue is in reality a distributed linked list of processes. Each process only knows about its predecessor, and is notified when it disappears (it has assigned a task or it has crashed).

Please now observe the code in finding predecessor state. It gets the list of children in the queue and calls getPredecessor, which is a function that:

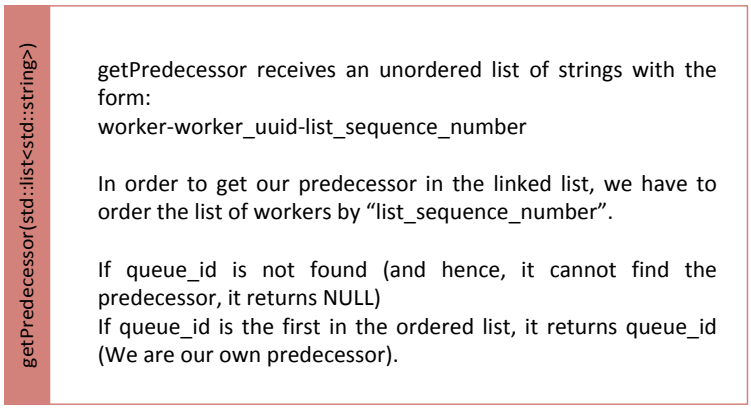
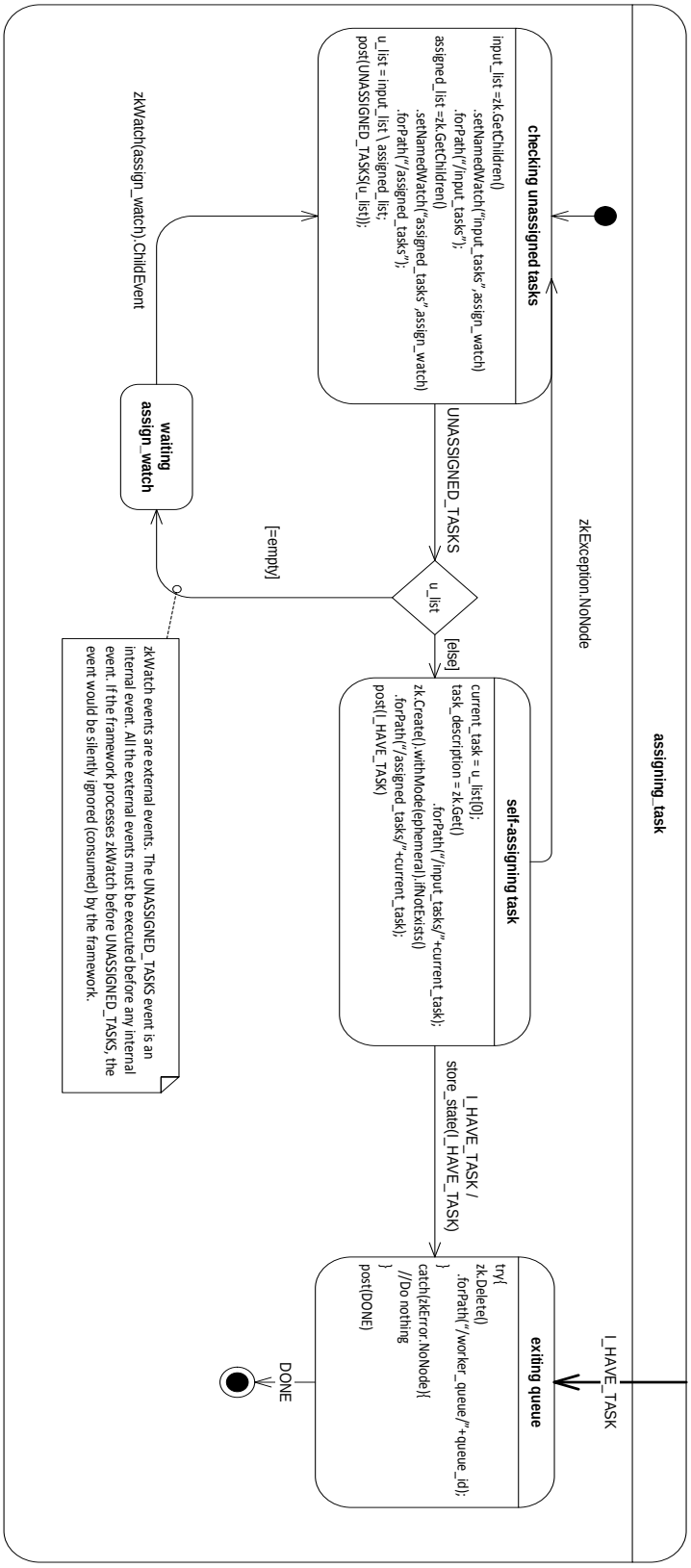



**Figure 79 getPredecessor function**

The getPredecessor should never return NULL. The only way that queue_id cannot appear on the list is because we have lost the session. If that was the case, getPredecessor would never be executed, since the state machine will move to the "waiting new session" state (see Figure 75).

### 8.4.2.4 *Assigning task state*



The important thing to consider here is how do we detect that there is a new task in the system. Zookeeper does not give us the option to get the last created child. Hence, every time that a new task is created, we have to take the entire input_task children list and compare it with the assigned_task. If we treat the lists as sets of elements, u_list (unassigned_list) contains a list of unassigned tasks.

Many things may happen to a task:

- A new task is submitted by a client in input_task
- The task has been executed and hence deleted from both the input_task and the assigned_task znode
- A worker crashes while executing a task. Since the children of the assigned_task znode are ephemeral, the assigned_task will disappear. A different worker has to execute it again.

If the system detects that there are no unassigned_tasks, it has to wait until something happens. However, Zookeeper Child Watches do not give us too much information. They just notify that something has happened to the children. This is very inconvenient! Every time that we receive a watch, we have to get the entire list of input_tasks and assigned_tasks. In a very busy environment with lots of tasks, this fact would create a scalability problem. **Our system can work in small-medium architectures, but not in very large systems,** since the message payload increases with the number of input tasks and the number of workers**.** In fact, we can detect the same problem while trying to find our predecessor in the worker_queue.

### 8.4.2.5 *Executing the task*

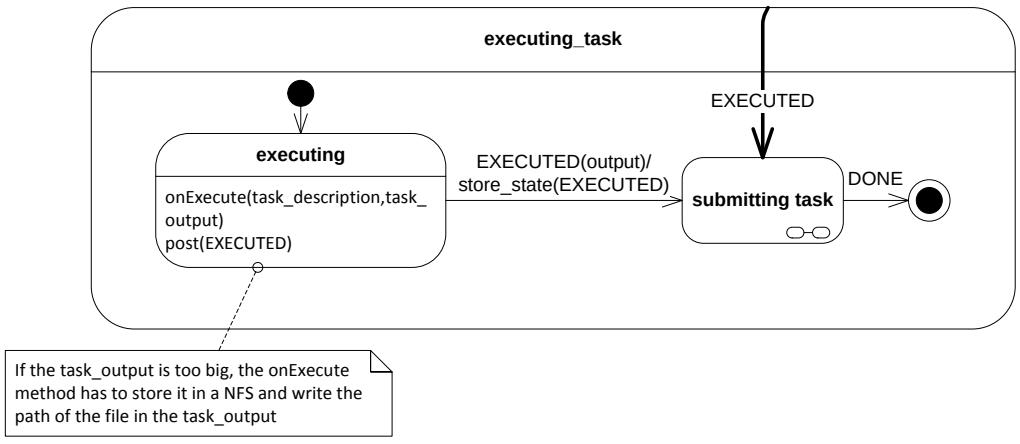



Figure 80 Executing task state

The most important detail is how do we have to submit the task. We have to do three different things:

- A: Delete the task from the input_task list
- B: Delete the task from the assigned_task list
- C: Submit the task in the proper client queue

As always, Zookeeper can fail at any state. Moreover, the computer may crash also at any moment. If we execute these operations one by one, and the system fails, the system would remain in an inconsistent state. For instance, if a node executed A, B but before executing C it crashed, then the task will never be submitted (the output is lost and the task is not re-executed).

If we remember, Zookeeper allows us to execute operations in a transaction. This is exactly what we need to do in this case:

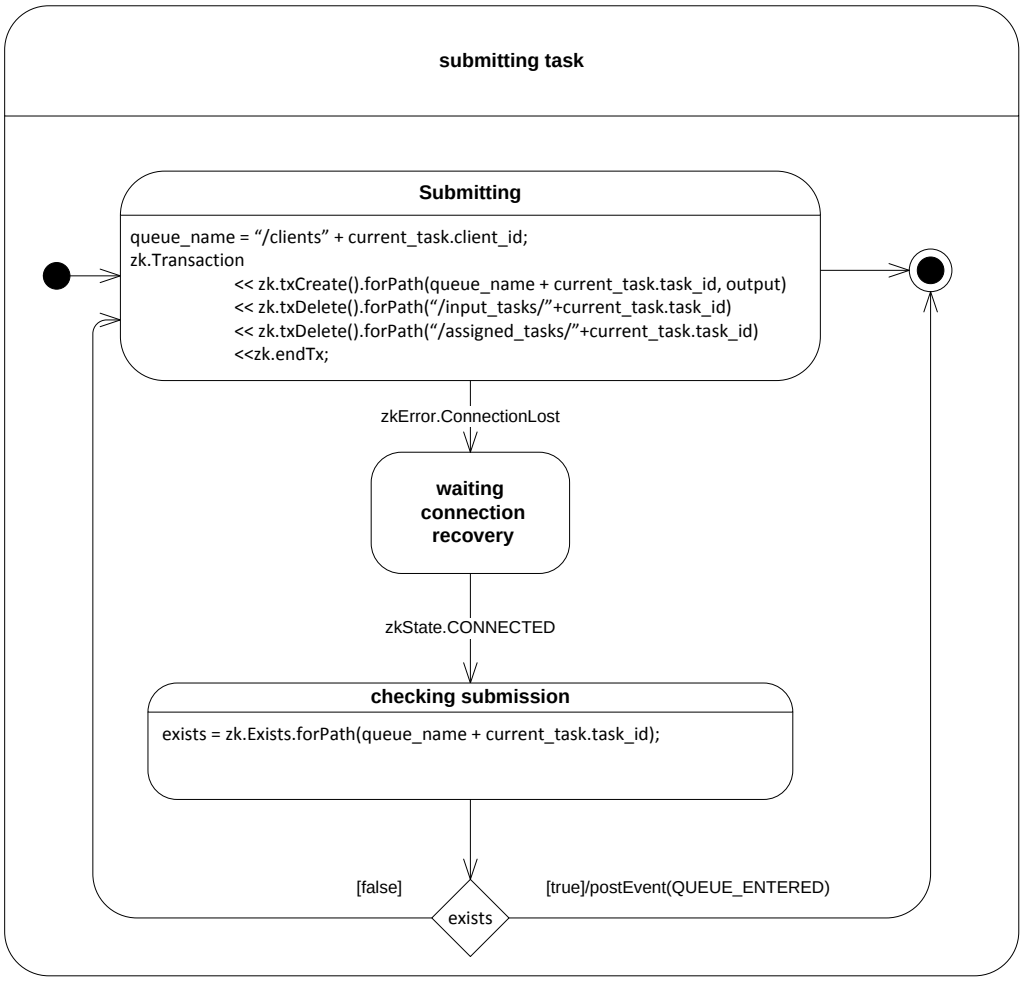



Figure 81 submitting a task using transactions

## 8.5 Extending the algorithm

The algorithm showed in the last sections is just an example of what can be achieved using zookeeper. Although many Zookeeper recipes implement abstractions like Failure Detection, Leader election or distributed barriers, we have shown that it is possible to build more complex systems.

Moreover, with the right tools, it is quite easy to extend the functionalities of the system, the only thing that we have to do is to design a more complex znode structure and glue it with a state machine design. Possible extensions are:

- Worker with multiple slots: Each slot is represented by an instance of a state machine. Hence, if we want a worker to have N slots, we just have to run N instances of the state machine (with different worker_id).
- Task priority: Tasks are ordered by name. Right now their name **is task-sequence_number**. If we change the name to **task-priority-sequence_number,** workers can order them first by priority and then by sequence number. However, due to Zookeeper's nature, this is not an efficient solution.
- Create different kinds of task: There is nothing stopping us from duplicating the znode structure and assign to it a different kind of task. Worker nodes should subscribe to the proper worker_queue. In fact, a single worker could subscribe to different worker_queues, simulating the behavior of queues in the grid engine (see Figure 18).

Another possibility that we would really like to explore is the ability of allowing asynchronous communication between clients and workers. The client should be able to at least stop the execution, and the worker should be able to at least communicate the progress of the execution.

# 9 Conclusion & Overview

The main goal of the thesis, designing and implementing a task distribution framework, has proved to be an excuse to find and learn technologies to implement distributed systems in a more general way. We think that the main contribution of our work is the proposal to design and develop Zookeeper algorithms (recipes) using hierarchical state machines and the automatic session recovery of Zookeeper session.

Recipes explained in Zookeeper's paper or found over the internet are always incomplete, since they do not handle all the possible corner cases. Some other recipes, like the ones found in the Curator framework are not easy to understand. There is no common theoretical framework to represent such an algorithm, and we believe that our state machine approach is a great step towards full fault-tolerance algorithms using Zookeeper. It allows to express and handle groups of exceptions in different levels in the hierarchy.

Zookeeper offers a really easy to use API with few operations. However, we find that it is fairly difficult to fully understand how to use it and/or devise which consequences and implications the errors and operations have. Zookeeper's documentation is not as good as it could be, so we have devoted a big part of our time to extend the explanations that can be found in the Zookeeper's paper, making sure that all the concepts are understood even by newcomer, treating topics like linearizability from a practical point of view.

In order to link the Zookeeper and the state machine world, we have also developed QZK, a self-healing zookeeper actor which communicates both synchronously and asynchronously via message passing with other actors. In fact, a hierarchical state machine can be thought as an actor that receive events from itself and from the outside, and process them sequentially. Those events can be received from one or more than one message queue with different priorities. A Zookeeper state machine is, hence, a state machine that receives and sends events to the QZK actor and handles them properly. Although we have not been able to fully implement all the tools needed to develop the final task distribution framework (we have had no time to implement the hierarchical state machines), we have shown that we were on the right track.

The base cores needed to implement state machines and actors in sequential programming languages like C++ are event-driven frameworks and signal-slot mechanisms. ZeroMQ is a cross-platform messaging library which plays nicely with asynchronous operations, and its ability to use it with different transports allows us to create both multithreaded, multiprocess and multimachine

programs seamlessly. Using it, we have developed our own event-driven and actor library, QZMQ. Although QZMQ is far from its completion, we have successfully created services like RPC Google Protobuffer servers and clients with a few lines of code using the QZMQ actor facilities. We believe that a proper actor framework using ZeroMQ has the potential to be even more flexible than other actor implementations like Erlang (ZeroMQ let us connect actors in a great variety of topologies, whereas Erlang implements only one-to-one communication).

While developing QZMQ, we have also studied the Quintiq software architecture in order to discover their main weak points and ways to improve it. We realized that a good starting point would be to distribute algorithms in a computer cluster, and we have studied some of the most used task distribution engines. After studying them, we realized that it would be impossible to develop all the needed services in 3 months. After a short survey of the common messaging middleware, we discovered Zookeeper, which we finally used to design the task distribution framework.

After having developed the Zookeeper C++ binding and the ZeroMQ high level library from scratch, we think that they have a great potential and that it would be very interesting to integrate them in frameworks like Qt. Its signal-slot mechanism together with its mature event loop implementation and the State Machine Framework would perfectly work with our proposal of Zookeeper hierarchical state machines. We think that it has the potential to be an easy to use framework for fast prototyping of distributed systems.

The structure of the thesis shows the evolution of our work, going from the general to the specific and explaining all the mental processes that have taken us from the first idea to the final design (and for this reason, it might seem a little bit chaotic). In fact, the idea of using hierarchical extended state machines in order to design Zookeeper algorithms appeared one month before finishing the thesis, and is the product of hundreds of hours trying to understand Zookeeper and trying to develop reliable recipes with it.

We believe that science has to be fun and inspiring, and documents explaining it should be also like this. We hope that the reader has enjoyed reading it as much as we have enjoyed the whole process.

# Appendix

## Becoming a C++ demigod in 4 months

One of the main challenges of this thesis has been to learn to properly develop with C++ in a very short period of time. In this section we will summarize all the steps that we have had to follow to do it, providing all the sources from which we have learnt. As we will see, it is not enough to learn just the syntax.

### Learning C++

C++ is considered to be one of the most complex languages to master. It is thought as a big Swiss army knife in the sense that it contains many different "programming flavors":

- **C flavor:** C++ shares many basic elements with the C programming language. Here we can find the basic types, pointers, references and pointer functions, structs and unions, for example. It is mainly used to interface with old libraries or to do low-level operations.
- **Object Oriented flavor:** It includes the concept of classes, inheritance, multiple inheritance, polymorphism, etc. The usage of classes introduces new programming idioms like Resource Acquisition Is Initialization (RAII), which handles resource allocation and deallocation automatically.
- **Generic programming:** At the beginning it was thought to allow the programmer to create containers of "things". The principal exponent of generic programming is the Standard Template Library (STL). However, it was discovered that the template system was even more powerful, thing that introduces another flavor:
- **Template metaprogramming:** Templates are used to create programs that run at compile time. This also allows creating templates (metaclasses) that generate new classes at compile time. If the object oriented flavor allows us to have run-time polymorphism, generic programming introduces compile time polymorphism.
- **New generation C++ (C++11):** A new C++ standard that introduces lots of new concepts: Lambdas, constexpr, move semantics… and that also improves the language itself (for example, it specifies a memory model for concurrency or includes variadic templates).

A C++ program can be entirely written using only one of this "flavors", but many programs will probably mix at least two of these flavors. Learning C++

should be a process which takes years, but I had no previous experience with it (JAVA background). This is the list of books that I have read:

- **Introduction to C++:**
  - *Accelerated C++: Practical Programming by Example*[39]
  - *Thinking in C++: Introduction to Standard C* [40]
  - *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*[41]
- **Standard Template Library:**
  - *The C++ Standard Library: A Tutorial and Reference*[42]
- **Template metaprogramming:**
  - *C++ Templates: The Complete Guide*[43]
  - *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*[44]

After reading some of these books, you realize that it was just the tip of the iceberg. One thing is to write C++ code, and a completely different thing is to write **platform independent code**. Other languages like JAVA work on top of a Virtual Machine, but C++ uses directly OS libraries, so you end up writing OS dependent code. The standard library handles some of the differences, but very few. The **Boost** library is a compendium of different OS independent libraries which can be used across a broad spectrum of applications. They provide the typical facilities from other standard libraries (threading, timer, containers, files system…) but also many high complex functionalities (Graph libraries, asynchronous input/output, testing, template metaprogramming frameworks…).

Although learning the Boost libraries is not a must, they are so useful for rapid prototyping that I consider that some of them have the same importance as the STL. Both the STL and the Boost libraries use templates extensively, so it is important to understand them (this is why I have included in the book list two template metaprogramming books).

Moreover, when the code increases its complexity, there is a need to perfectly understand how things work under the hoods. For example, using exceptions is relatively easy, but if the life-cycle of the objects in your program is complex, you need to fully understand concepts like stack-unwinding or how does the object construction and destruction work. Although the final reference is the C++ standard, it is too technical. Hence, the best book to learn the C++ internals is:

- The C++ Programming Language [45]

## *Developing with C++*

Creating small projects (with 3 or 4 source files) is easy enough to be handled by hand. However, as the project grows in size and complexity, the usage of a good IDE becomes a necessity. I have chosen Visual Studio 2010. However, using an IDE is not a panacea, there are many different things that you have to keep in mind as a programmer :

- After compilation there is another step called linkage. You really need to understand how do they work, and the importance of properly using header and source files. The book from Stroustrup is a good start, but you quickly need more information:
    - Compiler, assembler, linker and loader: a brief story [46]
    - Write Great Code: Volume 1: Understanding the Machine [47]
    - Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level[48]
- Debugging is as important as programming. A good reference to learn how to debug with VS is:
    - Mastering Debugging in Visual Studio 2010 - A Beginner's Guide [49]
- To use version control. In this case, I decided to use GIT
    - Pro Git [50]
- One thing is to understand how to build source code, and another is to be able to control the process in a big project. Although VS allows us to handle most of the process using the GUI, when you start linking to third party libraries and you need to, for instance, create test cases, things get more complicated. The first option is, of course, to try to use the VS facilities, like Custom Build Rules and Property Sheets. However, you suddenly realize that they are not powerful enough. Project and solution files (that store metadata of the build process) uses a special markup. The natural next step is to learn how to use it:
    - Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build [51]

  After reading that book, I realized that working with the markup language used with MSBuild was boring and that it was really difficult to express ideas. Moreover, this solution could only be used in Windows systems.
- The solution was to learn how to use CMAKE, a cross-platform and open-source build system. Although it uses its own language, it is quite fast to learn. A CMAKE script can generate VS projects, but also make files, nmake files… Moreover, CMAKEs allows us to easily integrate third party libraries into the building steps of our project.
    - Mastering CMake [52]
- In order to test networking multithreaded code, the programmer needs some tools to control the application. Understanding the environment where the

application runs is also very important. In our case, the environment was Windows, so I read some chapters from:

- Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7 [53]

## Using the tools that you have learnt

After all these readings, I had the tools. However, producing quality code is not just about the tools, but also about how to use them.

- Object-oriented code can degenerate in an explosion of classes and inheritance trees, and also in an uncontrolled relation between objects at run-time among others. It is important to use well-known methodologies (patterns):
  - Design Patterns: Elements of Reusable Object-Oriented Software [54]
- Being able to expose the ideas is also very important. For example, in order to present algorithms. In my case I've used state chart UML diagrams in order to express and program the Zookeeper algorithm. It is important to both understand how to represent state-machines, but also the different techniques to implement them:
  - Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems [32]
  - Qt:The State Machine Framework [33]

## What has been needed to develop the proof of concept

- IDE: Visual Studio 2010 Professional
- Build system: CMAKE 2.8.9
- Third party libraries:
  - Boost
    - Signals: Controll the poller life-cycle
    - Chrono: Tickless timer implementation
    - System: zstore
    - Random + uuid: Generation of unique identifiers
    - Thread: cross-plattform thread implementation
    - Enable_shared_from_this: singleton implementation
    - Interprocess: Usage of the atomic integer implementation (Singleton)
  - ZeroMQ: Threading and network communication
  - Cxxtest: Unit testing framework

    - Google protobuffer: Serialization of messages between actors
    - Zookeeper:
    - CPLEX
    - Log4cxx: Logging facilities
    - Doxygen: Documentation system
- Libraries developed:
    - QZMQ:
        - ZeroMQ binding for C++
        - Event-driven framework
        - Actor implementations
    - QZK
        - Integration of the zookeeper C library into QZMQ.
        - Task distribution framework
    - QPLEX
        - CPLEX worker and client using QZK
- External software:
    - Zookeeper Java server
    - Gradle: Used to build exhibitor in Windows
    - Netflix Exhibitor: Controls zookeeper clusters
    - Netflix curator: Java library that extends zookeeper. Used to learn from it.
    - From Sysinternals:
        - TCP view: Allows to see in real time all the TCP sockets on the system
        - Process Explorer: See threads in real time
    - Windows firewall Notifier: Allows to quickly create Windows firewall rules in real time.

## What went wrong

The following problems have been fixed in order to use some of the previously mentioned libraries:

- Protobuffer
    - The template function **make_pair** is used with explicit template arguments, for example make_pair<string,string>, and do not compile in neither VS 2010 nor 2012. As it it stated in http://msdn.microsoft.com/en-us/library/bb531344.aspx, there are two solutions:
        - Use make_pair without explicit arguments

        - Make_pair is an auxiliary function which helps to create **pairs,** so if we want to call make_pair with explicit arguments, we can manually create the pair: From make_pair<string,string> to pair<string,string>.
    - In order to get the first or the second element of a pair, the form p.get<N> is used. However, it does not compile in VS 2010. As a fix, we can use the function std::pair::get<N>(p), where p is a pair instance.
    - There is no protobuffer integration with Visual Studio. A custom made CMAKE script let us automatically recompile protobuffer definitions when they change.
- Log4cxx
    - The entire project fails to compile under VS 2010. Fortunately there is a project in github that allow us to compile it: https://github.com/venkatperi/log4cxx
- Zookeeper C client library
    - Although it uses the pthread library (a UNIX threading library), it got a native Windows version in the 3.4.0 release. A Visual Studio 2005 solution[31] file is provided, but it is not correctly converted by VS2010, so we cannot even compile the library on Windows! A general solution to try to convert them is to use the tool **Visual Studio Project Converter** (http://vsprojectconverter.codeplex.com/). However, it did not work. Since we are using CMAKE as our preferred build system, the natural step was to create our own CMAKE script to build it. The script is also able to automatically generate the doxygen documentation.
    - Windows support is just a patch, and it does not compile in VS2010. The following source code changes had to be done:
        - **zk_log.c** : Include **winport.h** if WIN32 is defined, since it implements the **gettimeofday** function.
        - **Winstdint.h**: Remove everything and substitute it with **#include <stdint.h>.** The new C++ standard (C++11) has all the integer definitions.
        - Many POSIX functions return well known error codes, like EWOULDBLOCK or EINPROGRESS. And this works fine with POSIX TCP sockets. However, in windows ou have to use WINDOWS sockets (defined in **winsock.h**). These functions return different error codes:

[31] [31] A solution file describes all the building steps, the code to compile, libraries to link against...

WSAEWOULDBLOCK and WSAEINPROGRESS. The library implementers have used the <errno.h> header file, which defines in windows all the POSIX error numbers. However, the value of EINPROGRESS is not equal to WSAEINPROGRESS! This mens that error codes are incorrectly interpreted, and everything crashed in very subtile ways. As a workaround, the solution is to continue including <errno.h>, but after it use the following code:

```
/*********************************************/
/* THIS IS A HACK                            */
/*********************************************/
#undef EWOULDBLOCK
#undef EINPROGRESS
#define EWOULDBLOCK WSAEWOULDBLOCK //macro redefinition
#define EINPROGRESS WSAEINPROGRESS
```

- **Zookeeper.h**: The zookeeper library will be used as a dynamic library (DLL). DLL functions use a different calling convention than normal functions, in order to properly export the symbol names:
  - Building the library: __declspec(dllexport)
  - Using the library: __declspec(dllimport)

  They have forgotten to properly export all the zoo_xxx_op_init function declarations. The solution is to add **ZOOAPI** before them.

## Running zookeeper servers

The official administrator's guide [55] explains how to configure a cluster of zookeeper servers. However, it does not provide any solution for the following problems:

- If the zookeeper server process crashes, it is not automatically reexecuted. If you want to do it, you need a supervisor process, like supervisord (only on UNIX systems).
- The servers create transactional logs and snapshots of the znodes, which are stored on disk. However, Zookeeper does not automatically remove old snapshots and log files (since they could be useful for debug purposes).

Nonetheless, these files grow really fast, and can generate many GBs per day.

- Cluster configuration is static. If for some reason we want to increase the number of servers, the entire system must be stopped and restarted again with the new configuration.

The solution is to use Exhibitor, an unofficial supervisor system for Zookeeper [56]. It automatically restart the zookeeper process if it crashes, deletes the outdated log files and let us **change dynamically the number of servers that form the cluster**. Unfortunately, it does not run on Windows, and I've had no time to modify the source code. However, it should not be very difficult to do.

## Distributed parallel optimization: the CPLEX remote object

CPLEX introduced in version 12.5 the ability to construct distributed parallel optimization algorithms.

CPLEX provides multiple APIs for different programming languages. All of them use as its base core the CPLEX Callable Library, which is a C API (see below).

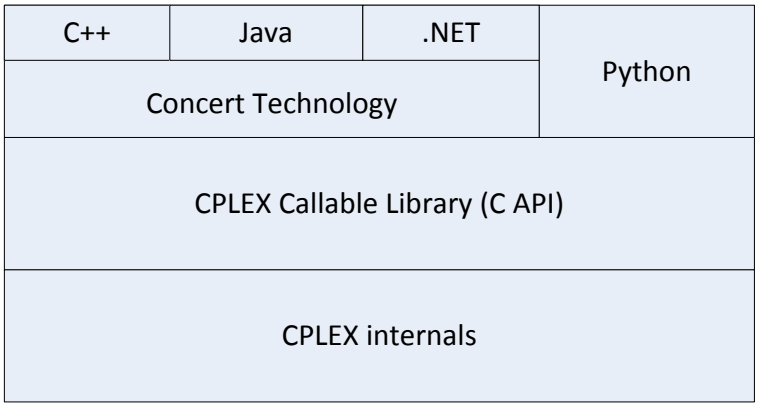



Figure 82 CPLEX technologies

From all the APIs, the CPLEX functionalities are accessed via an **environment** variable, which stores all the context information needed by the CPLEX internals.

The approach to developing distributed CPLEX algorithms has been to add a new layer between the CPLEX Callable Library and the different language bindings: the CPLEX remote object API (see Figure 83).

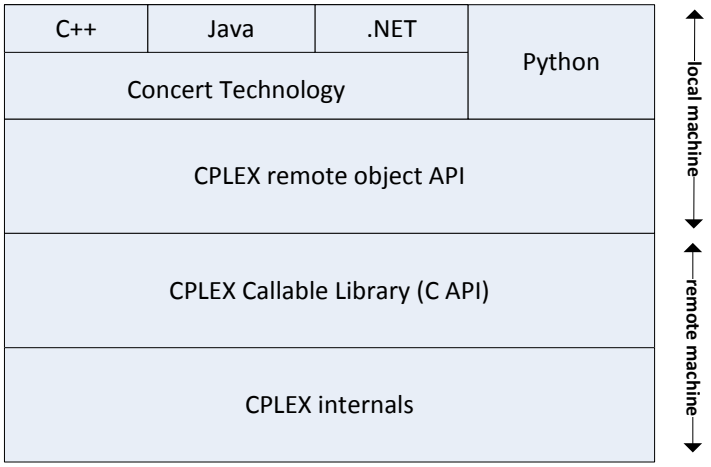
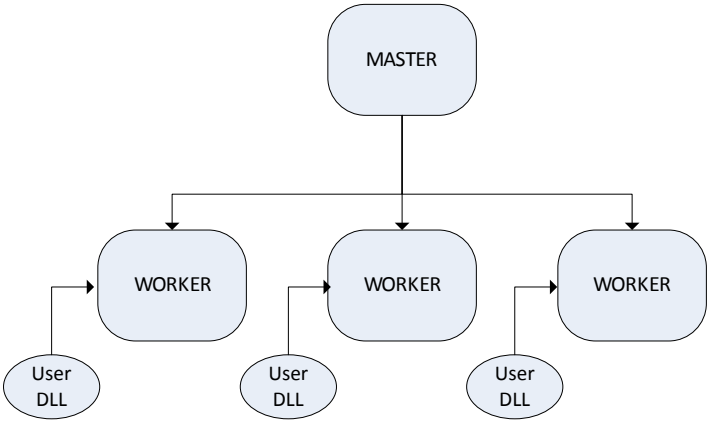



Figure 83 CPLEX remote object API

The remote object API lets the application user to create an environment object. The methods of this object are in reality stubs that connect to a remote process. That is, the design follows a typical RMI (see section 2.2). In such a way, it is relatively easy to transform single-machine CPLEX algorithms to distributed algorithms.

There are three major components in the distributed application:

- **Master**: Written by the application user. It has to handle the connection between the server and the workers, poll the workers for the solutions, control the life-cycle of the workers, submit problems to be executed by the workers….
- **Worker**: Each instance of this node is a remote Callable Library Object. It does not run user-code. However, its behavior can be extended using user provided Dynamically Loaded Libraries (DLLs).




The communication between Master and Workers can be performed using 4 different kinds of transport:

- **Local:** This is the non-distributed version, where the environment object lives in the same process.
- **Process:** A new process is created, and the communication between master and that process uses names pipes.
- **Mpi:** Message Passing Interface is used.
- **TCP/IP**

The great thing is that the API abstract the transport method used, so the only thing that the programmer has to do is to configure the transport, and then forget about it.

However, as we said, the master is written by the application user so it is he, who has to handle everything.

- Initiate the communication with the machines. This is the same as creating a remote environment object using CPXXopenCPLEXremote.
- Send the problems to the workers (CPXXreadcopyprob). The simplest solution is to send the problem process per process. However, CPLEX allows the user to create an environment **group** (CPXXcreateenvgroup). Using a group, it is possible to use the multicast operations(CPXXreadcopyprob_multicast), which may improve the "send operation" performance (for instance, if MPI is used in a LAN network where it is possible to broadcast messages).
- From the CPLEX tutorial, we see that now the master has to do the following actions:

```
while ( all machines are solving )
```

```
  display progress message
  if ( some optimization condition is true )
    stop all machines
stop all machines that are still running
report the results
```

The most interesting part of the API is that it is possible to perform asynchronous operations on each of these environments:

- CPXX<operation>_async: Executes the operation asynchronously.
    - It is possible to access the status of the execution calling CPXXasynctest
    - It is possible to stop the execution using CPXXasynckill
    - It is possible to wait until an asynchronous operations finishes using CPXX<operation>_join. When the operation returns, the user can query the objective function value and algorithm status.
- The remote workers can communicate asynchronously with the master. The master installs callbacks that will be executed in a CPLEX thread when the worker sends him a message. There are two message flows:
    - Information messages: Inform about the state of the algorithm. These messages are sent at given steps of the algorithms. The master can use them to decide that a solution has been found distributedly.
    - User remote functions: We explained before that the user can extend the behavior of the workers using user-defined DLLs. The master can explicitly trigger the execution of one of these functions. The execution is asynchronous, and the result of it is handled by an user-function callback.
- CPLEX provides support for some common types of data(int, long, double…). If the data that the asynchronous function is more complex, the API provides a CPXSERIALIZER and CPXDESERIALIZER.

The solution provided by CPLEX is great because it is highly integrated with the CPLEX API. It lets the user execute almost any CPLEX function remotely. However, it has a really BIG disadvantage against our framework: It does not provide any kind of fault-tolerance. They have followed a master-slave architecture statically configured, where if anything fails the algorithm has to be restarted. It shows the same lack of fault-tolerance that MPI has. Luckily, things do not fail too often, so these kind of techniques works pretty well in a cluster.

## General recommendations for Quintiq

Although this document is large, one of its multiple purposes is to try to be as a practical introduction to distributed systems for Quintiq. We have studied ZeroMQ and Zookeeper, and we have briefly mentioned distributed abstractions like failure detection and leader election. The aim of this section is to propose applications of some of the concepts studied within the Quintiq system and/or reason why or why not a given technology should be used.

### Zookeeper

Zookeeper has proven to be great for coordination purposes. Quintiq heavily relies on configuration files in order to connect to the different nodes in the system. Moreover, if for example a dispatcher dies all the clients connected to it must be manually restarted and connected to a different one. Of course, the user needs to know that the given dispatcher has access to the dataset the client user is interested in. This problem is easily solvable using Zookeeper. Zookeeper could be used as a centralized fault-tolerant configuration repository:

- Provide a real-time representation of all the nodes running with all their configurations. For example, dispatchers could update their information with all the datasets that they have filtered, or the cached parts of a dataset. This could also be used as a real-time diagnostic tool: Each node could update periodically its znode with information of CPU or memory usage. Another application could control these statistics and act conveniently (for example, executing new dispatchers).
- Allow dynamic system reconfiguration. For instance, let us suppose that a dispatcher crashes. Right now all its children must be restarted and reconnected to a different node. Using zookeeper, the znodes could discover other dispatchers, and choose the one that contains the information they are interested in (they have not filtered out the dataset or they have our interests cached). It would be possible to reconnect to the best dispatcher.
- A different branch can be used to provide static (or almost static) configurations. In that way, the deployment of new nodes would be almost automatic. New deployment would have to download the configuration. It is possible to play with the version number of the znodes in order to avoid downloading the configuration files each time that the nodes initiate.

## ZeroMQ

ZeroMQ is a high level transport abstraction that let us quickly prototype lots of different architecture without having to deal with reconnections, connection losses or many other problems. Its asynchronousity makes it perfect to be a network stack replacement from the one used by Quintiq, which is Boost ASIO. It could be used to easily re-implement the publish-subscribe architecture. However our main concerns about ZeroMQ are also their main benefits: **since all the connection problems are automatically handled by the library, it is very difficult to detect when something fails.** Depending on the type of socket, ZeroMQ follows two different strategies: either try to reconnect forever (request socket) or silently discard the message sent (router socket). The bad thing is that it is fairly difficult to avoid the infinite reconnection process, but it is impossible to detect when a message has been discarded.

The last sentence is quite disturbing: If we want to detect that a node has been disconnected, we must use a different socket to implement some kind of failure detection mechanism. Although this play nicely with the idea of separating different flows of information using different sockets, this is not the optimal solution (at least regarding OS resources).

Another problem with ZeroMQ is that it is not possible to know from which node we are receiving a message. That is, we cannot know its IP or endpoint. In fact, that's completely normal knowing that ZeroMQ provides different transports (some of them do not have the concept of IP). The only socket that knows about identities is the router socket. However, these identities are either generated by it randomly or are sent as part of the message. Due to the router's nature, implementing routing strategies different from the standard ones is fairly difficult.

Another problem with ZeroMQ is that it uses its own internal message format, which makes it incompatible with standards like HTTP or FTP.

Although we have mentioned some of the problems of ZeroMQ, we do not want to discredit this library. We think that these drawbacks should not stop us from using it.

## Failure detection

Zookeeper allows us to detect failed nodes using ephemeral znodes. However, the main drawback of using Zookeeper is that all the nodes must be connected to it, and must maintain a TCP connection sending ping messages. In the nowadays Quintiq architecture, nodes are only interested to know if their immediate parent

or their immediate children are alive, and hence we really do not need Zookeeper for this. We have also seen that ZeroMQ is not the best choice to implement failure detection.

The Quintiq implementation uses Windows socket facilities to detect that a node has crashed. However, this is far away from the best possible solution. The Internet can be considered a partially synchronous system. This means that it is not possible to implement a Perfect Failure Detector (that is, that eventually all the crashed nodes are detected and that any correct node is suspected to have failed). Hence, a failure detection algorithm will try to minimize the probability to detect incorrectly that a correct node has failed.

Using the current Quintiq implementation, if the socket is closed due to a connection error, the node is detected as failed. However, connection errors are normally transient and quickly recovered automatically. We think that it would be a great idea to invest some time to implement some failure detection mechanisms, which could be used to increase the stability of the system due to transient connection losses.

## The problem with time

If we do not consider Albert Einstein's theory of relativity, time is the same for everyone. We typically use it for mainly two purposes: to calculate durations and to order events.

Durations are used for instance to detect that a given node has crashed, or to measure the round trip time of a message. However, order is even more important: We use it to establish causality between events. It is used to timestamp logs or to maintain consistency in a transaction system. Let us talk a little bit more about the logs.

Logs are mainly used to debug programs, since they let us understand or discover the root cause of a certain problem. Logging within the same machine is easy: we can rely on the system clock, and order all the logs via their timestamp. The problem arises when we need to merge logs from different machines. The timestamp method is completely invalid, due to unsynchronized clocks.

As you guessed, a common solution is to synchronize them using mechanisms like the Network Time Protocol. However this solution is not perfect and cannot be used in time critical applications (its accuracy is in the order of the milliseconds). Other solutions rely on expensive hardware installations, like GPS

and atomic clocks to synchronize their machines around the world (TransLattice Elastic Database or Google Spanner).

However, it is also possible to order things in a distributed environment using **logical time**, which only uses message passing techniques and does not need any kind of clock to determine the causal relation between events.

It would be interesting to implement a distributed logging engine using logical time, which could be used for instance to debug distributed CPLEX algorithms.

# Bibliography


[1] K. Raymond, "Reference model of open distributed processing (RM-ODP): Introduction," in *IFIP TC6 International Conference on Open Distributed Processing*, 1995, pp. 3–14.

[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2011.

[3] M. Jelasity and O. Babaoglu, "T-Man: Gossip-based overlay topology management," *Engineering Self-Organising Systems*, pp. 1–15, 2006.

[4] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition*. 2004.

[5] C. Caloian, "Quintiq Software Transactional Memory." .

[6] S. James and P. Crowley, "Fast content distribution on datacenter networks," in *Architectures for Networking and Communications Systems (ANCS), 2011 Seventh ACM/IEEE Symposium on*, 2011, pp. 87–88.

[7] J. Basney, M. Livny, and T. Tannenbaum, "Deploying a high throughput computing cluster," *High performance cluster computing*, vol. 1, no. 5, pp. 356–361, 1999.

[8] "Beginner's Guide to Oracle Grid Engine 6.2." .

[9] "Oracle® Grid Engine Installation and Upgrade Guide Release 6.2 Update 7." .

[10] "N1 Grid Engine 6 User's Guide." .

[11] "condor Manual." Condor Team, University of Wisconsin–Madison.

[12] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," in *Grid Computing*, 2003, pp. 299–335.

[13] "Berkeley Lab Checkpoint/Restart (BLCR)." [Online]. Available: http://crd.lbl.gov/groups-depts/future-technologies-group/projects/BLCR/. [Accessed: 03-Dec-2012].

[14] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 59–72, 2007.

[15] "Cloudera Hadoop Training: MapReduce and HDFS," *Vimeo*, 11-Mar-2009. [Online]. Available: http://vimeo.com/3584536. [Accessed: 21-Aug-2012].

[16] "Highly Availabile Central Manager." [Online]. Available: http://dsl.cs.technion.ac.il/projects/gozal/project_pages/ha/ha.html. [Accessed: 30-Nov-2012].

[17] imatix, "Multithreading Magic - zeromq." [Online]. Available: http://www.zeromq.org/whitepapers:multithreading-magic. [Accessed: 30-Nov-2012].

[18] Microsoft, "Solving 11 Likely Problems In Your Multithreaded Code." .

[19] "An Introduction to Lock-Free Programming." [Online]. Available: http://preshing.com/20120612/an-introduction-to-lock-free-programming.

[20] D. C. Schmidt and I. Pyarali, *The Design and Use of the ACE Reactor-An Object-Oriented Framework for Event Demultiplexing*. 2007.

[21] D. Pollak, *Beginning Scala*. Apress, 2009.

[22] P. Hintjens, "ØMQ - The Guide." [Online]. Available: http://zguide.zeromq.org/page:all. [Accessed: 28-Aug-2012].

[23] D. Charousset, "libcppa - An actor library for C++ with transparent and extensible group semantic." .
[24] "Cloudless." [Online]. Available: http://cloudless.io/.
[25] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.
[26] "Apache Mesos." [Online]. Available: http://incubator.apache.org/mesos/.
[27] "Netflix Curator." [Online]. Available: https://github.com/Netflix/curator/wiki.
[28] "ZooKeeper Failure Detector Model." [Online]. Available: http://wiki.apache.org/hadoop/ZooKeeper/GSoCFailureDetector.
[29] "Handling the errors a ZooKeeper throws at you." [Online]. Available: http://wiki.apache.org/hadoop/ZooKeeper/ErrorHandling.
[30] "ZooKeeper Programmer's Guide." [Online]. Available: http://zookeeper.apache.org/doc/r3.4.5/zookeeperProgrammers.html.
[31] "ZooKeeper Recipes and Solutions." [Online]. Available: http://zookeeper.apache.org/doc/trunk/recipes.html.
[32] M. Samek, *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*, 2nd ed. Newnes, 2008.
[33] "Qt: The State Machine Framework." [Online]. Available: http://doc.qt.digia.com/qt/statemachine-api.html.
[34] N. Trifunovic, "C++ Exceptions: Pros and Cons." [Online]. Available: http://www.codeproject.com/Articles/38449/C-Exceptions-Pros-and-Cons.
[35] "Google C++ Style Guide." [Online]. Available: http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml.
[36] R. Chen, "Cleaner, more elegant, and wrong." [Online]. Available: http://blogs.msdn.com/oldnewthing/archive/2004/04/22/118161.aspx.
[37] "Protocol Buffers: Developer Guide." [Online]. Available: https://developers.google.com/protocol-buffers/docs/overview?hl=es.
[38] A. Yera, "Zookeeper task distribution framework: step by step." [Online]. Available: http://prezi.com/b33ut-09nx1z/zookeeper-algorithm/?auth_key=38ab5d1a7dc48408b4969e38a0f9d1c201bd3043&kw=view-b33ut-09nx1z&rc=ref-12949682.
[39] A. Koenig and B. E. Moo, *Accelerated C++: Practical Programming by Example*, 1st ed. Addison-Wesley Professional, 2000.
[40] B. Eckel, *Thinking in C++: Introduction to Standard C++, Volume One (2nd Edition)*, 2nd ed. Prentice Hall, 2000.
[41] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed. Addison-Wesley Professional, 2005.
[42] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd ed. Addison-Wesley Professional, 2012.
[43] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, 1st ed. Addison-Wesley Professional, 2002.
[44] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.
[45] B. Stroustrup, *The C++ programming language*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[46] "COMPILER, ASSEMBLER, LINKER AND LOADER: A BRIEF STORY." [Online]. Available: http://www.tenouk.com/ModuleW.html.
[47] R. Hyde, *Write Great Code: Volume 1: Understanding the Machine*, 1st ed. No Starch Press, 2004.
[48] R. Hyde, *Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level*, 1st ed. No Starch Press, 2006.
[49] A. Jana, "Mastering Debugging in Visual Studio 2010 - A Beginner's Guide." 06-May-2010.
[50] S. Chacon, *Pro Git*, 1st ed. Apress, 2009.
[51] S. I. Hashimi and W. Bartholomew, *Inside the Microsoft Build Engine: Using MSBuild and Team Foundation Build*, Second Edition. Microsoft Press, 2011.
[52] K. Martin and B. Hoffman, *Mastering CMake*, 5th ed. Kitware, Inc., 2010.
[53] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*, Sixth Edition. Microsoft Press, 2012.
[54] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.
[55] "ZooKeeper Administrator's Guide." [Online]. Available: http://zookeeper.apache.org/doc/r3.4.5/zookeeperAdmin.html. [Accessed: 28-Nov-2012].
[56] "Netflix Exhibitor." [Online]. Available: https://github.com/Netflix/exhibitor/wiki. [Accessed: 28-Nov-2012].