Intelligent Systems Group

informatika fakultatea facultad de informática

eman ta zabal zazu

Universidad del País Vasco Euskal Herriko Unibertsitatea

Konputagailuen Arkitektura eta Teknologia Saila

Departamento de Arquitectura y Tecnología de Computadores

Contributions to High-Throughput Computing Based on the Peer-to-Peer Paradigm

by

Carlos Pérez Miguel

Supervised by José Miguel-Alonso and Alexander Mendiburu

Dissertation submitted to the Department of Computer Architecture and Technology of the University of the Basque Country (UPV/EHU) as partial fulfilment of the requirements for the PhD degree in Computer Science

Donostia-San Sebastián, 2015

Why do you go away? So that you can come back. So that you can see the place you came from with new eyes and extra colors. And the people there see you differently, too. Coming back to where you started is not the same as never leaving.

Terry Pratchett.

A mi familia y amigos
por su apoyo y paciencia.

Acknowledgements

There are so many people I would like to thank for helping me and supporting me along these years. I will try to keep this list as brief as possible.

To my two advisors Jose and Alex for their guidance and patience. To the members of the Intelligent Systems Group and the PhD candidates in the San Sebastian School of Computer Science for their support. In particular, I would like to thank the Computer Architecture section of the ISG for the useful discussions and support. To Prof. Ke Tang and Thomas Weise from the University of Science and Technology of China. Thanks for hosting me in my research visit.

This work has been supported by the Spanish Ministry of Science and Innovation [projects TIN2007-68023-C02-02, TIN2008-06815-C02-01, TIN2010-14931 and Consolider Ingenio 2010 CSD2007- 00018], the Basque Government [Saiotek, Research Groups 2007-2012, IT-242-07] and the Carlos III Health Institute [COMBIOMED Network]. This work was partially carried out when the author was visiting the University of Science and Technology of China (USTC) under the grant NICaiA (Nature Inspired Computation and its Applications, PIRSES-GA-2009-247619). The author was supported by a doctoral grant of the Basque Government. The author is an affiliated PhD student member of the European Network of Excellence on High Performance and Embedded Architecture and Compilation (HIPEAC).

Quisiera agradecer a mis padres Clemente y Carolina, por su apoyo, así como por la educación que me han dado y por enseñarme curiosidad y paciencia; a mi hermana Susana, por que siempre está ahí, gracias por su apoyo y consejo. Sois una parte importante de esta aventura. Sin vosotros esta tesis no hubiera sido posible.

Quisiera también agradecer a una serie de personas que han hecho posible esta tesis: César, Xabi, Antonio, Rubén, David, Iñaki, Fran, Pablo y Alex, gracias por todas las conversaciones, las comidas, los partidos de frontenis y las cervezas. Gracias también a los Bucléicos y demás parisinos y ex-parisinos por las suaradas, las buclas infinitas, las paellas y los pique-niques en cualquier esquina que se preste a ello; aunque ya no esté allí os tengo presentes. Y por último gracias también a los integrantes de la peña Ideales, por todas las risas y los duros. Sin vosotros todo esto no tendría sentido. Sois una parte importante de este trabajo.

Abstract

This dissertation focuses on High Throughput Computing (HTC) systems and how to build a working HTC system using Peer-to-Peer (P2P) technologies. The traditional HTC systems, designed to process the largest possible number of tasks per unit of time, revolve around a central node that implements a queue used to store and manage submitted tasks. This central node limits the scalability and fault tolerance of the HTC system. A usual solution involves the utilization of replicas of the master node that can replace it. This solution is, however, limited by the number of replicas used.

In this thesis, we propose an alternative solution that follows the P2P philosophy: a completely distributed system in which all worker nodes participate in the scheduling tasks, and with a physically distributed task queue implemented on top of a P2P storage system. The fault tolerance and scalability of this proposal is, therefore, limited only by the number of nodes in the system. The proper operation and scalability of our proposal have been validated through experimentation with a real system.

The data availability provided by Cassandra, the P2P data management framework used in our proposal, is analysed by means of several stochastic models. These models can be used to make predictions about the availability of any Cassandra deployment, as well as to select the best possible configuration of any Cassandra system. In order to validate the proposed models, an experimentation with real Cassandra clusters is made, showing that our models are good descriptors of Cassandra's availability.

Finally, we propose a set of scheduling policies that try to solve a common problem of HTC systems: re-execution of tasks due to a failure in the node where the task was running, without additional resource misspending. In order to reduce the number of re-executions, our proposals try to find good fits between the reliability of nodes and the estimated length of each task. An extensive simulation-based experimentation shows that our policies are capable of reducing the number of re-executions, improving system performance and utilization of nodes.

Table of Contents

1

Introduction

This introductory chapter is devoted to setting up the general background about the aspects addressed in this work: HTC systems and P2P networks. Further details about any topic related with the aforementioned elements, but not used directly in this dissertation, can be consulted in the included bibliography. This chapter is organized as follows: Sections 1.1 and 1.2 are devoted to presenting some background information about HTC systems and P2P networks respectively. Finally, Section 1.3 presents the main contributions and the structure of this dissertation.

1.1 High Throughput Computing Systems

A distributed system is composed of a collection of interconnected computers working together in order to achieve a common goal. We can find different types of distributed systems, depending on the level of coupling of the applications executed by the nodes. A High Throughput Computing (HTC) system is a loosely coupled form of distributed computing platform whose objective is to execute as many independent computational tasks per unit of time as possible. In contrast, a High Performance Computing (HPC) system runs tightly coupled tasks that belong to the same application, with the objective of completing the application in the shortest possible time. It must be noticed that it is usual to find combinations of both paradigms. In fact, most supercomputing centers combine HTC and HPC features, scheduling resources to efficiently run many parallel applications. In this dissertation we focus only on HTC systems, with loosely coupled nodes that execute independent tasks.

HTC systems are composed of a large number of computational resources tied together by some sort of interconnection network. These resources are used to execute applications submitted to the system by different users. Tasks can be executed in any order, although most HTC systems offer several scheduling policies, such as arrival order, the length of the task, fairness and priority.

As previously commented, in this dissertation we suppose that all submitted tasks are independent. However, on top of a basic HTC service scheduling independent tasks, it is possible to implement other kinds of tasks, such as *work-flows*, tasks that are composed of a set of interdependent subtasks that have to be executed in a given order, or *bag-of-tasks*, kinds of tasks composed of a set of subtasks that operate over the same data but without interdependencies among them, so they can be executed in any order.

Users access an HTC system by using a queueing system that can be formed by one or multiple queues. This queueing system is usually implemented in a master node in charge of the scheduling process. Without loss of generality, in this work we are going to consider systems of only one queue. This queue is used by the users to submit tasks, which wait until the resources required to run them are available. These resources are specified by the user when the task is submitted to the system, and may include many requirements such as specific processor architectures, minimum size of memory, operating systems, libraries, etc.

The aforementioned master node is in charge of distributing tasks in the queue to the available worker nodes, controlling them and notifying the owner of the different tasks about their completion. Through the interface provided by this node, users can submit their tasks, control them and fetch the results when they are completed. In Figure 1.1 we show this general architecture: a set of workers, that only execute tasks, connected to a master node which is in charge of accepting and scheduling tasks to idle nodes.

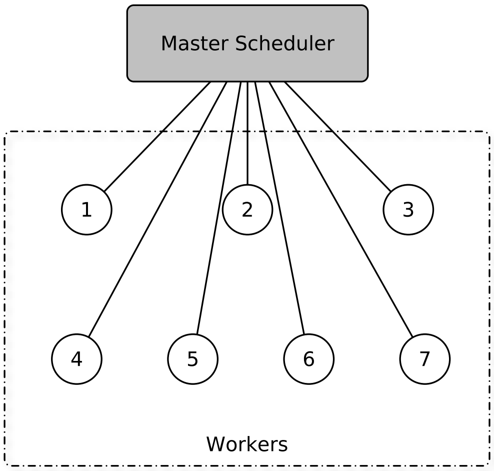


Fig. 1.1: Architecture of an HTC system.

This centralized architecture is usually the main point of failure of HTC systems. If the master fails, the system becomes unreachable and no new tasks can be submitted to the queue. Usual solutions to this problem consist of mirroring the master node in such a way that the failure of the master can be fixed instantly by swapping the failed node with its backup node. However, this solution, while extending the fault-tolerance of the system, is limited by the fault-tolerance of the set of replicas used. Also, centralism presents another problem: scalability in terms of number of worker nodes and enqueued tasks, which are limited by the local resources of the master.

In this dissertation we tackle the fault-tolerance problem of HTC systems using a different approach: we embrace the P2P philosophy. Using as a foundation a P2P storage system, all working nodes participate in holding the information related to submitted tasks, that is, the task queue. Then, every node in the system can access this queue to select an appropriate task to execute. With this approach, the fault tolerance and scalability of the HTC system is limited only by the number of nodes in the system. In Figure 1.2 we have a depiction of this architecture. P2P storage systems scatter and replicate information along the system nodes, in such a way that the HTC-P2P system can operate as long as there is a node alive. The fault-tolerance of the HTC system is, therefore, extended from the capabilities of a master node (or a group of masters) to the fault-tolerance of the entire set of nodes.

Among the HTC systems that can be found in the literature, we can cite systems to control dedicated clusters, such as TORQUE [1], Oracle Grid Engine [2] or HTCondor [3], and volunteer grid computing systems. The volunteer grid computing paradigm consists of a set of compute nodes belonging to different users and organizations that, altruistically, donate their idle resources to execute computational tasks for one or more research projects. Volunteer computing systems evolved from project-specific solutions (for example, Seti@Home [4]) to generic platforms that can be used by several projects (for example, Boinc [5]). They use a per-project central management node that controls a pool of distributed resources, and include mechanisms to deal with node failures. The central node, thus, may become an issue.

After this brief presentation of the HTC paradigm, we now provide some background information about peer-to-peer networks.

1.2 Peer-to-Peer Networks

Peer-to-Peer systems are distributed systems in which there is neither a central control point, nor a hierarchical structure among its members. In a P2P system, all nodes in the system have the same role, and are interconnected using some kind of network (usually, the Internet), defining an application-level virtual network, also called *overlay.* Nodes communicate using this overlay, in order to find information, share resources or allow human users to communicate.

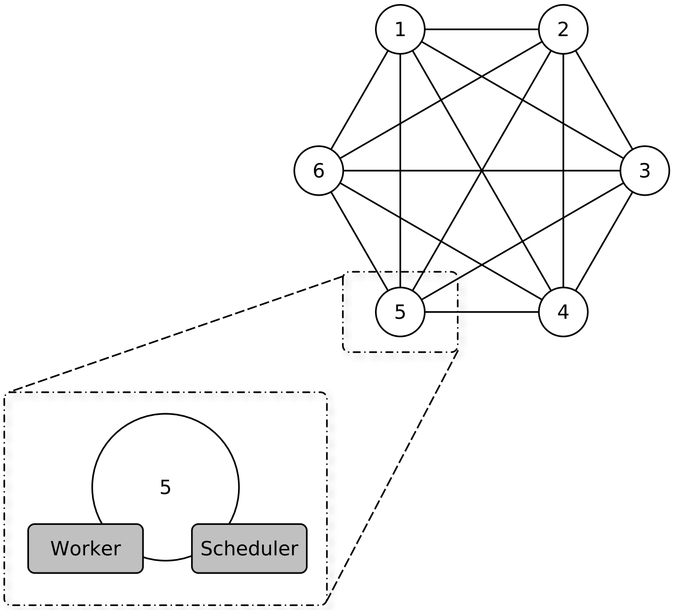


Fig. 1.2: Architecture of a distributed HTC system.

Unlike Grid systems, P2P systems do not interconnect well-defined groups using highly reliable networks. They are based on a large number of unreliable resources using unreliable network connections. Even under these limitations, they manage to provide some interesting services and features such as scalability, fault tolerance, efficient information search, highly redundant storage, persistence or anonymity.

According to Lua et al. [6], we can classify P2P systems into two basic types of *overlays*: structured or non-structured. Those of the later type are formed by randomly connected nodes. Since there is no structure, *flooding* routing protocols are used to communicate peers. This way any network node can be reached from any other point in the system, but at the expense of an important efficiency penalty.

In Figure 1.3 we can see an example of a flooding algorithm, *Breadth-First Search*, in which each node forwards the search request to all of its neighbours until a hop limit value is reached. In the example we can see that node 1 wants some information stored in node 5 but does not know its exact location, so at t_0 it contacts its neighbour nodes 2, 3 and 4. These nodes do not possess the requested information, so they forward the request to their respective neighbours, flooding the network with repeated messages. This process continues until the request reaches its destination or until the maximum hop count value is reached. If this happens, the request never reaches its destination and the search is unsuccessful. In the example, the request arrives after only 1 hop to the destination, node 5. At this moment node 5 knows that node 1 wants to exchange some information, so it establishes a direct communication channel with node 1.

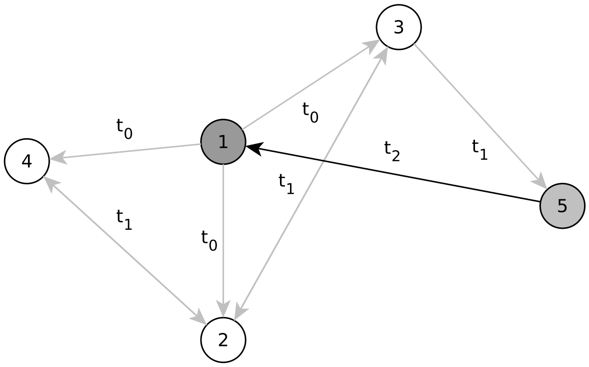


Fig. 1.3: Searching for information in a non-structured P2P network.

We can find several examples of this type of P2P networks in the literature, including Gnutella [7] and FreeNet [8], both file sharing systems based on this P2P topology.

When we talk about structured overlays, we refer to systems in which there is a well defined virtual topology, and each piece of the network *content* (whatever it means, depending on the application) is stored in a well-defined network node. If we focus on data storage, we say that this type of P2P systems implement Distributed Hash Tables (DHT) [9, 10], in which objects are located in a node (or nodes) chosen in a deterministic way.

Let us consider an overlay network with N nodes, each one with a different *ID*, and a (much larger) *key space*. A *hash function* provides a map of a key onto a node ID – there is a single map, in such a way that the node that takes care of a key is perfectly identified. This mapping defines a DHT, and is the basis of highly-scalable, distributed storage networks. In a DHT, information is stored in the form of {key, value} pairs accessible by a hash-like API, providing functions to insert and modify key-value pairs (*put(key, value)*) and to access them (*value=get(key)*).

In Figure 1.4 we can see a possible mapping of a key space in a 16-node DHT. There are 64 possible keys and each node is responsible for 4 of them. Each stored object is identified by one of these keys.

Implicitly, a *routing protocol* is used in order to deliver these requests (read or write operations) to the key owner node. Different routing protocols

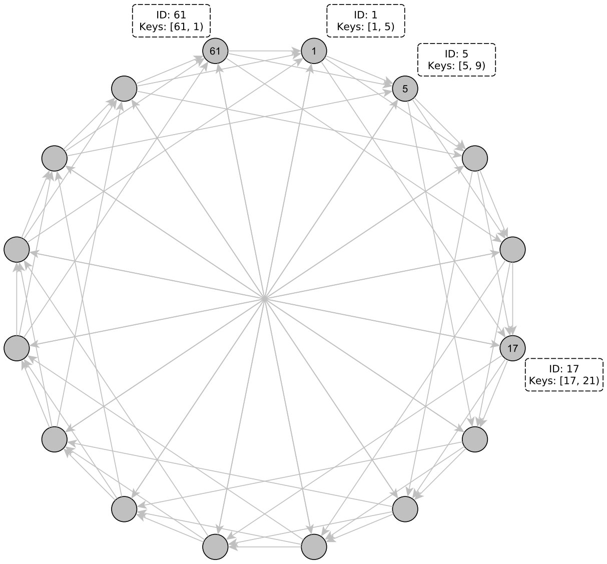


Fig. 1.4: Key space [1-64] mapped over a 16-node DHT.

have been proposed in the literature but all of them share one characteristic: the routing is done in a progressive manner, as a function of the distance to the destination node. Therefore, when a given node wants to communicate with the node that takes care of a certain key, it sends the message through the neighbour which is closer (in terms of assigned keys) to the destination. The specific definition of proximity of keys-nodes depends on the particular structure of the DHT, and varies from system to system. Most DHT systems guarantee that any object can be reached in a number of jumps in order $O(\log N)$, being N the number of network nodes.

In Figure 1.5 we show an example of routing in Chord [10], a structured P2P network that implements a DHT, for the 16-node network of the previous example. As previously mentioned, there are 64 possible keys in this DHT and each node is responsible for 4 consecutive keys. In order to increase speed in the routing process, each node is connected to other 4 nodes at different distances. In the example, the node with ID 1 wants to send a message to node 45. From its possible connections, the nearest to 45 is node 33. Node 33 forwards the message to the nearest of its neighbours, 41, which is directly connected to 45.

The weakest point in DHT systems is their behaviour in the case of fast changes in system configuration, a term also known as *churning*. Informally, this refers to nodes that join or leave the overlay, forcing a network reorganization and a modification in the key mappings. Latency can increase in this case, and several proposals exist that try to reduce this problem. In [11] an

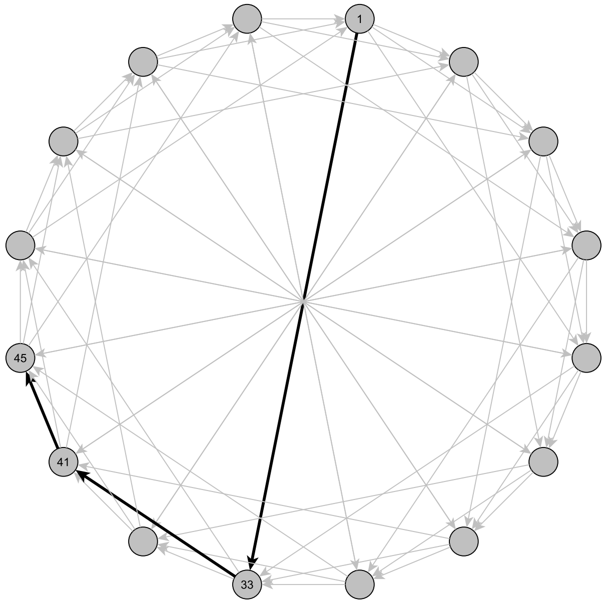


Fig. 1.5: Routing in a 16-node DHT, from node with ID 1 to node with ID 45.

algorithm is proposed that tries to reach the optimum latency in P2P networks, without losing the scalability of DHTs; also, authors of [12] propose an algorithm to maintain load balance in adverse conditions. Among the examples of this kind of P2P systems, we can find Chord [10], Tapestry [13], CAN [14], Kademlia [15], Dynamo [16] or Cassandra [17].

1.3 Organization of this dissertation

The work presented in this dissertation revolves around High Throughput Computing systems, their requirements and how to address them using Peer-to-Peer technologies. It tries to provide answers to these motivating questions: Can we build an HTC system totally decentralized with the support of a Peer-to-Peer network, with the same characteristics of a centralized system but without its drawbacks? Can we solve other typical fault-tolerance issues of HTC systems, mainly the problem of avoiding unfinished executions because of failures in the worker nodes, in a way compatible with the P2P philosophy? Which levels of availability will be provided by this HTC-P2P system?

The main contributions of this work, which try to provide answers to these questions, are:

- the design and development of a functional HTC system totally distributed, with the support of the Cassandra P2P storage system, without any main point of failure.
- theoretical models of Cassandra's fault-tolerance when facing two different types of node failures: transient and memory-less failures. These models can be used to analyse any system built over Cassandra or to make predictions about Cassandra's availability.
- a collection of scheduling policies that can be used within this HTC-P2P system as well as with other HTC systems, including a FCFS policy and several failure-aware scheduling policies.

The remainder of this dissertation is organized as follows. In Chapter 2 we present the design and development of our HTC system distributed over a P2P network. We describe the architecture of the system and its main characteristics. We validate the system by performing several functionality and scalability tests.

In Chapter 3 we analyse the fault tolerance of the P2P storage in which our HTC-P2P proposal is based: Cassandra. We propose two stochastic models of Cassandra in two different failure situations: transient failures and memory-less failures. By using these two models, a system administrator could select the best possible configuration in order to maximize the availability of the information stored in Cassandra, taking into account the use given to the system and the fault-tolerance characteristics of the different nodes forming the system.

In Chapter 4 we focus on the scheduling process of the proposed HTC-P2P system in the presence of failures, and present several scheduling policies that improve system utilization and job throughput. We address the problem of re-execution of failed tasks due to failed executions by carefully choosing the best possible worker node in terms of future availability. Experimentation shows that failure-aware scheduling policies improve the productivity obtained from the HTC system.

Finally, Chapter 5 closes this dissertation with several conclusions that summarize the contributions of our work, and with an outlook of future research lines.

2

HTC over P2P

As discussed in Chapter 1, existing HTC systems have one important characteristic that make them potentially weak: they require a central administration point. This central point imposes limitations in system scalability and fault tolerance. In this chapter we present a High Throughput Computing System totally based upon the Peer-to-Peer (P2P) paradigm in order to overcome these limitations. In the pure peer-to-peer philosophy, all the members of the proposed system are capable of carrying out administrative tasks to maintain the system operational, in addition to executing jobs. Ideally, this HTC-P2P system should perform like a non-P2P one in the absence of failures, and should scale to large networks while showing these expected characteristics: totally decentralized, not requiring permanent connection, and able to implement different scheduling policies such as running tasks in a (non-strict) FCFS order.

This chapter is structured as follows: in Section 2.1 we discuss the general caracteristics that a P2P computing system should have. Section 2.2 is devoted to presenting several P2P storage systems that could be valid for building our HTC-P2P system. We present the characteristics of the selected P2P storage system, Cassandra, in Section 2.3. Section 2.4 shows the characteristics of the proposed HTC-P2P system and how we have implemented them in Cassandra. In Section 2.5 we carry out some experiments designed to verify that it works as expected. Related work on P2P computing systems is discussed in Section 2.6. The chapter ends in Section 2.7 with some conclusions and future work on this subject.

2.1 P2P Computing

A distributed computing system can be defined as a collection of computers interconnected by a communication network. These computers join their resources in order to collectively do computational tasks. Each computer in the system has its own, independent resources; however, from the user's point of view, the system should be seen as a single *resource pool*. An interface is given to the users in order to access the system without taking care of its complexity. Actual HTC systems revolve around central points of administration. This central resource, in case of failure, can make the whole system unusable.

A true Peer-to-Peer Computing system overcomes this disadvantage by distributing management capabilities among all the system nodes. An HTC system based on the P2P model system should have these desirable characteristics:

1. Fully distributed, without centralized administration point or points: every node in the system must have the same role and every administrative task must be performed without the intervention of any central authority.
2. Users should be able to submit tasks from their machines and then disconnect from the system: users can be part of the system, executing tasks, or only submitting them. In this last situation, the rest of the system must store all the required information to execute each task and also the results of each computation for a posterior retrieval.
3. Different scheduling policies should be supported. In particular FCFS, as it is considered a “fair” execution order, is expected.

In conventional, non-P2P HTC systems, a central task queue implemented in a master node takes care of global scheduling policies and execution order. We propose emulating this data structure, but using a decentralized version of it. In this chapter we present a distributed scheduler that implements a FCFS scheduling policy. However, our proposal is flexible enough to permit the implementation of other scheduling policies. In fact, in Chapter 4 we present several failure-aware scheduling policies that try to maximize the throughput of the system by selecting, in terms of stability, the most appropriate node for each task.

This distributed queue can be implemented using the storage characteristics of a DHT, so all the information required to schedule tasks would be distributed along, and available to, all the nodes in the HTC system. However, in order to achieve the expected goals, additional features are necessary for the DHT, including persistence, resilience and consistency. In the literature we can find diverse proposals [18, 19, 20, 21, 22, 23, 24] to build a distributed and highly scalable storage system from a P2P overlay, using replication and coherence algorithms.

On top of a reliable DHT-based storage system, we can create a distributed queue object, so all nodes in the system would know which tasks are waiting and in which order they have to be run. All nodes implement a local scheduler that will check the queue object to find the next awaiting task, and execute it when the node is idle. Besides, the DHT storage system can be used as a blackboard to store all the data needed by the tasks (programs, parameters, input data, output data). This would allow nodes to launch tasks and then disconnect from the system, if necessary. Whenever they rejoin it, if the task has finished, results will be readily available in the system. The utilization of a DHT permits every node in the system to make decisions about how to run tasks, execute them in a certain order defined by a scheduling policy, and make this without any central authority or any user intervention.

In the next section we review a collection of DHT systems that could be valid for our purposes, and justify the choice of a particular one: Cassandra.

2.2 P2P storage systems

We propose to use a DHT system to store the information about the tasks in our HTC system. But what are the characteristics required from a DHT in order to consider it adequate for our purpose? We have stated that, in order to build a distributed queue for our tasks, we have to include in the design of the DHT some fault tolerance features, which are usually achieved via replication. Consequently, we also need coherence protocols to maintain the replicas synchronized, and failure detectors to detect which replicas are off-line.

One of the first production systems integrating all these techniques was Amazon's Dynamo [16]. Dynamo is a key-value storage system based on Distributed Hash Tables which assures lookups in constant time and, due to its replication and quorum techniques, provides high levels of scalability and availability, with eventual consistency [25]. Brewer's CAP theorem [26] (for Consistency, Availability and Partition tolerance) states that, in distributed systems, consistency, availability and partition-tolerance cannot be achieved at the same time. So Dynamo designers decided to limit the consistency of the system providing *eventual consistency* and gaining in system stability. Note that the "eventual consistency" term means that all updates in the system reach all replicas after some (undefined) time.

Dynamo has been the origin of a collection of systems based on it. Some of them are key-value stores, such as Riak [27] or Scalaris [28]. Others are document oriented, such as CouchDB [29], and others are wide columns stores such as Hadoop File System [30] or Cassandra [17]. In all of these systems it is possible to implement our target distributed queue. We have chosen Cassandra to build our prototype for several reasons. According to different benchmarks from Yahoo [31], Cassandra has smaller read/write latency than other storage systems designed for *cloud computing*. We can also highlight its tunable consistency level and the richness of its data model, which enables several implementations of a queue. Other systems provide strong consistency, but at the cost of serializing writes, or under the assumption of reliable environments. With Cassandra we will have to deal with eventual consistency, but we will have a robust, scalable storage system.

2.3 Cassandra

Cassandra was developed in 2008 by Facebook to improve its Inbox Search tool. It was released to the Apache Foundation in 2009. It provides consistent hashing/order preserving partitioning, a gossip-based algorithm to control membership, replication, anti-entropy algorithms to maintain consistency between replicas and failure detection. These features give us a totally distributed, fault tolerant and symmetric storage system with eventual consistency. In this section we explain the general characteristics of this storage system, just focusing on its data-model; however, in the next chapter, a deeper study of Cassandra's replication model will be made introducing stochastic models for predicting availability.

Cassandra is a partitioned row store, where rows are organized into tables, named *ColumnFamilies*. Each row is composed of several columns and is indexed by one or more of these columns, composing the primary key of the table. This primary key is divided into two parts: the partition key and the clustering key. The first part is called partition key because it determines in which node (*partition*) a row is going to be stored, and can be formed by one or multiple columns. The partition key is used in combination with a hash function, the *partitioner*, to determine the node in which the row is going to be stored and to define an order among rows in each table. Cassandra permits to define customized partitioners, but several built-in functions are available:

- Random Partitioner: it uniformly distributes data across the system by using MD5 hashes of the keys. The possible range of hash values is from 0 to $2^{127} - 1$.
- Murmur3 Partitioner: uniformly distributes data across the cluster based on MurmurHash hashes. This hash function is known to be faster than MD5. The possible range of hash values is from -2^{63} to $+2^{63}$
- ByteOrdered Partitioner: keeps an ordered distribution of data lexically by key bytes. This partitioner is not recommended by Cassandra's designers because it may yield an unbalanced distribution of data and can cause hot spots when performing sequential writes.
- OrderPreserving Partitioner: keeps a lexically ordered distribution of data assuming that keys are UTF8 values. As the ByteOrdered Partitioner, this partitioner can yield an unbalanced distribution of data so it is not an recommended.

The clustering key, formed by zero or more columns, is in charge of grouping and ordering the different rows (stored in the same partition) with the same partition key. These rows will be stored on disk respecting the order defined by the clustering key. When querying for rows in different partitions, Cassandra can use the clustering order to retrieve the results in order. In the first part of the query, already sorted rows from each partition are gathered by the node performing the query. In the last part of the query, a final sorting is made before returning the results to the client. This sorting is performed by using the Mergesort algorithm [32]. Note that if the client wants to order results by clustering key, Cassandra requires to specify in the query which partition keys are requested. We could make a more general query, without specifying the partition keys, but Cassandra would not be able to order the results by the clustering key.

In Tables 2.1 and 2.2 we can see two examples of tables with different numbers of columns acting as primary key (columns in bold represent those columns included in the primary key). In these examples we have not defined clustering keys, so the entire primary key is used as the partition key (in bold and italics in the examples). In the first example, with only one column acting as the primary key, we represent a table containing information about several songs in a music service. Each song has a unique identifier that is used as the primary key for the table. In the second one we can see an example of a table with a composed partition key. The reason for choosing this kind of key is in situations in which the data is not identified uniquely by a column and we want a complete distribution of the rows (each row in a different node). In the second example we show a table storing information about albums in which we use the combination of "Name" and "Artist" to identify uniquely each album.

Songs				
ID	Title	Album	Artist	Data
song$_1$	Bohemian Rhapsody	A night at the opera	Queen	...
song$_2$	Palabras para Julia	En el Olympia	Paco Ibáñez	...
song$_3$	Where is my mind?	Surfer Rosa	Pixies	...
⋮	⋮	⋮	⋮	⋮

Table 2.1: Example of ColumnFamily representing a list of songs.

Albums			
Name	***Artist***	Songs	Year
A night at the opera	***Queen***	12	1975
Master of Puppets	***Metallica***	8	1986
The Man Who Sold the World	***David Bowie***	9	1970
⋮	⋮	⋮	⋮

Table 2.2: Example of a ColumnFamily representing a list of albums.

As an example of the use of clustering keys, we can imagine a table to store multiple playlists in a music service. Each row represents one song of a certain playlist; however, we would like to maintain some order among songs so, when a playlist is requested, each song is retrieved in order. Therefore, we have to include an extra column identifying the order of songs in the playlist. The partition key of the playlist is an unique identifier per playlist while the clustering key is the order of songs per playlist. In Table 2.3 we can see a representation of this playlist table. The column in bold and italics is the partition key while the column just in bold is the clustering key. Both columns form the primary key. Note that if we used the “order” column as part of the partition key instead of the clustering key, we would not be able to request the different elements of a certain playlists in order (in this case the order would be imposed by the partitioner function used).

Playlists			
Playlist ID	**Order**	Song ID	Owner
playlist$_1$	1	song$_1$	user$_1$
playlist$_2$	1	song$_1$	user$_1$
playlist$_1$	2	song$_2$	user$_1$
playlist$_2$	2	song$_3$	user$_1$
⋮	⋮	⋮	⋮

Table 2.3: Example of a ColumnFamily representing several playlists.

As can be seen, the main criteria used in Cassandra to access data is the primary key defined per table. However, we can also define secondary indexes over any non-primary column, which leads to a fast and efficient lookup of data matching a certain condition. These indexes are stored in a distributed way as each node is in charge of locally indexing the rows it stores. Cassandra updates them automatically but they must be created manually by the user. These indexes permit us to make queries on tables searching for specific rows that match one or more criteria. For example, if we define a secondary index for the “Artist” column of the table “Songs” (see Table 2.1) we could use it to search for all the songs published by a certain artist with a single request. Moreover, in order to fix conditions on a particular column, a secondary index must exist for that column.

After presenting the main characteristics of Cassandra and its data model, we now describe how our proposal of HTC-P2P system has been implemented over Cassandra using these characteristics.

2.4 Design of the HTC-P2P system over Cassandra

We introduce first the expected functionalities of our proposal of HTC-P2P and, secondly, how they are implemented on top of Cassandra.

2.4.1 Functionalities of the proposed HTC-P2P system

The proposed HTC-P2P system is composed of workers that execute both Cassandra and the local task scheduler. These workers are in charge of maintaining the structure of Cassandra and executing tasks in a certain order. In this chapter we use a non-strict FCFS order, but others are possible. Particularly, we will describe additional scheduling policies in Chapter 4.

To execute a task, the worker accesses the files associated to that task stored in Cassandra. Once these files are available, the node executes the task, storing the results again in Cassandra.

Users can connect to any worker in order to insert tasks and manage several aspects of the HTC-P2P system, such as their workers (each user may add one or several workers to the system), files and a list of their banned users (users whose tasks will not be executed by the banner's workers). This is possible thanks to the utilization of a set of commands available in each worker.

These are the main features of the proposed HTC-P2P system:

- The default scheduling policy is a non-strict FCFS order. Task execution order depends on the availability of appropriate workers.
- Each user can upload files that are stored in a file system implemented on top of Cassandra. These files are used by the user's tasks as executables and/or input files. The outputs of Tasks are also stored in the HTC-P2P system as files.
- There is a blacklist system permitting users to ban other users.
- Workers survey other workers executing tasks, in order to detect failed nodes and relaunch failed executions. This surveillance is made through a heartbeat system.
- Any worker can be used as the entry point to the HTC system.
- Users can:
  - Insert tasks from any worker.
  - Manage the state of their tasks.
  - Retrieve the results of tasks.
  - Manage their files.
  - Manage their black lists.

The operation of our system is as follows: when a user wants to submit a new task from a worker node, it inserts the task's relevant data into Cassandra with state "Waiting" with a unique identifier. Additional files needed by each task are stored into a distributed file system implemented on top of Cassandra. Finally, it enqueues the task into the queue. After that, the user is free to disconnect from the system: no additional information is required from the submitting node. The results of executing a task are again stored in Cassandra, ready for downloading when the user decides to rejoin the HTC system.

As previously mentioned, at each node there is a scheduling process in charge of selecting tasks to run. When a node is idle and wants to execute a task from the queue, the scheduler looks up the queue until it finds an appropriate task, changing its state to Running and dequeuing it. After that, it executes the task locally and, when finished, stores in Cassandra all the results obtained from the execution. Finally, the task's state is changed to "Finished".

During the execution of a task, the worker stores *heartbeats* periodically in Cassandra. This allows other workers to survey the busy workers in order to detect the failed ones. If one failed worker is detected, the tasks being executed in that node are reinserted into the queue for re-execution. This process of surveying other nodes is performed periodically by every alive node in the system. In Figure 2.1 we can see the different states in which a task can be.

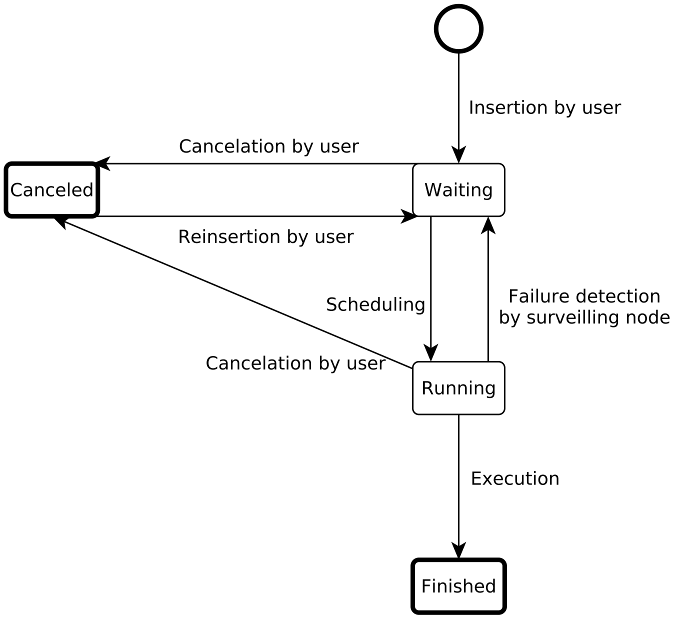


Fig. 2.1: Workflow of states of tasks .

Note that the attributes of a task must match the ones from the worker that wants to execute it. This process is called *matchmaking* and is in charge of searching for good fits between tasks and workers. In other HTC systems, for example HTCondor, this process is carried out by using an advertising system where workers and tasks publish their characteristics. The manager of this advertising system compares the different ads in order to pair tasks with suitable workers. In our proposal each worker performs matchmaking during the scheduling of tasks. Each task has some attributes that are indexed using secondary indexes. These indexes are used by the workers to quickly search for suitable tasks. This way, workers can make a simple request to retrieve all the waiting tasks that match their characteristics. We will talk later about how we have implemented matchmaking over Cassandra. We now present how Cassandra supports the necessary data structures for this HTC-P2P system.

2.4.2 Implementation

This proposal has been implemented using Cassandra version 1.2, with the Cequel Cassandra Ruby client library version 1.2 [33]. Here are the different tables required to implement the aforementioned features. The implementation of these tables in CQL, the Cassandra Query Language [34], can be found at the end of this dissertation as an annex.

- **Tasks**: This table performs as the main store of tasks of the system. It stores the different attributes of each task together with the state of the task (See Figure 2.1). When a new task is inserted into the system, it is stored in this table with the state "Waiting". After that its attributes, for matchmaking purposes, are inserted into the **Queue** table. The primary key of this table is the unique identifier of each task[1]. In addition to the attributes and the state already mentioned, this table includes a column to store error messages and information about insertion, execution and finalization times.
- **Queue**: This table acts as the main queue of the system. Each row stores information needed by a task in order to be successfully executed, such as its attributes together with the task owner's identifier. The row primary key is composed by a number obtained from the insertion time of the task, which we call "bucket", used as the partition key and the ID of each task as the clustering key. The combination of both columns, the bucket number and the ID, permits us to distribute the queue along all the nodes of the system and to maintain a certain order among tasks. We will explain how we maintain an ordered queue using these attributes and how we obtain this "bucket" number later in this section. This table has the following attributes, some of them indexed via secondary indexes to perform the matchmaking between tasks and workers:
  - bucket, used as partition key.
  - task_id: the ID of the task.
  - user_id: the ID of the owner.
  - proc: the type of processor needed.
  - mem: maximum amount of memory needed by the task.
  - so: operating system required by the task.
  - disc: required disk space.
  - libs: required libraries.
  - tins: insertion time.

[1] a Universally Unique Identifier version 1 [35], or UUID, which is a 128-bit value formed from a timestamp and the MAC address of the node generating it.

- **Users**: This table stores information about each user in the system. The attributes of each user are its ID, user name, complete name, email address, and a password digest required for authentication purposes. The primary key of this table is the user ID.
- **Workers**: This table stores information about each worker in the system: its owner, hostname, processor, memory, operating system, disc space and installed libraries. It also includes a *heartbeat* attribute used by other workers in the system to detect failed nodes in order to restart any work being executed by them. The primary key of this table is the worker ID.
- **Files**: This table is used to store the metadata of files in Cassandra. Each row represents a file and its attributes, such as the file's owner, the name of the file, the path of the file, the size and the number of chunks in which the file is distributed. The primary key of the table is the file ID.
- **Chunks**: Each file stored in Cassandra is split into several chunks, each one stored individually. Each chunk has a chunk ID, the ID of the file to which it belongs, the size of the chunk, a SHA1 checksum and the data stored in the chunk. Its primary key is the chunk ID.
- **File_chunks**: This table stores a record of the chunks belonging to a file. In this table, the file ID is used as the partition key while the chunk ID is used for clustering purposes.
- **Task_files**: This table stores a list of the files belonging to a specific task. Each row contains the task and file IDs, and the file type (input file, binary file or output file). The task ID is used as the partition key and the file ID as the clustering key.
- **Blacklist**: This table implements the blacklist system. Each user may define which users are banned. If a certain user is banned by another user, her/his tasks will not be executed in nodes belonging to the banner. The primary key of the table is a combination of a user ID and the blocked user ID.
- **Task_workers**: This table's purpose is two-fold: first, it permits the system to know which worker has executed a task and, secondly, it permits workers to compete in order to decide which worker will execute the task. Each row has an attribute, *score*, which is a timestamp in order to give priority to the first worker interested in the task; however, it could be replaced by any score permitting the system to implement different scheduling policies (this issue will be contemplated in Chapter 4). The primary key of the table is formed by the task ID, the score and the worker ID.

With respect to how we have implemented a distributed queue in Cassandra, there are several ways to use its data model to store a queue. The approach we have chosen assures the complete distribution of the elements stored in the queue while allowing their in-order retrieval. There are two main methods to query for rows in a table and retrieve them in order. The first option is to use an order-preserving partitioner. However, this solution may lead to an unbalanced distribution of data and to the creation of hot spots. However, using a non-ordered partitioner, although guaranteeing the randomization of the data placement across the system, impedes the retrieving of rows in a certain order.

Another solution consists of using a non-ordered partitioner with a clustering key for ordering purposes. If we specify the partition keys required in each query, Cassandra can use the clustering key to obtain the rows from different partitions in order. Therefore, we decided to implement our distributed queue using a non-ordered partitioner, particularly the Random Partitioner, and make groups of tasks, which we call "buckets", that are W seconds close in time. Each task in the queue is identified by two values: its "bucket" number, built from the insertion time, as the partition key and a UUID (built also from the insertion time), as the clustering key. The partition key leads to a complete distribution of task groups, buckets, across the system. Each bucket is placed in a different point of the system because of the non-ordered partitioner. The clustering key permits us to order the tasks in the queue by timestamp. In the case that several tasks are inserted in the same instant of time, and thus with the same timestamp and bucket number, the UUID can be used to distinguish from each row as is generated from the MAC address of the node and a random sequence.

To calculate the number of bucket, b, for a certain task, the time of insertion, t_{ins}, is used together with the size of the bucket, W, and the number of buckets N which is a parameter known by every node in the system. Equation 2.1 summarizes how b is obtained from the insertion time.

$$b = \left\lfloor \frac{t_{ins}}{W} \right\rfloor modN \tag{2.1}$$

The distribution of the different rows in the system depends on the number of buckets, N, and the number of nodes of the system. Therefore, if there are more buckets than nodes in the system, the queue will be totally distributed and this query requires accessing every single node in the system, where each access has constant cost. This implementation of the queue implies that, if some of these nodes have failed and do not respond to the query, only the buckets stored in those nodes will be affected and the system will still return an answer to the query. With respect to the W parameter, it could be adjusted according to the frequency tasks which are submitted to the system.

So, if we have a queue using the implementation described above, with N buckets of size W, the CQL query in Listing 2.1 can be used to obtain the first n elements from the queue. Note that, even if we are requesting all the elements of the queue, the "WHERE IN" clause is required by Cassandra in order to use the "ORDER BY" modifier, which is what Cassandra uses to order results by means of the clustering key.

```
SELECT * FROM Queue
WHERE bucket IN (0, 1,...,N-1)
ORDER BY task_id
LIMIT n
```

Listing 2.1: Example of CQL query to obtain the first n elements from the queue.

As we have already mentioned, each row in the queue has several attributes. These attributes characterize each task and can be indexed via secondary indexes, permitting us to query the system searching for specific tasks that match some attributes (matchmaking). For example, a worker node with a “power64” CPU, 2 GB of RAM, “Linux” as operating system and 20 GB of free disk would execute the following query in CQL to obtain the first 100 tasks that fit its attributes in a system with 5 buckets:

```
SELECT * FROM queue
WHERE bucket IN (0, 1, 2, 3, 4)
AND so = 'Linux'
AND proc = 'power64'
AND mem <= '2'
AND disc <= '20'
ORDER BY task_id
LIMIT 100
```

Listing 2.2: Example of CQL query to obtain the first 100 elements from the queue matching several attributes.

While developing our system, we found that the eventual consistency provided by Cassandra caused some problems. As previously mentioned, Cassandra uses replication in order to increase fault-tolerance. As Cassandra does not provide total consistency for each operation, depending on the consistency level used (the number of replicas required to answer correctly to a certain operation) the write operation takes some time to reach all the replicas and, thus, subsequent reads could obtain stale data. Also, there are no atomic transactions in Cassandra: the process performed by each worker to claim a task requires several write and read operations. This, in combination with the eventual consistency, can lead to several workers executing the same task. We call this situation a *collision* between workers.

As a preliminary solution, we decided to use the **Task_workers** table as an extra queue to order the different nodes interested in a task. To obtain all the possible workers interested in a particular task ordered by its score, the following query is used:

```
SELECT * FROM task_workers
WHERE task_id = 'task_1'
ORDER BY score
```

However, this approach only reduces the probability of a collision, because the same consistency problem occurs when writing in this table. A possible solution could be the use of an external locking mechanism, such as Zookeeper [36], a Yahoo project that implements consensus. The original Cassandra used Zookeeper as the locking mechanism, but this functionality was removed when it was released to the Apache Foundation. Zookeeper could be used to lock (serialize) the dequeuing of tasks avoiding more than one node to execute the task. However, this solution would reduce Cassandra's fault tolerance because the entire system's fault tolerance would depend on the Zookeeper server or servers.

We measured that our preliminary solution is able to reduce collisions from 30% to 0.5% (in a 30-node system). At the moment, we find this an acceptable overhead. However, if a collision happens, as each worker executing a certain task uses different rows to store the output from the execution, we can just use one of the results and discard the remaining ones.

2.5 Evaluation

Although the design that we present in this chapter is based on Cassandra 1.2, our first prototype of the scheduler was developed with Cassandra 0.6. Using this prototype, we performed several experiments to check the validity of the system and measure its scalability.

Cassandra 0.6 has several differences and limitations, compared to Cassandra 1.2: (1) limited implementation of clustering keys, (2) limited capacity to order rows by clustering key, (3) lack of secondary indexes and (4) the use of a different client protocol. Due to the limited capacity to order rows in Cassandra 0.6, the ordering of the different rows between buckets must be done in the client or using an ordered partitioner, which is not recommended. Therefore, the implementation of the distributed queue in this prototype was made using only one bucket. As each bucket is placed by the partitioner in one node and replicated in $n-1$ additional nodes, this prototype has a smaller fault tolerance than one using N (> 1) buckets. Finally, this prototype did not implement matchmaking (every node could execute any task). However, we still think that the results obtained can be valuable.

In our experiments we have used a workload generator [37] that generates synthetic workloads for HTC clusters, based on probability distributions. Briefly, this generator produces a sequence of tasks with different execution and arriving times. Varying these two parameters, the duration of tasks and the frequency of the insertion of tasks, we can test our system under different utilization scenarios.

During the experiments we have measured these performance figures:

- Bounded Slowdown: this is the waiting time of a task plus the running time, normalized by the latter, and eliminating the emphasis on very short tasks by means of putting in the denominator a commonly used threshold of 10 seconds. In Equation 2.2 we can see the definition of bounded slowdown, being w the waiting time of a task and r its running time. Regarding this value, note that in persistent saturation scenarios (those in which the needs of the arriving tasks exceeds the capacity of the system), it grows limitlessly.

$$bounded\ slowdown = max\left(1, \frac{w+r}{max(10,r)}\right) \tag{2.2}$$

- Node utilization: the proportion of time devoted by nodes to execute tasks. Expressed as a value between 0 and 1, or as a percentage.
- Scheduling Time: the time needed by an idle node to schedule a waiting task.
- Collisions: the number of tasks in each workload that have been executed by more than one worker.

We have tested our HTC-P2P system in two real setups. The first one uses a cluster with 20 nodes, running on each of them Cassandra version 0.6 and Ubuntu Server 8.10. Each node features an Intel Pentium 4 3.20 GHz with 1.5 GB of RAM and 80 GB of hard drive. The workload generator has been adjusted to generate 20 tasks per node globally. Experiments last the time required to consume all the generated tasks, and have been designed mainly to assess the behaviour of our system under different workloads. Each test has been repeated 10 times.

The second set of experiments has been carried out using different clusters of sizes 10, 20, 40 and 80 nodes. These clusters were built using Amazon's Elastic Cloud Computing platform [38]. Each node has one core, 1.7 GB of RAM and 160 GB of hard drive, running the same software described for the first scenario. These tests have been designed to measure the scalability of our proposal. In this case each test has been executed a minimum of 2 and a maximum of 5 times due to cost restrictions and after confirming that measured figures from different runs show small variability.

Figures 2.2 to 2.4 summarize the results of the first set of experiments. In these experiments the maximum task length goes from 2 to 150 seconds, and load varies from 0.2 to 2.0 tasks/second per node. Figure 2.2 shows the measured system's bounded slowdown for these load scenarios. Figure 2.3 shows system's usage. In order to analyse system performance both figures should be studied together. The bounded slowdown is very small in almost all the situations. In the top-right corner it becomes the highest; this area corresponds to a heavily loaded system, in which arriving tasks have to wait for nodes to become available after processing those tasks already in the system. The higher the load, the longer is the waiting time. The increase in the bounded slowdown is, therefore, a consequence of the system's load, but not an effect of a bad scheduling.

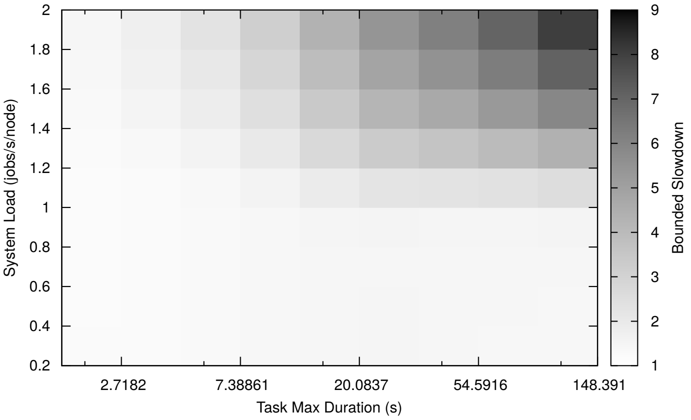


Fig. 2.2: Bounded Slowdown (average of 10 runs). 20-node cluster. System load: 0.2 - 2.0 tasks/second/node. Task maximum duration: 2 - 150 seconds

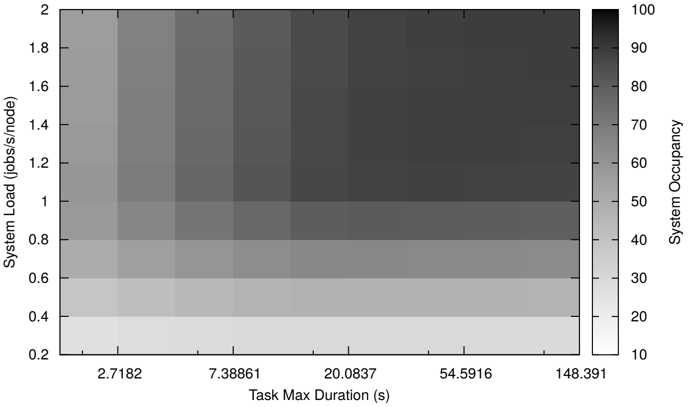


Fig. 2.3: Node utilization (average of 10 runs). 20-node cluster. System load: 0.2 - 2.0 tasks/second/node. Task maximum duration: 2 - 150 seconds

Figure 2.4 illustrates the collision problem explained in Section 2.4. We have measured how often a task is selected to be run by two or more workers. For loads higher than 1.0 tasks/second per node this chance is lower that 1%, independently of the execution time per task. The number of collisions rises to more than 5% for low values of load. Obviously, the more idle nodes there are in the system, the more likely it is that two nodes try to execute the same task. As previously mentioned, this percentage is higher than 30% when our partial solution is not used. This indicates that our proposal to tackle the eventual consistency problem works well.

Scalability tests have been carried out using the following parameters of the workload generator. The maximum task size was fixed to 150 seconds.

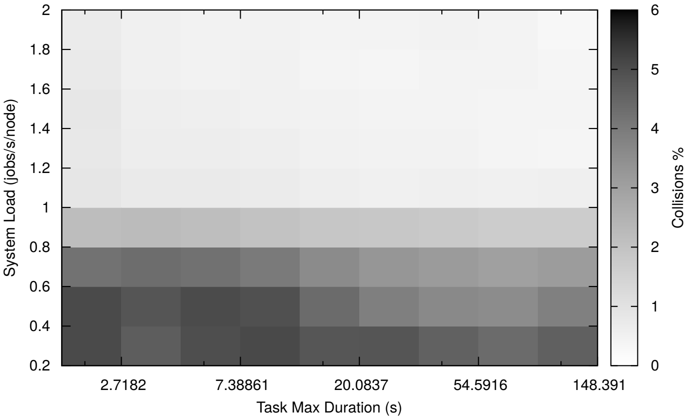


Fig. 2.4: Percentage of Collisions (average of 10 runs). 20-node cluster. System load: 0.2 - 2.0 tasks/second/node. Task maximum duration: 2 - 150 seconds

Load varies from 0.2 to 2.0 tasks/second per node. Experiments were run for cluster sizes 10, 20, 40 and 80.

In Figure 2.5 we can see the node utilization for the four different sizes. Notice how at load 1.0 tasks/second per node, utilization is near 85%, for all sizes. In Table 2.4 we can see a summary of node utilization together with the (per-node) deviation of this figure. At low loads deviation is higher than 7% (meaning that it is highly probable that some nodes are idle while others are busy), but when load increases, deviation decreases to less than 4% for any system size. Therefore, for all cluster sizes tested, the system is able to accept a high load level and to balance it evenly among participating nodes.

In Figures 2.6 and 2.7 we can see scheduling times and percentages of collisions for different system sizes. If we focus on the scheduling time, we can see that it is kept below 0.04 seconds for small systems (10, 20 nodes), and below 0.06 for larger sizes (40, 80 nodes) – but in this case only for highly loaded systems. The scheduling time is related to the cost of accessing Cassandra. The fact that the scheduling process takes small times, smaller as the system load rises, proves that our HTC-P2P system scales well with the system size thanks to the scalability of Cassandra.

The percentage of collisions shows a similar trend: it is higher at low loads (as expected), and decreases substantially at higher loads, stabilizing around 0.5%. We can state, therefore, that our way of dealing with the eventual consistency problem scales well with system size as well as load.

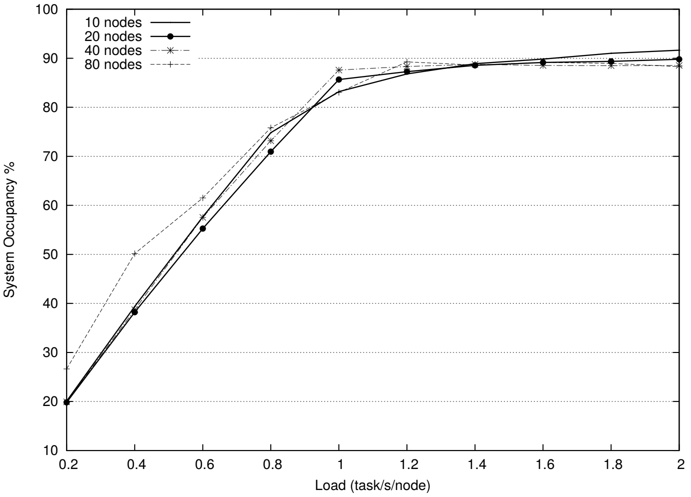


Fig. 2.5: Average node utilization (in percentage). Variable size EC2 cluster. System load: 0.2 - 2.0 tasks/second/node

Node Utilization								
Load	**System size**							
	10		20		40		80	
	Mean	σ	Mean	σ	Mean	σ	Mean	σ
0.2	20.1	6.93	19.8	7.65	20.0	8.5	26.7	10.4
0.4	39.2	5.89	38.1	11.9	38.6	12.2	47.5	15.8
0.6	57.7	3.97	55.5	8.48	56.4	9.57	65.5	15.4
0.8	74.7	5.69	71.0	6.69	73.2	6.63	77.8	7.76
1.0	83.1	3.99	85.8	3.49	87.8	3.48	84.4	9.28
1.2	86.6	3.49	87.5	3.92	87.7	3.71	89.1	9.24
1.4	88.8	3.03	88.7	4.05	87.7	3.49	89.0	3.79
1.6	89.9	3.75	89.4	3.32	87.9	3.45	89.1	3.65
1.8	90.8	3.51	89.1	3.58	87.6	3.71	88.8	3.60
2.0	91.5	3.56	90.2	3.66	87.8	3.33	88.4	3.98

Table 2.4: Node utilization in percentage. Node average and deviation, for different system sizes and values of applied load

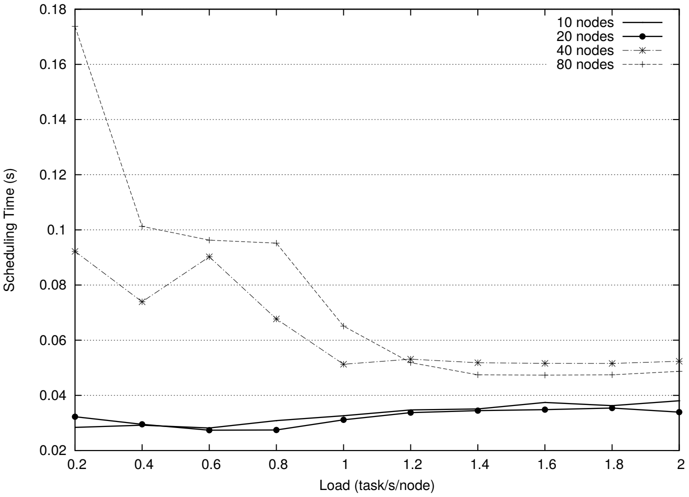


Fig. 2.6: Scheduling time for different system sizes. EC2 clusters. System load: 0.2 - 2.0 tasks/second/node

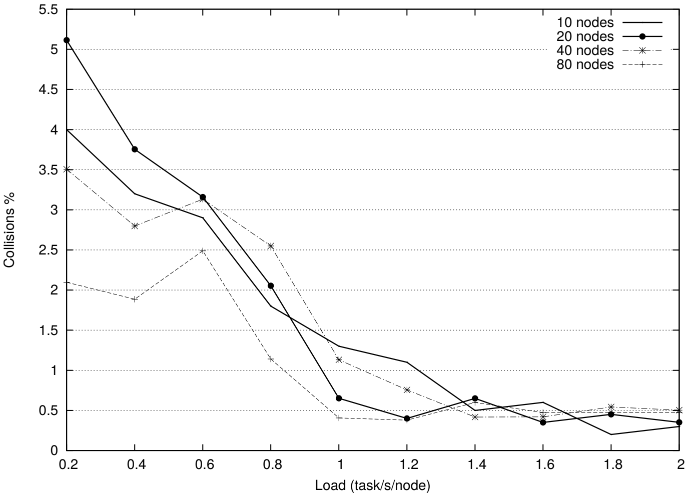


Fig. 2.7: Percentage of collisions for different system sizes. EC2 clusters. System load: 0.2 - 2.0 tasks/second/node

2.6 Related Work on P2P Computing

In the literature we can find several proposals of P2P computing systems. Next we briefly describe some of them, paying attention to the way they fulfill or fail to fulfill the desirable properties described in Section 2.1.

Some proposals fail to meet the first property: they use some centralized mechanism for task scheduling. CompuP2P [39] is based on a DHT which divides the nodes set between resource sellers and buyers. A *leader* node is chosen, which takes care of the resource market. In [40] Chmaj et al. present a system based on a structured overlay in which one of the nodes has the master role, the tracker. This tracker, similar to the concept of tracker in the BitTorrent protocol [41], is needed in order to schedule computing tasks and distribute the files needed by the nodes.

A common model used for P2P computing systems is the *super-peer* model, in which a group of nodes, the super-peers, form a P2P overlay. Workers are connected to super-peers, which are in charge of scheduling tasks. When a user wants to run any task, he asks nodes in the super-peer overlay to search for idle workers. Examples of this model are JACEP2P-V2 [42] and Mining@home [43]. A similar idea is implemented in CoDiP2P [44], which is based on a tree-structured P2P network. It is built with the JXTA Java library, used to construct non-structured P2P networks. In the tree structure, nodes are differentiated between masters and slaves, and grouped in tree regions. In each region, there is a master node in charge of the scheduling procedure. The middleware is written in Java, and it can only execute Java programs linked to a library developed by the CoDiP2P team. All these P2P systems have several manager nodes, which could pose advantages in terms of scalability. Still, they are not fully distributed: failures in a master node would negatively affect, if not the whole system, parts of it.

WaveGrid [45] consists of a DHT overlay in which nodes are divided by time-zones, so that, at a particular time, only the nodes in the *night* zone are used. When the sun rises, tasks migrate to another time-zone still at night. In this system the launcher node is the one in charge of choosing the execution node; therefore, it should be permanently connected.

In [46, 47, 48, 49, 50] DHTs are used to store information about idle nodes and resource claims in order to improve Grid systems. In these systems, tasks are assigned to nodes in the overlay network using the DHT routing capabilities. These nodes will be in charge of scheduling the tasks using the information stored in the DHT to search for idle nodes, using a matchmaking algorithm that performs range queries on the DHT over several criteria. This has been widely studied in the literature [51, 52, 53, 54, 55, 56]. These systems can be considered as fully distributed P2P computing systems, but they fail to meet the third criterion: they lack any mechanism to ensure that tasks are being executed in a particular order, or even that they will be eventually executed.

2.7 Conclusions

In this chapter we have described an HTC system based on P2P networks. Our goal was to build a totally decentralized HTC system based on highly scalable and reliable P2P storage, maintaining these target characteristics: (1) lack of central management points, (2) disconnected operation (3) and flexible scheduling with support for (non-strict) FCFS execution order of tasks. For this purpose, we have implemented a distributed queue system that manages the execution of tasks over a group of nodes in a totally decentralized way. We have explained the main characteristics of our design, and how it has been implemented based on the Cassandra P2P storage system. We have carried out some experiment sets to assess system behaviour, confirming that it works as expected.

Some issues remain related to the eventual consistency provided by the DHT storage. A mechanism to reduce it to manageable levels has already been proposed and tested: the chance of collision is around $0.5-6\%$ depending on the system load. Cassandra is still evolving quickly. In more recent versions of Cassandra, 2.0 and above, a version of the Paxos consensus algorithm [57] has been implemented, permitting limited per-row transactions. Using this type of transactions, the problem of collisions could be solved, but at the expense of higher latency and lower fault-tolerance to perform the claiming operation. For the moment we have not tested this possible solution, but we plan to do so in future releases of our HTC-P2P system.

The actual behaviour of our system under failure scenarios has not been tested exhaustively. In Chapter 3, we have modelled the availability of Cassandra. These models could be used to predict the availability of our HTC-P2P system under several failure scenarios, however we have yet to carry out extensive experiments measuring how the failures affect our HTC-P2P system.

We plan to enhance the HTC-P2P system by adding tools to simplify its utilization, allowing users to submit not only individual tasks but also more sophisticated workflows, such as those based on acyclic data flows (applications that consists of a series of distinct jobs, where each task reads data from stable storage and writes it back to stable storage) or multi-pass applications, those that include iterative algorithms or streaming applications. Finally we have plans to include mechanisms to allow task checkpointing in order to allow higher degrees of fault-tolerance and load balance via task migration.

3

Modeling the availability of Cassandra

Recent works on Distributed Hash Tables (DHT) [10], have attracted attention because of the possibilities that this type of systems offers in terms of availability, scalability and fault tolerance. One of these proposals is Cassandra [17], a non-relational database system, proposed by Facebook and based on the replication model of Amazon's Dynamo [16]. This replication model is considered to be highly scalable, providing a high availability. However, we could ask ourselves how reliable these systems are. If we consider actual systems serving thousands of requests per unit of time, even small gaps of data unavailability could correspond to large amounts of lost revenue.

In this chapter we study the capability of Cassandra, the P2P storage system in which we base the HTC-P2P system described in Chapter 2, to deal with node failures. To this extent, we have developed analytical models of the availability of Cassandra when confronting two different types of failures that resemble typical malfunction situations:

- Transient failures: those which imply the recovery of a failed node after some time, without any loss of the already stored information. However, note that these nodes will lose updates during their off-line period. This type of failures can be caused by network or power outages and also by node disconnections caused by users (churning).
- Memory-less failures: these would include a total crash of a node, losing the stored information. The node has to be totally or parcially replaced (for example, substituting a broken hard disk) and the data that should be stored in that node must be recovered.

The models we propose are based on the stochastic modeling of replica groups using Markov chains [58]. Markov chains are stochastic processes that obey the Markov property, which establishes that future states of a system depend only on the current state but not on the previous ones. More information about these mathematical models can be found in [59].

As Cassandra's replication model is based on Dynamo, these models could also be applicable, with some effort, to the latter one as well as to other systems based on it, such as LinkedIn Project Voldemort [60] or Basho Riak [27]. The first model we present depends only on the replication strategy of Cassandra, so it is directly applicable to these systems. The second model comprises the reparation processes implemented in Cassandra, which could be different in each of these systems so, in order to adapt the model to them, their reparation processes should be modelled properly.

In order to validate the proposed models, we have run different experiments on a real Cassandra cluster simulating failures and measuring system availability while performing sets of I/O operations. We define the system as available when it successfully replies when an operation is requested. We compare the availability predicted by our models with that measured on the real system, and conclude that our model accurately describes the availability of Cassandra.

The remainder of the chapter is structured as follows. Section 3.1 presents an overview of Cassandra and its main characteristics. Section 3.2 explains the models that approximate Cassandra's behavior. In Section 3.3 we carry out a set of experiments on a real system with the aim of comparing the results obtained with the availability provided by the theoretical models. In Section 3.4 we can find several use cases showing how the models can be used to analyze a variety of applications based on Cassandra. Section 3.5 is devoted to provide a summary of the related work about failure models in P2P networks. In Section 3.6 we present some conclusions and a discussion of future lines of work.

3.1 An overview of the architecture of Cassandra

Cassandra is a distributed storage system based on the concept of Distributed Hash Table (DHT) [10] that uses data replication in order to improve availability. As said in Chapter 1, a DHT is a structured peer-to-peer system which stores {key, value} pairs. Most of the existing DHTs use a look-up algorithm which permits the system to reach any piece of data with a complexity of $O(\log m)$ hops, where m is the number of nodes in the system. This is due to the use of routing tables with partial information, usually with $O(\log m)$ entries, obtaining a good trade-off between time and space costs. However, in Cassandra, nodes are able to reach any point of the system in constant time ($O(1)$ complexity) at the expense of having routing tables of size $O(m)$ at each node.

In Cassandra, the data model is slightly different from a typical DHT. Instead of storing single values, each value is an structured tuple named Column Family. Each column of the tuple has a name, a value and a timestamp. This data model, proposed in [61] for Google's BigTable, is becoming a popular alternative to the relational model in contexts where scalability and fault tolerance are critical.

All the nodes in Cassandra are peers; all have the same roles and responsibilities. Nodes are virtually interconnected in a one-directional ring fashion. In this ring, each node is identified by a token, which is used to calculate its position in the ring. The token may have different lengths depending on the partitioner used (see Section 2.3 in Chapter 2), but usually it is a 128-bit string, which allows a maximum of 2^{128} keys.

Values are stored as follows: each value is identified by a key; this key is transformed into a 128-bit string using a strong hash function, MD5 if the system uses the Random Order Partitioner. This string is then used to select the node that stores the value, usually the one whose token is the nearest counterclockwise in the key space. As Cassandra uses replication, there are n replicas for each object stored in different nodes. Therefore, for each stored object, Cassandra selects a group of $n - 1$ additional nodes, usually those following the first replica node along the ring. Cassandra can also be configured to carry out a different distribution of replicas. In Figure 3.1 we can see an example of a Cassandra system composed of 4 nodes, where 4 objects have been stored in the system with a replica factor of 3. Observe how the first replica of each object is stored in each of the nodes and how the two subsequent replicas are distributed in the following nodes of the ring making up four replica groups, one per object, being a replica group the group of nodes that hold all the replicas of one or more objects.

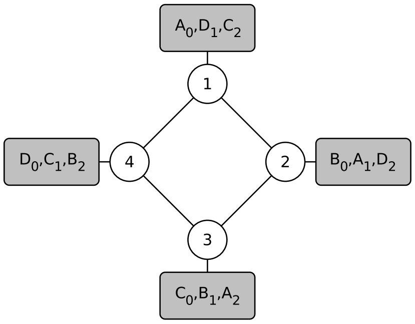


Fig. 3.1: Example of replica distribution in a system of 4 nodes with replica size of 3.

In order to provide a high availability and partition tolerance, Cassandra provides eventual consistency, which means that updates in any object takes some time to reach every replica in the system. As the data is replicated, the latest version of some object is stored on some node in the cluster, however older versions are still on other nodes and can be sometimes obtained until the latest version reaches all the nodes. This eventual consistency is tunable per operation by selecting the number of replicas, denoted by k, required to complete an operation satisfactorily. When performing read and write operations over Cassandra, a consistency level in Cassandra's terminology is chosen. In the list below, we define the consistency levels most widely used:

- ONE: only one node is required to reply correctly.
- QUORUM: $\lfloor \frac{n}{2} \rfloor + 1$ nodes must reply.
- ALL: all the n nodes participating in the replica group must reply.

The combination of these consistency levels for read and write operations will determine the time required by the system to reach a consistent state. According to [62], when the number of replicas queried for reads (r) plus the number of replicas queried for write operations (w) is larger than the total number of replicas (n), or $r + w > n$, we say that Cassandra has strong consistency, because the read operations always obtain the latest version of the requested data. However, higher consistency levels will result in higher latencies, so we could be interested in weaker consistencies to reduce latency. We have to take this effect into account when selecting the best replica size and consistency level used per operation.

In [62], Bailis et al. propose a model to quantify eventual consistency. Using their model we can obtain the probability of retrieving stale data $\triangle$ seconds after performing a write operation, depending on the consistency level used and the replica size selected. This model shows that the higher the consistency level selected, the smaller is the probability of obtaining stale data. They also prove that higher replica sizes imply higher latencies to obtain the latest possible version of a data element.

In order to handle failures, Cassandra has several built-in repairing processes. The first method is **Read Repair**. This mechanism is in charge of updating outdated replicas of an object when performing a read operation. During read operations, the system checks if the k replicas providing an answer are up-to-date. If not, after returning the reply to the client, obsolete replicas are updated in the background. A configurable parameter *read repair chance* determines the probability of Read Repair to be executed over the outdated replicas. However, this parameter only affects Read Repair behavior for consistency level ONE. When other consistencies are in use, outdated replicas are always repaired using this procedure.

The second method is called **Anti-Entropy Repair**. This method has to be triggered by a system administrator. When executed in a node, the node exchanges information with its replica group companions about the Column Families stored by them in order to search for the most up-to-date data. As a result, all the replicas are updated with the latest values. This procedure can be executed only when there are no failed nodes in the replica group.

Finally, there is a repairing process for writes, called **Hinted Handoff**. This process is in charge of replaying writes over missing replicas. When a write is performed and some of the replicas are off-line, a hint can be written in another alive node until the off-line replica goes back on-line. Note that any hint stored for replay does not count to achieve the requirements of the consistency level used, in terms of nodes reached by the write operation.

For the purpose of maintaining the system operational, Cassandra implements a membership protocol and a failure detector. Membership is based on Scuttlebutt [63], a very efficient gossip-based mechanism. Proposed in [64], gossip can be seen as an epidemic, where information plays the role of a virus, and learning about the information can be seen as the infection process. Each participant periodically communicates with a randomly selected node to exchange information. This way, the updates in the shared data are spread through the system with a logarithmic effort, $O(\log m)$.

Cassandra implements a push-pull gossip variation in each node to build a global view of the nodes belonging to the system. Once a second, every alive node selects another alive node at random in order to exchange information about the rest of nodes in the system. According to [65], the complexity of this gossip mode to spread some information to the entire system is $O\left(\log_3 m + \ln\left(\ln m\right)\right)$ rounds of gossip for $m \to \infty$, using $O\left(m \ln\left(\ln m\right)\right)$ messages to infect the entire system. In Cassandra each round is completed within one second but, as rounds are not synchronized between nodes, this time span can only be seen as an upper bound of the delay induced by gossip when spreading information.

Each time that a gossip message with information about an alive node is received, Cassandra stores the arrival time for each alive node in a heartbeat window. This information is used after the gossip round by the failure detector to estimate which nodes are still alive and which ones are considered dead. Cassandra uses the φ accrual failure detector [66] in order to estimate if a node is alive or not, avoiding communicating with it in the latter case. The basic idea of this failure detector is to estimate a suspicion level φ for each node. This value is computed dynamically, based on the values stored in the heartbeat window. This way current network and node status are properly reflected. This value is compared to a threshold Φ, which is used to decide if the node is dead. The value of this threshold determines the time required by the failure detector to identify a problem, as well as the achieved accuracy. The higher the value of Φ, the longer it takes to detect a dead node, but also the lower the risk of making a mistake.

Now that we have introduced the structure of the system, let us study its behavior from the point of view of the user. A write or read request submitted by a client application can be served by any node in the system. When a client connects to a Cassandra node to perform an operation, that node acts as the coordinator. The task of the coordinator of the operation is to behave as a proxy between the client and the nodes that store the requested data. Additionally, the coordinator determines which nodes in the ring are accessed, based on the system configuration and the replica placement strategy.

With regard to writes, the coordinator node sends the write request to all the replicas. When the required number of nodes, as determined by the consistency level, responds affirmatively as regards the completion of the operation, the answer is sent to the client. In Figure 3.2 we can observe a write operation in a Cassandra system with replica size of 3 and using consistency level ONE. Node 6 performs as a proxy for client A and forwards the operation to each replica (1, 2, and 3). Because of the consistency level, the coordinator only waits for one of the answers before notifying the client about the result. The remaining nodes in the replica set perform the update asynchronously.

In the case of read operations, the coordinator sends the read request to k replicas and waits until the nodes respond. The coordinator compares all the timestamps of the replies and returns the data corresponding to the most recent timestamp, executing Read Repair if needed over the remaining $n-k$ nodes. In Figure 3.3 we show a read operation in a 3-replica system using QUORUM consistency level. In this case, nodes 1 and 3 answer simultaneously. Let us suppose that node 2 has out-of-date data. As the consistency level is QUORUM, the data is sent to the client because there are $\lfloor\frac{3}{2}\rfloor + 1$ answers (from 1 and 3). In addition, a Read Repair process is executed in parallel in order to update the data stored in node 2.

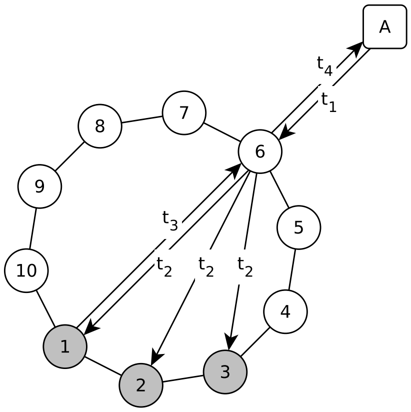


Fig. 3.2: Example of a write operation with $k = 3$ and consistency level ONE. Client is node A, replicas are nodes 1, 2 and 3. Gray circles represent replicas containing up-to-date data.

Having described the architecture of Cassandra and its functionalities, let us see our proposals to model it.

3.2 Stochastic failure models for Cassandra

A Cassandra system with m nodes can be seen as an aggregation of m replica groups, where each group stores several objects. Assuming that workloads

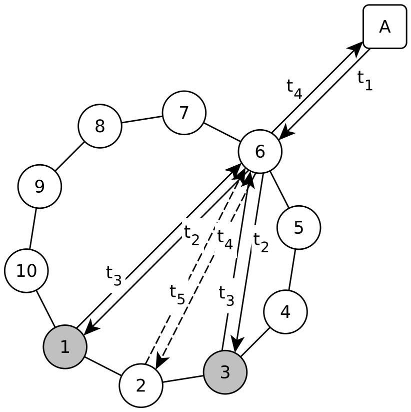


Fig. 3.3: Example of a read operation with $k = 3$ and consistency level QUORUM. Client is node A, replicas are nodes 1, 2 and 3. Gray circles represent replicas containing up-to-date data.

are evenly distributed along the entire set of stored objects, we can model the whole system only focusing on one particular replica group. In the case workloads are note evenly distributed, the whole system could be modelled by a mixture of models (one for each replica group).

Each replica group consists of n nodes that hold copies of the same data. The replica group is considered to be working if at least k nodes in the group are working, where k depends on the consistency level chosen. If level ONE is selected then $k = 1$, because only one of the replicas needs to be present in order to successfully complete an operation. For QUORUM consistency level, $k = \lfloor \frac{n}{2} \rfloor + 1$. Finally, consistency level ALL means that $k = n$. An n-component system that works if and only if at least k of the n components work is called a k-out-of-n:G system [67].

In this Section we propose two different models, one for each type of failure: transient and memory-less. These models will be helpful to analyse the theoretical behavior of Cassandra. Their common characteristics are the following:

1. The replica group is a k-out-of-n:G structure.
2. A repaired node is as good as a no-failed one.
3. Each node is responsible for its own recovery. There is no master node in charge of the recovery process.
4. Each node is independent from the others.
5. Uptime and downtime of each node follow exponential distributions with parameters λ and μ respectively, the same for all nodes.

3.2.1 Modelling transient failures

Kuo et al. [67] propose a general repairable k-out-of-n:G system model with multiple repair facilities, which is based on Markov models of replica groups widely used in the literature. For example, in [68, 69] these Markov models were used to analyze the reliability of computer networks. Focusing on P2P networks, data persistence [70] and optimal data placement [71] have also been studied by means of Markov models. Chun et al. used in [20] a similar model to propose an efficient replica maintenance algorithm for Carbonite, another DHT storage system.

Kuo et al. assumed that, once the system has more than $n-k$ failed components, its life cycle is complete and no more degradation is possible. In the case of Cassandra, even when there are not enough nodes to perform operations correctly with a certain consistency level, the system continues working so that more degradation is possible. Moreover, the failed nodes could also recover by themselves, so that the replica group could eventually return to a working state.

The failure model we propose can be described as follows:

1. We only consider transient failures.
2. The replica group is considered failed when the number of failed nodes is larger than $n-k$.
3. Even in a failed state, more units can fail and recover.

Notation:

- $N(t)$: state of the replica group at time t. It denotes the number of failed nodes, and can take values $\{0,1,\ldots,n-k+1,\ldots,n\}$.
- i: number of failed nodes in the replica group, $i \in \{0,1,\ldots,n\}$.
- λ_i: failure rate of the replica group when there are i failed nodes.
- μ_i: recuperation rate of the replica group when there are i failed nodes.
- $P_i(t)$: probability that there are i failed nodes in the replica group at time t.
- $A(t)$: point availability at time t. The probability of the replica group being operational at time t.
- A: steady-state availability of the replica group.

If we assume that on-line time and off-line time of a given node are exponentially distributed, then $\{N(t), t \geq 0\}$ forms a continuous-time homogeneous Markov process [72] with state space $\Omega = \{0,1,\ldots,n-k+1,\ldots,n\}$. The set of working states is $W = \{0,1,\ldots,n-k\}$ and the set of failed states is $F = \{n-k+1,\ldots,n\}$.

Figure 3.4 is a graphical representation of the proposed model. When the group is in state i, there are i failed nodes and $n-i$ working nodes. The failure rate at state i is $\lambda_i = (n-i)\lambda$. If we assume that there are n recuperation facilities, then $\mu_i = i\mu$.

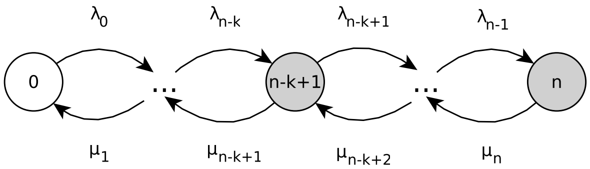


Fig. 3.4: General repairable Markov model describing a k-out-of-n:G system with transient failures. The gray states are non-working states.

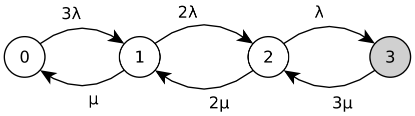


Fig. 3.5: An example of a Markov model of a 3-replica group with consistency level ONE.

In Figure 3.5 we can see an example with three nodes and consistency level ONE, that is, $k = 1$. In this case only the third state is considered as failed.

The point availability can be computed as:

$$A(t) = 1 - \sum_{i=n-k+1}^{n} P_i(t) \tag{3.1}$$

Equation 3.1 shows that the availability of a k-out-of-n:G system is the probability of having k or more nodes working at any time. To calculate the steady-state availability, that is, the availability of the replica group when it is in a steady state, the following equation is used:

$$A = \lim_{t \to \infty} A(t) \tag{3.2}$$

There is a closed form solution [72] for this equation (3.2):

$$A = 1 - \sum_{i=n-k+1}^{n} \binom{n}{i} \left(\frac{\lambda}{\mu}\right)^{i} \left(1 + \frac{\lambda}{\mu}\right)^{-n} \tag{3.3}$$

When considering transient failures, we expect no data loss along the lifetime of the system, but the number of failures affects the availability of the stored objects; so we consider the steady-state availability a good metric to measure the fault tolerance of the system.

We define a *stable* system as one in which μ is several orders of magnitude larger than λ ($\mu \gg \lambda$). An *unstable* system is one in which $\mu \sim \lambda$. Examples of stable systems are storage clusters such as those considered by Ford et al. in [73]. A volunteer computing system such as BOINC [5] could be an example of an unstable scenario.

Using this model, we have computed the theoretical steady-state availability for a replica group with three different replica sizes, $n = \{3, 9, 27\}$ using three consistency levels: ONE, QUORUM and ALL. In order to analyze the behavior of Cassandra in different stability scenarios, we have tested these ranges for λ and μ: $\lambda = \{10^{-4}, 10^{-5}, \ldots, 10^{-8}\}$ and $\mu = \{10^{-4}, 10^{-5}, \ldots, 10^{-8}\}$. In Figure 3.6 we can see the availabilities for each configuration.

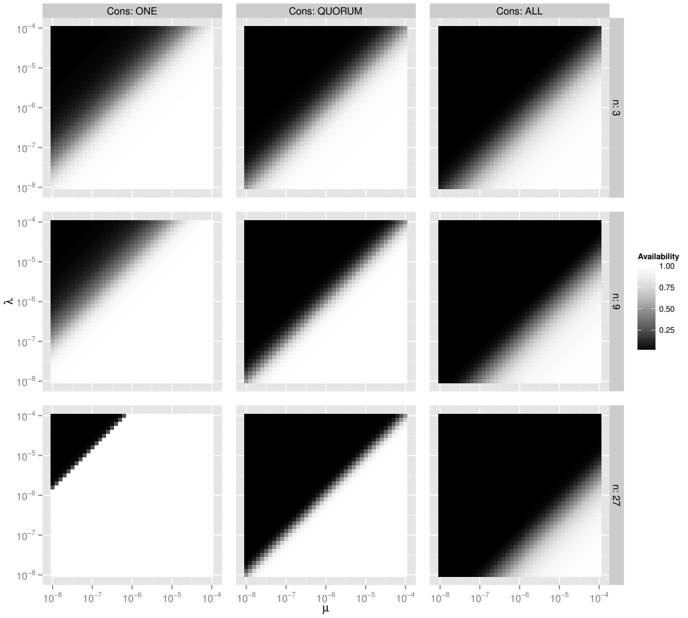


Fig. 3.6: Predicted steady-state availability for different replica sizes (3, 9 and 27) and different consistency levels (ONE, QUORUM and ALL), under different stability scenarios for transient failures.

As can be expected, the availability predicted by the model grows with the recuperation rate μ. Regarding the behavior of Cassandra with different consistency levels and replica sizes, when using consistency level ONE, Cassandra is able to work in rather unstable scenarios. This behavior is improved if Cassandra uses larger replica sizes. Using consistency level QUORUM permits the system to work in the scenarios where $\mu \sim \lambda$. In this case, even the utilization of this consistency level in addition with larger replica sizes does not change the availability of the system but, the larger the replica sizes, the faster is the transition of the system from total unavailability to total availability. In the case of using consistency level ALL, the availability of the system deteriorates as the replica size grows. In this case, a system using consistency level ALL can be usable only in fairly stable environments. This behavior is derived from the number of operational states in the model (see Figure 3.4), which increases with the replica size – but not for consistency level ALL, a case in which only a single state is operational. Therefore, for ALL, large replica sizes are counter-productive in terms of availability. We should not ignore that, in addition to availability, getting up-to-date data is also desirable, and higher consistency levels would be recommendable for this purpose.

The model, therefore, shows that Cassandra can operate with excellent availability in stable systems. However, when this property is not met, actual availability strongly depends on system properties (failure and reparation rates) and Cassandra configuration parameters (replica size and consistency level). In this case, the model we propose can be of great help for the system administrator in order to choose the best parameters for the service wanted to be offered.

3.2.2 Modeling memory-less failures

In order to use Markov models to describe memory-less failures, we need to introduce the concept of fault coverage. It is defined as the probability of a system to recover given that faults occur [74]. When a failure is not perfectly covered, the system falls into a situation where no more reparation is possible. In [75], Akhtar proposes the utilization of Markov models with absorbing states to describe this situation: an absorbing state is one in which all the nodes have failed and, thus, there is no possible recuperation. Akhtar considers a certain parameter ρ, that controls the probability of a failure being repaired correctly. If a failure is not perfectly recovered, the model considers that the system has failed catastrophically and falls into the absorbing state.

In the case of Cassandra, we have three different repairing methods that can affect the coverage of the failure. Hinted Handoff does not repair a replica if the node has lost all the information it stored before a failure; it is only useful to finish write operations in the presence of failures, so we are not going to consider it. Anti-Entropy Repair can be executed by an administrator after a node returns from a memory-less failure if there are no more failed nodes in that group. As a consequence of this repairing method all the nodes from a replica group are repaired. Read Repair can also be useful, as it can repair any object if a read operation is performed over it. If these repairing processes do not succeed before more alive nodes fail, we say that the failures have not been perfectly covered and, therefore, the replica group eventually falls into an absorbing state, where the system is considered as failed. The model can be described as follows:

1. We only take into account memory-less failures.
2. Nodes can be: alive, failed, or alive but not repaired.

3. Each node executes Anti-Entropy Repair when going back on-line.
4. Read Repair is executed for all read operations depending on the consistency level:
   - With consistency level ONE, the execution of Read Repair will depend on the parameter *read repair chance* which determines the probability of Read Repair to be executed over the outdated replicas.
   - With higher consistencies Read Repair is executed in each read operation, the same effect as if *read repair chance* $= 1$.
5. Read operations follow an exponential distribution of parameter λ_{io}, the same for all nodes.
6. Each object has a probability P_{access} of being accessed. This probability depends on the access pattern. An access pattern where there are some objects more accessed than others will result in different per-object P_{access}.
7. The replica group is considered failed when there is no data stored in the alive nodes, or when there are no alive nodes.
8. Once the replica group has failed, it is not repairable.

Notation:

- n: number of nodes of the replica group.
- k: consistency level used.
- i: number of alive but not repaired nodes in the replica group, $i \in \{0, 1, \ldots, n\}$.
- j: number of failed nodes in the replica group, $j \in \{0, 1, \ldots, n\}$.
- $\lambda_{i,j}$: failure rate of the replica group when there are i alive but not repaired nodes and j failed nodes.
- μ_j: recuperation rate of the replica group when there are j failed nodes.
- λ_i: failure rate of the replica group when there are i alive but not repaired nodes.
- α: rate of Read Repair of the system over one object.
- β: rate of Anti-Entropy Repair of a node.
- λ_{io}: read rate of each reader.
- P_{access}: probability of accessing the modeled object.
- rr: *read repair chance*. For consistency level ONE, $0 \leq rr \leq 1$. For other consistencies the system considers $rr = 1$.

Each state of the Markov model represents the situation of all nodes in the replica group: they can be alive, failed or alive but not repaired. A state has the form $X_{i,j}$, where i represents the number of alive but not repaired nodes and j is the number of failed nodes. The absorbing state is called X_n, representing all the states $X_{i,j}$ where $i + j = n$ and thus, the system is not recoverable.

Alive nodes fail with a failure rate of $\lambda_{i,j} = (n - (i + j))\lambda$. Failed nodes come back alive with a rate of $\mu_j = j\mu$. Alive but not repaired nodes fail with a failure rate of $\lambda_i = i\lambda$.

Regarding the Read Repair process, the rate α depends on the probability of reading the object stored in that replica group, which depends on the number of possible readers, m, the read rate of each reader, λ_{io}, and the probability of reading a specific object from the total number of objects, P_{access}. Therefore:

$$\alpha = P_{access} \cdot m \cdot \lambda_{io} \tag{3.4}$$

It should be noted that Equation 3.4 depends on the modeled data access pattern. In a real scenario, it is usual to access some objects more often than others, so, the probability of finding those "popular" objects updated is higher than for other, less accessed, objects. Therefore, these "popular" objects will be repaired more frequently.

Finally, the Anti-Entropy Repair rate β depends on the duration of an Anti-Entropy Repair process. We describe in Section 3.3.2 how we have determined the β rate.

Reparations cannot start until other nodes have noticed that the repairing node is alive. This process is done using the gossip protocol explained before. This process will condition the reparation time required by each reparation process. If we consider that the time required by each reparation process is the inverse of each rate, α and β, we can modify these times to include the time required by gossip to detect the node as alive. As said, the gossip complexity is $O\left(\log_3 m + \ln\left(\ln m\right)\right)$. We will use this expression as an upper bound of the time required by gossip to spread the information about the node.

$$t_{gossip} = \log_3 m + \ln\left(\ln m\right) \tag{3.5}$$

Taking into account this expression, we could modify the reparation times required by each repairing method to include t_{gossip}:

$$t_{RR} = \frac{1}{\alpha} + t_{gossip} \tag{3.6}$$

$$t_{AE} = \frac{1}{\beta} + t_{gossip} \tag{3.7}$$

Being t_{RR} the ReadRepair reparation time and t_{AE} the Anti-Entropy reparation time. The inverse of each of the reparation times will be the rate of each repairing method, taking into account gossip.

The model also needs to take into account that, after a node recovers from a failure and goes back to the on-line state, some time is required to set up again the Cassandra process. From the point of view of availability, this boot time t_{boot} is off-line time. Therefore, the offline time of the node must be computed as:

$$t_{offline} = \frac{1}{\mu} + t_{boot} \tag{3.8}$$

The inverse of this time will be the reparation rate parameter of each node.

In Figure 3.7 we show this failure model with imperfect failure coverage. The model has $\frac{n(n+1)}{2} + 1$ states. In Figure 3.8 we can see an example of this model for a 3-replica group.

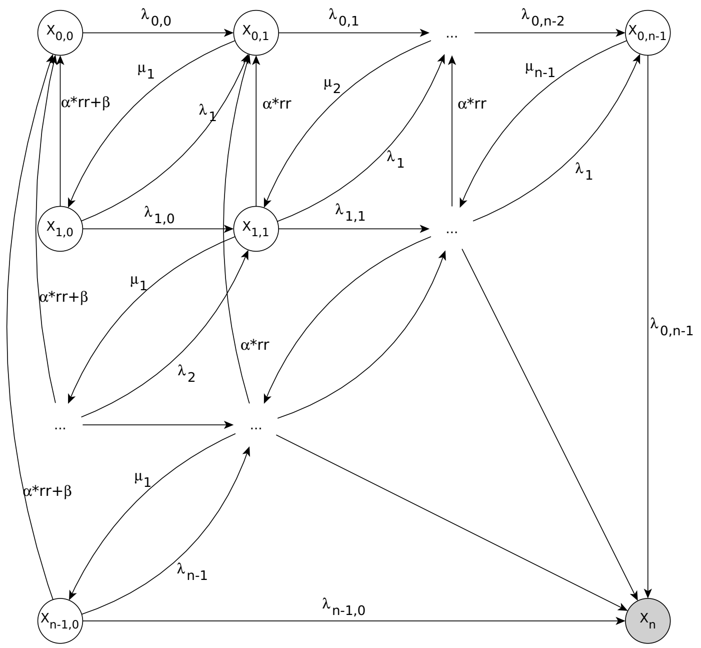


Fig. 3.7: A repairable Markov model describing a k-out-of-n:G system with memory-less failures. The gray state is the absorbing one.

We have used this model to compute the availability of the system and the Mean Time To Failure, or MTTF, which is the time required by a replica group to arrive to state X_n.

To compute both metrics we have used the Queueing Toolbox for Octave [76] and SHARPE [77]. Both are general hierarchical modelling tools that analyse stochastic models of reliability, availability and performance. These packages take into account the related Markov reward model [78]. The Markov reward model is built by considering the original Markov model, where each state has an associated reward rate $r_{i,j}$. From this reward model, we can calculate the mean time spent in each state, $\tau_{i,j}$. The sum of all these times is the $MTTF$. By considering the fraction of time from the $MTTF$ spent in each state and the associated reward rate of the state, we can obtain the availability A of the model for a specific configuration during this period:

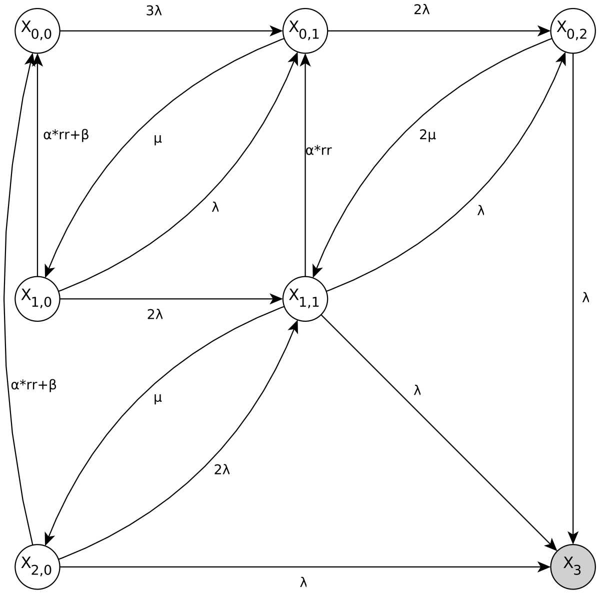


Fig. 3.8: A Markov model of a 3-replica group with memory-less failures. The gray state is the absorbing one.

$$MTTF = \tau_n + \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} \tau_{i,j} \tag{3.9}$$

$$A = r_n \cdot \frac{\tau_n}{MTTF} + \sum_{i=0}^{n-1} \sum_{j=0}^{n-i-1} r_{i,j} \cdot \frac{\tau_{i,j}}{MTTF} \tag{3.10}$$

where τ_n and r_n are the time spent and reward rate of the state X_n.

To calculate these equations, we need the values for $r_{i,j}$ and $\tau_{i,j}$. The last one is given by $\tau_{i,j} = \int_0^\infty P_{i,j}(x)dx$, where $P_{i,j}(x)$ is the probability of being in state $X_{i,j}$ at time x. With respect to the reward rate, it depends on the consistency level used and can be computed as:

$$r_{i,j} = \begin{cases} \frac{\binom{n-j}{k} - \binom{i}{k}}{\binom{n}{k}} & n - j \geq k \wedge k < n \\ 1 & j = 0 \wedge k = n \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

The reward rates $r_{i,j}$ are the fraction of requests received by the system that will be answered correctly, which depends on the number of nodes alive and repaired. For example in Equation 3.11, when $n - j \geq k \wedge k < n$ each request will access k nodes, so the reward rate is the number of combinations of k nodes taken from the $n-j$ alive nodes, minus the number of combinations in which only alive but unrepaired nodes are accessed, divided by the number of combinations of k nodes taken from the total n nodes in the replica group.

We have computed the MTTF and availability of a system with replica size 3, 300 data objects, all of them with the same probability of access, $\lambda_{io} = 0.1$, different values of $\lambda = \{0.01, 0.02, \ldots, 0.1\}$, and $\mu = 10\lambda$. The choice of these parameters was done to study availability in relatively unstable systems. From Equation 3.4 we compute $\alpha = 0.01$. Finally, we have tested different values of β to study the impact of this repairing method in Cassandra ($\beta = \{0.01, 0.1, 1\}$). Figure 3.9 shows the results obtained. We can see that the more unstable the system is, the shorter the MTTF is and the smaller the availability is. For the selected parameters, larger values of β (which means shorter execution times of the Anti-Entropy Repair) result in higher MTTFs. However, our model states that, for more stable systems, β has almost no influence on the MTTF. This is because in these systems the impact of repairing a node is not significant compared to the long duration of on-line times of the nodes. However, it should be noted that this parameter depends also on the amount of data stored in each node. Large datasets will take longer to be transfered when repairing a failed node, negatively affecting system availability.

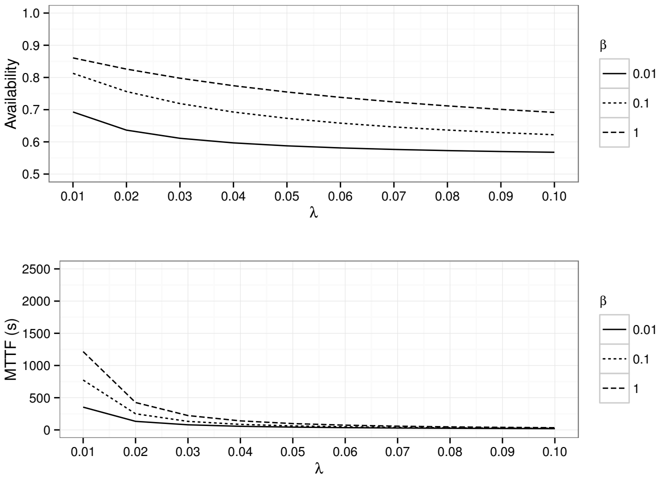


Fig. 3.9: Availability and MTTF for a system with memory-less failures for different values of β, $n = 3$, $k = 1$, $\alpha = 0.01$, $\lambda_{io} = 0.1$, $\lambda = \{0.01, 0.02, \ldots, 0.1\}$ and $\mu = 10\lambda$.

The frequent activation of ReadRepair (in scenarios with consistency ONE, because ReadRepair is always executed for higher consistency levels) improves significantly the values of MTTF and availability. We show this in Figure 3.10, which summarizes the availability and $MTTF$ computed for a system with replica size 3, 300 data objects, $\lambda = \{0.01, 0.02, \ldots, 0.1\}$, and $\mu = 10\lambda$, with consistency ONE. β has been fixed to 0.01, $\alpha = 1$, and ReadRepair chance $rr = \{0.01, 0.1, 1\}$. Clearly, MTTF and availability grow with rr.

We must take into consideration that the ReadRepair reparation process is executed only as a side-effect of a read operation over an object that is not up-to-date. Therefore, it will rarely run for infrequently-accessed objects, while being executed more frequently for popular objects. The administrator may select, in a per-table basis, the value of rr.

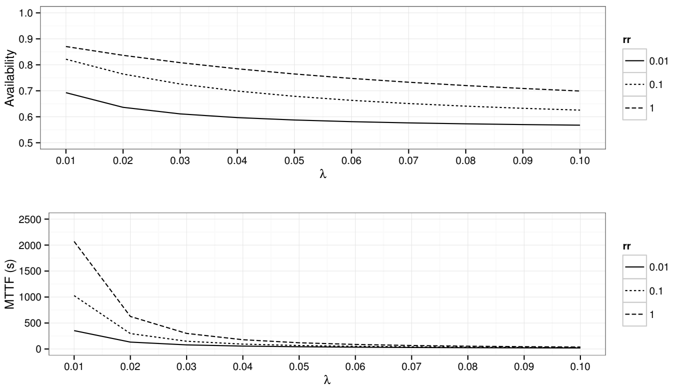


Fig. 3.10: Availability and MTTF for a system with memory-less failures for different values of ReadRepair chance rr, $n = 3$, $k = 1$, $\beta = 0.01$, $\alpha = 1$, $\lambda = \{0.01, 0.02, \ldots, 0.1\}$ and $\mu = 10\lambda$.

3.3 Validating the models

In order to validate our models, we have made an analysis of the behavior of a real Cassandra system, comparing the obtained availability with that predicted by the models.

The experimentation platform consists of a cluster of 30 nodes. Each node has an Intel(R) Core(TM)2 6320 @1.86GHz processor, 2 GB of RAM, a 150 GB hard-drive and a Gigabit Ethernet card. All nodes are configured with Ubuntu Server 12.04.1, Apache Cassandra 1.2.1 and Ruby Cassandra 0.12.2. In addition, each node is configured with a token which guarantees that the key space is evenly distributed along the nodes.

Our purpose is to measure the availability of the system when performing I/O operations over multiple objects. For this purpose, we have created a Column Family to store the different objects to be accessed during the experiments. In total, we have 10 objects per node on average, each one filled with L random bytes, where L is sampled from a uniform distribution with parameters 2^{10} and 2^{20}. Therefore, the length of the data chunks varies from 1KiB to 1MiB.

With respect to the operations, they are performed from each alive node in the system. I/O operations follow an exponential distribution with parameter λ_{io}. For each I/O operation, we measure the time spent to perform the operation and whether it succeeded or not.

The remainder of this section is devoted to describing the experimental study, focusing first on transient failures, and secondly on memory-less failures.

3.3.1 Validating the model for transient failures

To emulate transient failures in a node, we have simply paused the Cassandra process running on it. The process can be easily resumed later, maintaining the previous data, but missing the possible updates during the off-line period.

In this experiment, we monitor a Cassandra cluster after a warm-up period, during which the ratio of nodes leaving / returning to the system stabilizes. The information shown in this Section is obtained averaging data from 20 measurement intervals. In each of these intervals we measure the availability of an object as the number of correct I/O operations performed over it. The steady-state system availability is computed as the average of the availabilities of all the objects.

In order to validate our model, we will first perform an experiment with a system that keeps the assumptions of our model: uptime and downtime of each node follow exponential distributions with parameters λ and μ respectively, and all data elements are accessed uniformly.

The choice of parameters λ and μ will make our Cassandra work in relatively unstable situations. This has been chosen on purpose for two reasons. The first one is to complete the set of experiments in reasonable time. The second is to compare the experiments with the model in the most interesting cases, trying to avoid extreme situations of full or null availability.

The failure rate has been fixed to $\lambda = 0.01$, while the repair rate μ takes values $\{0.01, 0.02, \ldots, 0.1\}$. The I/O rate has been fixed to $\lambda_{io} = 0.1$, that is, on average each alive node requests one operation every ten seconds. As in [31], we have tested different proportions of reads and writes; however, we have not observed any difference in the availability of the system with different proportions of I/O operations; therefore, we decided to maintain the same proportion between read and write operations. Finally, we have tested the system with three replica sizes, 3, 5, and 7, and three levels of consistency, ONE, QUORUM and ALL. Results can be seen in Figure 3.11. Boxplots represent average (minimum, maximum, lower, median and upper quartile) availability of data in the cluster. We have also plotted the expected availability obtained from our theoretical model as a solid line.

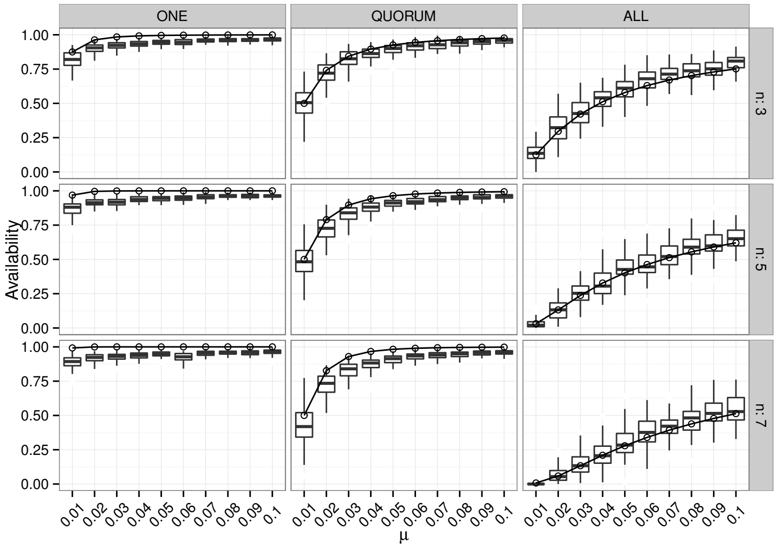


Fig. 3.11: Average availability for a replica group with transient failures compared to the theoretical model. $\lambda_{io} = 0.1$, $\lambda = 0.01$, $\mu = \{0.01, 0.02, \ldots, 0.1\}$. Three replica sizes, 3, 5, and 7, and three levels of consistency, ONE, QUORUM and ALL. Boxplots show the measured availability. Model predictions are plotted as a solid line.

The figure shows that there is a good fit between measured availability values and those predicted by the model. As expected, availability is better when μ increases with respect to λ and, consequently, the off-line periods become smaller. The experiments also show the impact of different replica sizes and consistency levels on availability.

The behavior observed in these experiments validates our model as a good descriptor of Cassandra when dealing with transient failures if the reality keeps the assumptions of our model; however, we could ask ourselves if the model is still valid when these assumptions are not fulfilled by the real system. In order to prove this, we have performed additional experiments changing some conditions: using non-uniform data accesses and non-exponentially distributed failures.

As the access pattern we have chosen the Zipfian distribution. Using this distribution, some objects have higher probability of being accessed than others (P_{access} increase with the object's identifier), so that more I/O operations will be performed over these objects. The Zipfian distribution has two parameters: the number of objects and a certain parameter s that controls the distribution of the probability among the different elements. In our experiments, the number of elements is 300 and $s = 1$.

In order to simulate non-exponential failures, we have chosen another common distribution when approximating the behavior of nodes, the Weibull distribution. This distribution has two parameters: a *rate* and a *shape*. We will use two Weibull distributions, one to model node failures, and the other to model reparations. The failure rate has been fixed to $\lambda = 0.01$, while the repair rate μ takes values $\{0.01, 0.02, \ldots, 0.1\}$. As to the shape parameters, in [79] Javadi et al. analyzed several failure traces from real systems. They concluded that the shape parameter of Weibull-distributed failures takes values in the range $(0.33 - 0.85)$, which means that the expected survival time of nodes grows with time. Following these recommendations, we have chosen to fix the failure shape parameter to $k_f = 0.7$. With respect to the shape of the repair distribution, in the same study the authors conclude that it takes values in the range $(0.35 - 0.65)$, so we have fixed this parameter to $k_r = 0.5$.

In Figure 3.12 we show the four possible combinations of distributions (uniform and zipfian) and failure distribution (exponential and Weibull) for the case of a system using replica size 3, QUORUM consistency level and $\lambda_{io} = 1$. This parameter has been increased to 1 with respect to the previous experiment in order to reduce the variability among measurements and show the effects of the different distributions. In each plot we show also the predictions of our model as a solid line. The parameters λ and μ of the model have been estimated from the actual failure trace captured from each experiment.

Observing the Figure 3.12, changing the access distribution slightly increases the variance among measurement intervals but does not change the availability of the system when compared to a uniform access. With regard to the distributions of failures and reparations, we can see that using Weibull instead of exponential results in an increase in the measurement variance. We can also see that, apparently, a system whose nodes fail and recover following a Weibull distribution will have a smaller availability in the most extreme conditions, those where λ and μ are similar, than the same system following exponential distributions. This could be because the reparation rate of the Weibull distribution decreases over the time because of the shape parameter. The combination of a non-uniform data-access and a different failure distribution gives us a system with lower availability and more variance in its behavior. However, our model is still capable of reflecting the behavior of Cassandra and gives quite accurate predictions of the system. This is the case for each condition showed in the plot but, also, for each replica size, consistency level and workload we have tested.

3.3.2 Validating the model for memory-less failures

In this case, a node failure has been simulated by killing the Cassandra process and erasing the data stored in the node. This way, when it recovers from the

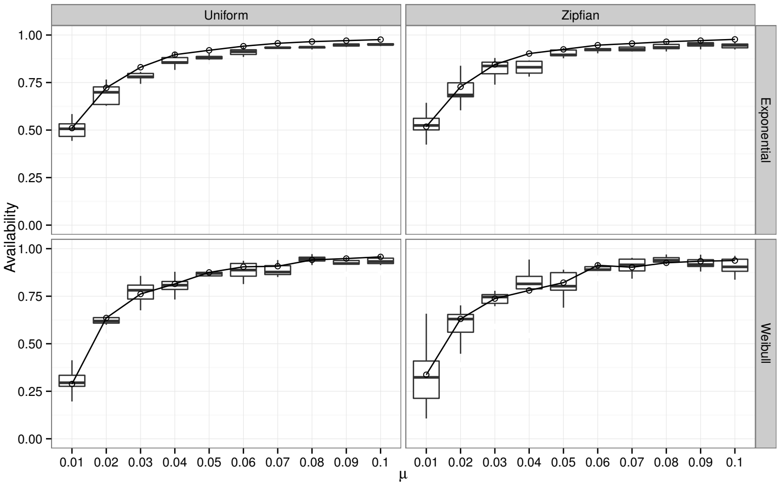


Fig. 3.12: Average availability for a replica group with transient failures compared to the theoretical model. $\lambda_{io} = 1$, $\lambda = 0.01$, $\mu = \{0.01, 0.02, \ldots, 0.1\}$ for two different data access distribution, Uniform and Zipfian, and two different failure distributions, Exponential and Weibull, for a replica 3 system with 30 nodes using QUORUM consistency level.

failure, it knows where to put itself in the token ring, but has to obtain all the lost data from its replica companions.

The model for this type of failures is used to study the ability of Cassandra to maintain data after a write operation in the presence of catastrophic failures. In real systems there will be backup policies that will permit system recovery, but in this case, the system will inevitably go towards an irrecoverable situation. The longer the expected time between catastrophic failures, the smaller the effort required by system administrators to recover from data losses.

We execute our experiment until no object is available in the system. We compute the Time To Failure (TTF) for each object in the cluster as the time elapsed from the beginning of the experiment until no more correct reads are possible because no data is available for that object. Using this value we obtain the availability of the object as the number of correct operations over it from the beginning of the experiment until the TTF of the object. The availability of the system is the average of the availability of all objects.

Regarding the parameter values, we have used: $\lambda_{io} = 1$, $\lambda = \{0.02, 0.025, 0.03, 0.035, 0.04\}$, $\mu = 10\lambda$ and replica groups of size three, five and seven. Due to the design of our experiment, it is not possible to measure the TTF of the objects using consistency levels higher than ONE. Read operations using higher consistency levels will fail more frequently and, therefore, measurement of the TTF would not be accurate. Therefore, we have only tested this kind of failures with consistency level ONE. With respect to the time required to boot each node, after it comes back on-line, it has been estimated to be 5 seconds, independently of the replica size used in the system. The use of the t_{boot} time in the model has been already discussed in Section 3.2.2.

With respect to the reparation processes, we set the ReadRepair chance $rr = 1$ in order to execute ReadRepair at each read operation. Anti-Entropy Repair has been configured to be executed after the Cassandra process has finished booting.

In order to compute the availability of the modeled system, we have to determine the values of model parameters α and β. With regard to the Read Repair rate, α, of the proposed model (see Equation 3.4), we have to consider the popularity of each object. In this experiment we have considered that each object has the same probability of being accessed, so $\alpha = \frac{1}{300} * 30 * 1 = 0.1$ repairs/second.

Regarding the Anti-Entropy Repair rate, β, we have performed some empirical measurements of the time required by this repairing process in the Cassandra cluster with different numbers of objects and replication factors. More specifically, we have measured the time required to perform Anti-Entropy repair in a cluster of 30 nodes with replica sizes $n = \{2, \ldots, 11\}$ and a number of stored objects that varies from 300 to 3000. The size of each object was sampled uniformly from the range (1KiB,. . . ,1MiB). Based on the obtained values, we have fitted the following regression function, $f(n, nobj)$, which describes the duration of the Anti-Entropy repairing process (Equation 3.12). The coefficient of determination of this fit is $R^2 = 0.9443$.

$$f(n, nobj) = e^{0.3465n+0.002448nobj+0.3851} \tag{3.12}$$

Using Equation 3.12, we determined the mean duration of the anti-entropy repair process for the three replica sizes. This mean duration will be the inverse of the β parameter.

Figure 3.13 summarizes the measured availability for this experiment (boxplots) together with the predicted value by the Markov model (solid line). It is to be highlighted that the experiment focuses on unstable situations and, therefore, the predicted and measured values are short and inadequate for production-grade systems. However, they are useful to check the accuracy of the proposed model.

Clearly, availability decreases when the replica size increases. This can be explained if we consider that higher replica sizes results in a larger number of invalid states: those in which only non-repaired nodes reply to a read request.

The most important insight from this figure is that measured availability follows the same trends predicted by the model. In terms of actual values, measured and predicted availability values are not identical, with differences ranging from 1.27% to 13.19%. Therefore, we can take our model as a good approximation to the real behavior of a real Cassandra cluster.

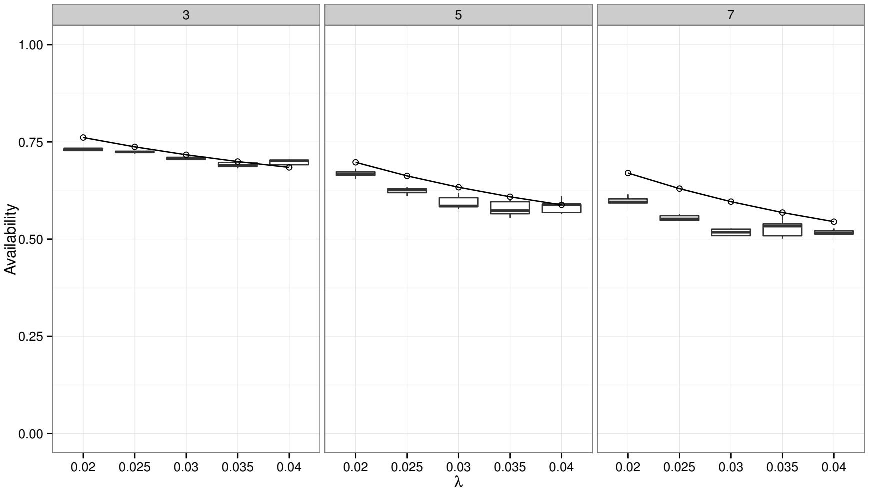


Fig. 3.13: Average availability for a 30-nodes system with $n = \{3, 5, 7\}$, $k = 1$, $\lambda_{io} = 1$, $\lambda = 0.02, \ldots, 0.04$ and $\mu = 10\lambda$ compared to the theoretical model with $\alpha = 0.1$. Boxplots show the measured availability. Model predictions are plotted as a solid line.

We have also validated the model against scenarios in which nodes fail and recover using non-exponential distributions, and the access pattern to objects is not uniform. We have repeated our experiments with a real cluster making the nodes fail following Weibull distributions with the same rates ($\lambda = 0.02, \ldots, 0.04$, $\mu = 10\lambda$), with shapes $k_f = 0.7$ for failures and $k_r = 0.5$ for reparations. Additionally, tests with I/O operations following a Zipfian distribution have been conducted. In the model, parameters λ and μ have been estimated from the actual failure trace captured from each experiment. The results of the comparison are summarized in Figure 3.14. As can be seen, in the case of uniform data access, the measured availability follows closely that predicted by the model. However, when the access pattern is not uniform, predicted availability is worse than the measured one, although both follow the same trend. In summary, this additional set of results confirms the overall capacity of the model to describe the availability of Cassandra.

We presume that the differences between model-predicted values and actual measurements are mostly caused by the time required to perform certain operations in Cassandra, that the model do not consider adequately. An example could be the duration of the ReadRepair process, which is instantaneous in the model but, in practice, depends on the size of the object being repaired, the characteristics of the underlying communications network, and the replica size. Also, our estimation of the duration of the Anti-Entropy repairing process requires additional tuning. Currently, we only consider the *number* of objects

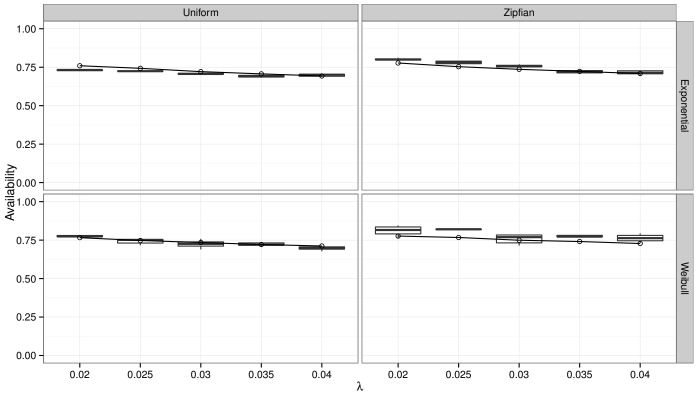


Fig. 3.14: Average availability for a 30-nodes system with $n = 3$, $k = 1$, $\lambda_{io} = 1$, $\lambda = 0.02, \ldots, 0.04$ and $\mu = 10\lambda$ compared to the theoretical model with $\alpha = 0.1$ and $\beta = 0.11544$ for different combinations of Exponential and Weibull failure distributions with different access patterns (uniform and zipf). Boxplots show the measured availability. Model predictions are plotted as a solid line.

in the system, but not their *sizes*. Finally, in our model we have considered that the number of readers is constant, m, which is the number of system nodes. However, in a real set-up, as nodes fail and recover, very often the number of active nodes is lower than m, reducing the probability of reading objects and, therefore, the probability of executing ReadRepair operations.

Even with these limitations, we consider that the proposed model provide a good approximation to the behavior of real Cassandra systems, sufficient to study their availability characteristics, and to make "what if" questions that could help an administrator to identify adequate configurations, for example in terms of consistency level, replica size and frequency of execution of the repairing processes. In the next Section we will discuss possible scenarios for which the model could be a valuable tool. As future work, we plan to analyze and include in the model the aforementioned delays and the changing I/O rate.

3.4 Using the models

The models proposed and validated in the previous sections can be used to select the best configuration for a Cassandra cluster given the characteristics of the data it will store and the parameters of the system in which it will run. In this Section we present several examples of the possible utilization of the models.

3.4.1 Availability in the event of network failures

Consider a web application using a Cassandra system to store its data. This web application accesses Cassandra following a “read-heavy” pattern: 95% of the operations are read-only, and the remaining 5% require updates. This behavior may resemble a newspaper web site or an online store. In this type of web applications, most of the clients are only viewing pages published by a minority of power users (the news writers or the site administrators). In a situation of network outages, a system like this could be evaluated using the transient failures model, which informs about the theoretical availability provided by Cassandra. Then, the administrator could use this prediction to select the best possible configuration.

In our example, we are going to work with a Cassandra system composed of 30 nodes. Each node will suffer from network failures with a rate of 10^{-6}. Each network outage will be repaired with a rate of 10^{-3}, which means that we can have a failure per node every 10 days and that the off-line time can be 15 minutes on average. Note that these failures do not imply the loss of any data item already stored.

As failures are transient, we can analyze this problem using our transient failures model to estimate the availability that will show the application. This model has the following parameters: number of replicas per object, consistency level used per operation, the failure rate and the reparation rate.

In Table 3.1 we show the availability obtained for each combination of consistency level for different replica sizes. The question to be answered is: Which is the best combination of replica size and consistency level in terms of availability? As an additional restriction, we have to consider that higher replica sizes imply higher costs in terms of network traffic, management of the consistency between replicas and higher read latencies, so lower replica size is desired.

According to the results, the best solution is to use ONE or QUORUM consistency levels for both read and write operations, having an almost perfect availability. Using ALL in one type of operation decreases the availability of the system and eliminates the positive effects of increasing the replica saize. However, we have to consider consistency. If we use a combination where $r+w \leq n$, read operations could obtain stale data for some time after the last write operation. We have to take this effect into account when selecting the best replica size and consistency level used per operation. In combination with our model, we could use the eventual consistency model proposed by Bailis et al. in [62] to select the best possible configuration in terms of consistency and availability.

Therefore, if stale data is not an issue for our application, the best possible configuration in terms of availability is to use always consistency level ONE. However, if we care about stale data, the best solution is to use QUORUM consistency level in both write and read operations instead of ALL. By doing this, we will have a totally consistent system with the highest possible avai-

Availability for a “read-heavy” access pattern				
Read Consistency	**Write Consistency**	**Replica size**		
		3	**5**	**7**
ONE	ONE	99.999	99.999	100
ONE	QUORUM	99.999	99.999	99.999
ONE	ALL	99.985	99.975	99.965
QUORUM	ONE	99.999	99.999	99.999
QUORUM	QUORUM	99.999	99.999	99.999
QUORUM	ALL	99.984	99.975	99.965
ALL	ONE	99.715	99.526	99.337
ALL	QUORUM	99.715	99.526	99.337
ALL	ALL	99.700	99.501	99.303

Table 3.1: Availability (in %) for several combinations of consistency levels for an application with 95% of reads and 5% of writes for a system of 30 nodes with $\lambda = 10^{-6}$ and $\mu = 10^{-3}$. Note that an availability of 99.999% may correspond to 315.36 failed seconds in a year, while an availability level of 99.303% corresponds to 60 hours of unavailability in a year.

lability. Finally, with respect to the replica size, the smallest possible replica factor will imply lower I/O latencies.

Note that in this example we have selected a “read-heavy” access pattern (95% reads, 5% writes), which is representative of most web applications. However, the models could be used with any possible access pattern. In fact, we have applied our model to “frequent update” (50% reads, 50% writes) and “write-heavy” (5% reads, 95% writes) patterns. Availability values differ depending on the access pattern but, as the conclusions in terms of what is the best combination of consistency and replica size are still valid, we have not included these results.

3.4.2 Availability and MTTF of an application in the event of disk failures

Apart from network failures, systems suffer from disk failures. We could analyze the same application described in Section 3.4.1 under this type of failures in order to know the time that the system will be capable of maintaining data without any backup. We are also interested in knowing which levels of availability will show this application under disk failures. We can use the memory-less failures model to estimate these two metrics. Remember that the studied system has 30 nodes. In this case we are not considering network failures. Instead we consider disk-failures. For illustrative purposes let us suppose that the failure and reparation rates of this type of failures are $\lambda = 10^{-6}$ and $\mu = 10^{-3}$. The application that will access Cassandra has 95% of reads and 5% of writes.

We have noted down in Table 3.2 the different availabilities for each combination of consistency level for different replica sizes together with the MTTF of the system. As can be seen, the predicted MTTF for this stable system with the smallest replica factor is higher than $10,000$ years, which proves the high resilience of Cassandra. The included repairing processes in Cassandra seem to be enough to repair any dead node.

With respect to the availability of the system, using ONE or QUORUM are again the best possible options, while using consistency level ALL decreases the availability, and implies higher I/O latencies. In the case of applications with an interest in consistency, QUORUM could be used for both read and write operations. If that is not the case, ONE is the best possible option. However, note that increasing the replica size when using QUORUM results in a decrease of data availability. In this case, this behavior can be explained if we consider that QUORUM will increase the number of invalid states because there will be states where only not repaired nodes will answer to a read request.

Availability and MTTF for a "read-heavy" access pattern				
Read Consistency	**Write Consistency**	**Replica size**		
		3	**5**	**7**
ONE	ONE	99.899	99.899	99.899
ONE	QUORUM	99.894	99.889	99.883
ONE	ALL	99.889	99.879	99.869
QUORUM	ONE	99.804	99.709	99.615
QUORUM	QUORUM	99.799	99.699	99.599
QUORUM	ALL	99.794	99.689	99.584
ALL	ONE	99.709	99.519	99.331
ALL	QUORUM	99.704	99.509	99.316
ALL	ALL	99.699	99.499	99.301
MTTF (s)		3.25×10^{11}	1.90×10^{17}	∞

Table 3.2: Availability (in %) and MTTF (in seconds) for several combinations of consistency levels for an application with 95% of reads and 5% of writes. Analyzed system has 30 nodes, $\lambda = 10^{-6}$ and $\mu = 10^{-3}$. In a year, an availability of 99.301% corresponds to 61.23 hours of unavailability.

3.4.3 Analyzing Cassandra when dealing with churning

Cassandra has been designed to run in stable conditions such as controlled data-centers. However, we would like to analyze it in other environments. The HTC-P2P system presented in Chapter 2 has been designed to operate in a wide range of situations: from very stable environments such as data-centers to less stable situations such as desktop grid computing systems, where churning is the most important cause of a node not being available. We could consider churning as a cause of transient failures. Therefore, the question that we can make to our model is, what levels of churning permit the use of Cassandra with some guarantees of availability?

In Figure 3.15 we can see different configurations of Cassandra for different ratios of λ and μ. The higher the ratio between these two values ($\mu \gg \lambda$), the more stable the system in which Cassandra is working. As can be seen, using consistency levels ONE and QUORUM permits the system to work with a high availability in quite unstable situations, where μ is very similar to λ. However, to use Cassandra with consistency level ALL, the off-line periods of the nodes need to be at least 1000 times smaller than the on-line periods. Nodes with these requirements do not fit into the definition of a usual volunteer computing node. We can also observe the effect of increasing the replica size. As previously mentioned, while increasing the number of replicas implies a higher availability using ONE and QUORUM, this is not the case with consistency level ALL.

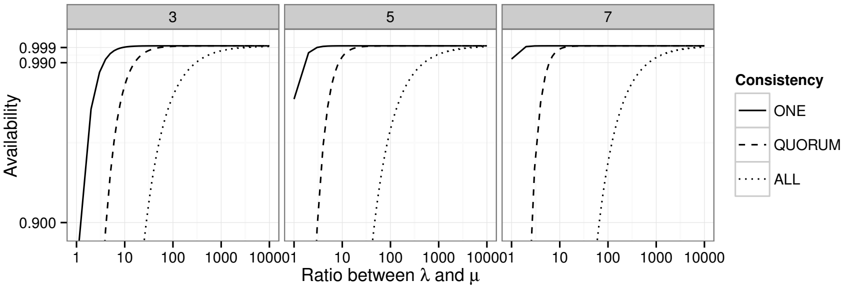


Fig. 3.15: Availability of Cassandra for different consistency levels and replica sizes for different ratios between λ and μ. Note the logarithmic scale in the x-axis.

In conclusion, we could say that, even if Cassandra is designed for very stable environments, its robustness permits utilizing it with a high availability in very unstable situations using consistencies ONE and QUORUM.

3.4.4 Analyzing the frequency of execution of Anti-Entropy repair

We could use the memory-less model to find the best frequency to execute Anti-Entropy repair in the system. This process implies computing checksums of each data stored in a replica group and comparing them to find the latest version of each and every object in the replica group. This process is expensive and should be executed occasionally. In our experimentation we have configured our platform to execute Anti-Entropy always after a node returns from a failure; however, we could add a parameter to control the probability of this process. We have analyzed the availability and MTTF of a system with 30 nodes, $\lambda = 10^{-5}$ (one failure per day and node), $\mu = 10^{-2}$ (each failure lasts 100 seconds) and replica size 2 when varying a parameter θ which determines the probability of Anti-Entropy repair ($\theta = 1$ always, $\theta = 0$ never). In Figure 3.16 we show the predicted MTTF in days and the availability of this system. As can be seen, if we execute Anti-Entropy only half of the times a node has failed, we can add 20 days of MTTF while halving the cost of the reparations, maintaining availability above 99.887%.

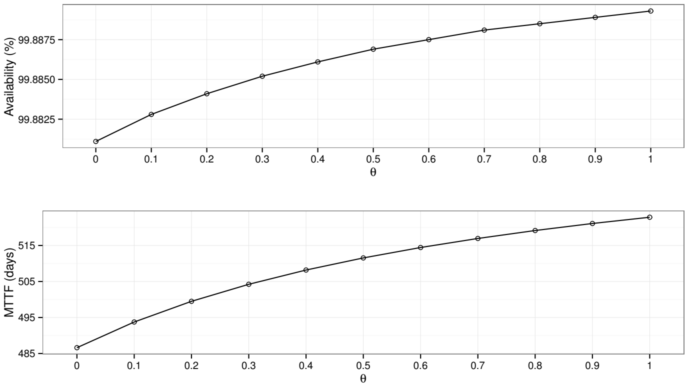


Fig. 3.16: MTTF for a system with 30 nodes, $\lambda = 10^{-5}$ and $\mu = 10^{-2}$ when variating the probability of executing Anti-Entropy repair (θ in the plot).

3.4.5 Analyzing the point availability with disk failures

In order to better understand the behavior of Cassandra in the event of catastrophic failures, we have analyzed how point availability decreases with time in a system with memory-less failures. We have studied a system with 30 nodes, failure rate of $\lambda = 10^{-4}$ and repair rate of $\mu = 10^{-2}$, with several replica sizes (3, 5 and 7) and consistency levels. At $t = 0$ an object is written in the system. In Figure 3.17 we can see how the availability of this object changes for each configuration.

As can be seen, availability decreases below 0.999 quite fast and remains stable for some time. The length of this period depends on the replica size. With respect to the level of availability attained during this period, this depends on the consistency level used. MTTF grows with the replica size, while availability is better for lower consistency levels. From the point of view of availability, and without considering other costs, large replica sizes with consistency ONE is the best option.

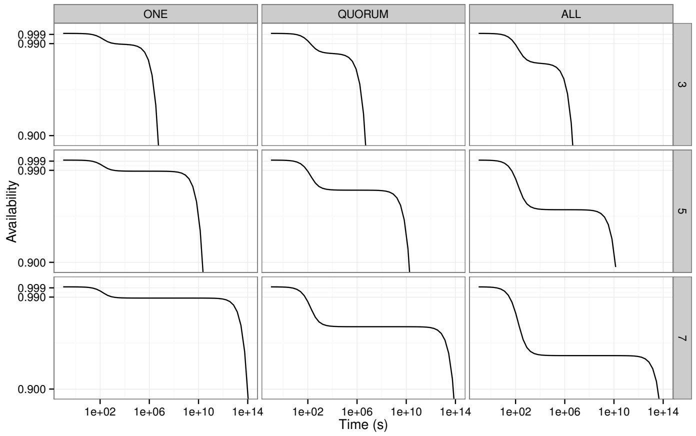


Fig. 3.17: Evolution of the availability for a system with 30 nodes, different replica sizes (3, 5 and 7), $\lambda = 10^{-4}$ and $\mu = 10^{-2}$ for different consistency levels.

3.5 Related work

To the best of our knowledge, this is the first attempt to model of Cassandra's availability. However, there are a few works close to this that should be mentioned. In [31], Yahoo! proposes a system for benchmarking different cloud data-storage systems, the Yahoo! Cloud Serving Benchmark (YCSB). YCSB executes operations over cloud systems in order to measure the performance and scalability of these systems. As future improvements to this tool, Yahoo! will focus on reliability. At the time of writing this chapter, no progress has been made in this area.

Authors of [80] propose a failure model of P2PMPI, a peer-to-peer implementation of the Message Passing Interface (MPI) [81], and use it to estimate the optimal degree of replication in this P2P system.

Carra et al. [82] propose a work similar to ours over KAD [83], a DHT based file sharing system without data replication. In KAD, each node is responsible for the objects it stores. Each node publishes references to its objects in other nodes to help find data. Unlike our proposal, they take into account only memory-less failures, and consider that nodes are periodically replaced with new ones without reparation costs. Based on the theory of reliability, they analyse the probability of finding an object published in the system, and study how this probability evolves over time. Finally, they use that model to propose improvements in KAD that could reduce the cost of maintaining the availability of the objects by reducing the amount of object references that can be created in the system.

There are several reliability analyses of different redundancy schemes in P2P networks, on which we have based the models described in this chapter targeting Cassandra. Houri et al. investigate in [84] the behavior of different replication and erasure code schemes (a form of forward error correction) [85] using a non-stochastic reliability analysis to estimate the probability of losing data and the cost of repairing data. In [86] we can find an statistical analysis of replication vs. erasure code schemes whose conclusion is that using erasure codes is less expensive in bandwidth terms, while it increases the Mean Time To Failure by several orders of magnitude. However, in [87] an stochastic analysis comparing erasure codes and replication concludes that simple replication is better when peer availability is low.

In [88] we can find an analysis of the impact of node churn in different erasure code schemes when the availability periods of nodes follow three different distributions: exponential, Pareto and Weibull.

When analyzing the behavior of networked systems, some works [20, 89, 90] assume independence between nodes. However, in [73] Ford et al. prove that failures in a data-center where nodes share switches and power units are not independent. Nevertheless, in this chapter we assume independence between nodes in order to reflect a wider range of situations, from volunteer desktop computing systems to corporate data-centers.

With respect to the distribution of failures and recoveries, the uptime and downtime of nodes are commonly modeled as exponentially distributed [20, 89, 91, 92, 93, 94, 71, 95], although several works [96, 97] prove that the time between disk failures is not exponentially distributed and exhibits autocorrelation and long-range dependences, so authors argue that long-tail distributions (such as Weibull) would be better approximations. Yet, in [73], Ford et al. argue that, in large and heterogeneous environments, the aggregation of large populations of disks with different ages tends to be stable, so the failure rate should be stable as well. When the failure rate is stable, there are no significant differences between both distributions. Therefore, in this chapter we have considered that modeling failures and recoveries using exponential distributions is also a reasonable approximation.

3.6 Conclusions and future work

In this chapter we have modeled the availability of Cassandra when confronting two types of failures: transient failures and memory-less failures. For this purpose, we have modeled replica groups of Cassandra as k-out-of-n:G systems. These models have been compared against experiments in a real Cassandra system in order to validate them.

Our model of Cassandra dealing with transient failures has demonstrated to be an excellent descriptor of the actual availability of this storage system when dealing with problems such as networking outages or churning.

The second model describes the behavior of Cassandra when dealing with memory-less failures (such as disk breakdowns), estimating for how long a data object is available even in the presence of these catastrophic failures. Regarding this model, we need to improve the estimation of the Anti-Entropy repair rate. Currently we are estimating β using the replica size and the number of objects. This estimation should also consider the size of the objects to be repaired. The ReadRepair parameter, α, must be improved too. Currently we assume that the ReadRepair process is instantaneous, but it requires reading the target object from all the replicas, updating the staled ones. The cost of this process depends on the replica size and should also take into account the size of the objects stored in the system.

The proposed models can be used to study the fault tolerance of Cassandra and to help in the selection of the best configuration for Cassandra depending on the environment in which it is going to be used, and also on the needs of the application that will store and retrieve data. Similarly, at run time, an application could vary some operational parameters (such as consistency levels and number of replicas) adapting to the running environment or the profile (mostly read-only, frequently updated, etc.) of the different applications that access the system. In general, the models can be of help to select the best consistency level and number of replicas.

Currently we keep two different models for transient and memory-less failures. In a real environment, actual failures can be of any of these classes, although with different probabilities. The proposed use cases reflect situations in which a type of failure is much more frequent than the other one and, thus, the selected model adequately reflects the behavior of Cassandra. It would be possible, though, to enhance the second model to include transient failures in addition to memory-less ones, by adding new states and transitions. This is a non-trivial effort which we plan to undertake as future work.

Cassandra is highly available in stable scenarios, and can even show a high availability when the system does not have this property, depending on the selected configuration. Several aspects of Cassandra could be improved in order to support a wider range of scenarios. The Anti-Entropy repairing process should be enhanced in order to better deal with catastrophic (memory-less) failures. Currently, it can be used only when a single node in the replica group fails, but it could be extended to operate with more than one failure. Another possible improvement could be a better detection of catastrophically failed nodes in order to create extra replicas of the objects stored in those nodes before more nodes fail, by triggering the Anti-Entropy repairing process.

Finally, note that while we have developed these models to Cassandra, the replication model of Cassandra is based in Dynamo and, thus, they could be applicable to this system as well as to others based on the same principles. Our transient failures model is directly applicable to these systems while our memory-less failures model could be adapted by taking into account the particularities in the reparation processes of each particular storage system.

4

Failure-aware scheduling in HTC systems

In this chapter we address an issue that also affects centralized HTC systems: the waste of resources derived from worker nodes leaving the system while executing tasks that remain unfinished, due to failures or churning. Note that, for scheduling purposes, there is no difference; simply, a node is not available. We will use the terms “failed node” and “failure” to simplify the discussion. The tasks being executed in failed nodes need to be resubmitted to the system for re-execution, causing additional (scheduling) overheads, and affecting the responsiveness of the system from the point of view of the user. Some common approaches to address this problem are checkpointing and task replication.

Checkpointing is a technique that permits a running task to periodically store snapshots of its status somewhere in the system. If the node in which it is running fails during the execution of the task, another worker can resume the execution from the last available check-point. Checkpointing does not eliminate the waste of resources entirely: the CPU cycles used since the last snapshot are still thrown away. Additionally, this technique requires extra space to store the snapshots.

Task replication consists of executing several simultaneous replicas of the same task in different nodes. If one of them fails, the execution can hopefully succeed in one or more of the remaining replicas. This mechanism improves system responsiveness from the point of view of the users submitting tasks, but the overhead to pay in terms of wasted resources is severe.

These two techniques try to minimize the impact of a failure in a node while it is executing a task. Note that the volume of wasted resources increases for long-lasting tasks and, therefore, the issue is not severe for short tasks. *Failure-aware* scheduling tries to characterize tasks and nodes in order to find appropriate task-to-node matches. If we know that a node is very stable, it would be the preferred choice for long-lasting tasks. Other nodes, more prone to fail, could be used for short tasks. The scheduler can make this kind of decisions, although there is a price to pay in terms of scheduling overheads: it takes longer to wait for the “right” node to run a task, instead of using the first available one. However, the number of aborted executions should be lower. Note that failure-aware scheduling techniques can be combined with checkpointing and task replication, in order to build an HTC system in which the effects of node failures are minimized.

When scheduling tasks in an HTC system there are two main alternatives for fault-tolerant scheduling algorithms: knowledge-based scheduling and knowledge-free scheduling. *Knowledge-based* scheduling requires using estimations of the duration of tasks to find good task-to-node matches, which are normally provided by the submitting users. In contrast, *knowledge-free* algorithms do not consider information about the task, and use replication in order to avoid failure-prone (or, simply, slow) nodes. This results in severe waste of resources because, from all the replicas executing a task, only the one finishing first is actually useful, and the remaining ones will be canceled. This wasted time could have been effectively used by other tasks, or the corresponding energy could have been saved by switching off idle nodes.

In this chapter we propose and evaluate a collection of failure-aware knowledge-based scheduling techniques. We have implemented and tested them in the context of our HTC-P2P system, obtaining important benefits in terms of system utilization and delays experienced by tasks. In addition, we want to remark that our proposals are perfectly applicable to centralized HTC systems.

In summary we propose to build, for each node, a failure model that characterizes its expected lifetime. When selecting the tasks to run, the scheduler (remember that each node has its own scheduler) will prioritize those whose duration fit into the node's predicted survival time. As several nodes can contend for the execution of the same task, a competition among nodes, based on a certain *score*, is implemented to obtain the best task-to-node match. An optimization of this process is to schedule simultaneously *groups* of tasks. With this proposal, we have been able to achieve a 20% increase in system utilization, taking as reference the (failure-agnostic) FCFS scheduling policy, in scenarios where there is enough diversity of nodes in terms of availability. Our proposals are also competitive when compared against other failure-aware scheduling algorithms proposed in the literature.

The remaining of this chapter is structured as follows. In Section 4.1, we present our proposals of failure-aware scheduling policies. In Section 4.2 we introduce the score functions used by these policies. In Section 4.3 we describe two failure-aware algorithms taken from the literature that will be used for comparison purposes. In Section 4.4 we explain the environment used in our simulation-based experiments. In Section 4.5 we show and discuss the results of the experiments. Results from additional experiments using non-exponentially distributed failures are presented in Section 4.6. Related work about scheduling in the presence of failures is presented in Section 4.7. We end with some conclusions and plans for future work in Section 4.8.

4.1 A proposal for failure-aware scheduling in an HTC-P2P system

The HTC-P2P system we have presented in Chapter 2 implements the scheduling process in a completely distributed manner. Nodes collaborate to maintain the structure of the P2P network and, thus, the data stored in the system – including the queue where submitted tasks await. This *distributed but shared* queue is implemented using Cassandra. In this data-storage system, nodes can reach any point of the system with a complexity of $O(1)$ hops, so the time required to access any item stored in this queue is constant, regardless of the number of components (nodes) of the system. Each node implements its own scheduler, that does not take into consideration the properties of other nodes. However, different forms of coordination between nodes can be implemented.

The HTC-P2P system can be modelled as a collection of n identical nodes with independent schedulers. They share a task queue Q, which is used by users to submit tasks, and by nodes to choose the tasks to execute. Each task j in Q has an user-defined expected execution time (length), l_j. A per-node adjustment of this length could be performed if nodes were not identical.

Although each scheduler is independent, they implement a heartbeat mechanism to monitor the remaining nodes in the system, in order to detect node failures and enable the re-execution of aborted tasks. When a node detects a failed partner, the task being executed by this failed node is cancelled and reinserted into Q.

Over this shared queue we can define several scheduling policies, whose performance can be analysed using a collection of metrics. From the point of view of the HTC system, the most important metric is the *task throughput*: the number of tasks per unit time that the system can process. Given a fixed number of tasks, a related metric is the *make-span*, or time required to process all these tasks.

If we consider the individual behaviour of the nodes, the most important metric is their *effective utilization*, that is, the portion of the node's on-line time that is effectively used to execute tasks. When a node fails, the time used to process aborted tasks (those initiated but not completed) is *wasted time*. Note that the time used by a long task that had to be aborted might have been useful time for a shorter task that runs until completion.

From the point of view of the tasks submitted to the system, we are interested in measuring the waiting overheads: the time spent in the queue (this is the *waiting time*), and the time wasted in incomplete executions (this is the per-task *wasted time*). The waiting time includes (1) a, typically short, time used by the scheduler to run the resource management algorithm and (2) a time that depends on the current utilization of the system: time to execute tasks ahead in the queue, time awaiting until free resources are available.

In the following sections we describe the scheduling algorithms proposed for this HTC-P2P system, starting with the baseline: the distributed version of First Come First Served (FCFS) that our proposal uses by default, which is a failure-agnostic and knowledge-free policy. Then we discuss two failure-aware proposals that allow nodes to compete for a given task, taking into account the estimated task length and the expected survival time of the nodes. We insist in that these algorithms are implemented by all nodes: there is no central scheduler.

4.1.1 Distributed First Come First Served scheduling

When a node is idle and willing to execute a task, it accesses Q and gets the task at the head of the queue, executing it. After completing the task, the node stores any resulting file into the storage system and signals the completion to the task's owner. If Q is empty, the node sleeps for τ_s seconds before retrying. This process is detailed in Algorithm 1.

Algorithm 1: FCFS scheduling

```
while true do
    if size(Q) > 0 then
        w := dequeue(Q);
        execute(w);
        store_results(w);
    else
        sleep($\tau_s$);
```

The main advantage of this policy is that the execution order follows the insertion order. However, as it is failure-agnostic, it allows a very unstable (failure-prone) node to choose an inadequate task (a long one). Therefore, we can expect many execution attempts for long tasks, until they are finally completed.

4.1.2 Competition scheduling

The re-executions of aborted tasks translate into wasted resources (that could have been used in a more effective manner) and also into longer response times (that generate a negative perception of the system from the user's viewpoint). We propose a failure-aware scheduling policy that tries to reduce re-executions. It is implemented by means of a per-task competition between nodes, in which a *score* function determines the winner node: the one that will run the task. It is sketched in Algorithm 2. A ready node selects the task at the head of Q and computes its score for that task. This score is used to enroll the node into a list of candidate workers for the task. After τ_c seconds, nodes check the candidate list. The node with the best score removes the task from the queue and runs it, while the remaining candidates abandon the competition. Note that, when the set of candidates contains just one node, it will run the task, independently of its score. Thus, a bad node-to-task assignment is still possible.

Algorithm 2: Competition scheduling

```
while true do
    if size(Q) > 0 then
        w := first(Q);
        score := calculate_score(w);
        inscribe_as_worker(w, score, nodeid);
        sleep($\tau_c$);
        if nodeid = best_scored_node(w) then
            remove_from_queue(Q, w);
            execute(w);
            store_results(w);
    else
        sleep($\tau_s$);
```

An important property of this scheduling policy is that, like FCFS, it respects the arrival order of tasks. However, less re-executions are expected as tasks will preferably go to the best nodes (how “good” a node is depends on the choice of score, discussed in Section 4.2). This should translate into increased system throughput and higher node utilization.

4.1.3 Competition scheduling in groups of tasks

Instead of considering for scheduling purposes just the task at the head of Q, this proposal is a generalization of the previous case, and analyses the group with the first G tasks in Q, for $G > 1$, looking for the task in the group that better matches the characteristics of the available nodes (see Algorithm 3).

A node ready to run a task selects a maximum of G tasks from the head of the queue and calculates its score for each of them. Then, the node competes *only* for the task for which it has the best score. The use of larger values of G increases the opportunity of finding a good match for the node, by considering several waiting tasks. This way, the probability of successfully completing the selected task should increase.

The main drawback of this method is that the arrival order of tasks is not strictly respected. Additionally, some tasks may suffer from severe extra delays when, even after reaching the head of the queue, they are not chosen because no node finds them “adequate”. In order to limit this problem, which could lead to starvation, we have included in our implementation a limit in the number of times a task at the head of the queue can be skipped.

Note that these algorithms have been described for an HTC-P2P environment, but could be easily implemented in a centralized set-up. The manager

Algorithm 3: Competition scheduling in groups

```
while true do
    if size(Q) > 0 then
        works := select_group(Q, G);
        best_score := 0;
        best_work := nil;
        for w in works do
            score := calculate_score(w);
            if score > best_score then
                best_score := score;
                best_work := w;
        w := best_work;
        score := best_score;
        inscribe_as_worker(w, score, nodeid);
        sleep($\tau_c$);
        if nodeid = best_scored_node(w) then
            remove_from_queue(Q, w);
            execute(w);
            store_results(w);
    else
        sleep($\tau_s$);
```

node needs to keep availability models of all nodes, using this information together with the centralized task queue to carry out the selection of the best node-to-task assignments.

4.2 Score functions

The competition-based scheduling algorithms defined in the previous section require some companion score functions. In this section we propose two different, although related, score functions to be applied to a ⟨task, node⟩ pair. The first one is based on the probability that the node survives enough time to complete the task. The second relates this expected survival time with the task's length, with the aim of obtaining a good node-to-task fit.

4.2.1 Based on the expected survival time of a node

The first score function we propose, f_1, is based on the probability of the node surviving enough time to complete the task. If we consider that X_i is a random variable that describes the lifetime of a given node i, the score for node i and task j (of length l_j) is computed as:

$$f_1(i,j) = P(X_i > t + l_j | X_i > t) \tag{4.1}$$

where t is the time since node i came on-line.

As done in Chapter 3, we are going to assume that each node i fails and recovers following exponential distributions with parameters λ_i and μ_i respectively. However, note that the proposed f_1 score function could be calculated using any other distribution.

One of the most important properties of the exponential distribution is that it is memoryless, which means that the probability that a certain node lives for at least $t + l_j$ seconds given that it has survived t seconds is the same as the initial probability that it lives for at least l_j seconds. Therefore, Equation 4.1 can be written as:

$$f_1(i,j) = P(X_i > l_j) = 1 - P(X_i \leq l_j) \tag{4.2}$$

Therefore, the score function for node i and task j can be expressed as 1 minus the cumulative distribution function of the exponential distribution:

$$f_1(i,j) = 1 - (1 - e^{-\lambda_i l_j}) = e^{-\lambda_i l_j} \tag{4.3}$$

In order to compute Equation 4.3, the value of λ_i must be known. In a real system, it can be estimated from a log of past failures. Given an independent and identically distributed sample $(x_i^1, \cdots, x_i^m)$ of past alive times for node i, the maximum likelihood estimate for parameter λ_i is:

$$\widehat{\lambda_i} = \frac{1}{\overline{x_i}} \tag{4.4}$$

where $\overline{x_i}$ is the mean of all the samples (alive times) for that node.

In Figure 4.1 we can see the values provided by this score for different failure rates and task lengths. As can be seen, the node with the lowest λ_i has always the highest score, so the system prioritizes the execution of tasks in this kind of nodes (the most stable ones). In contrast, failure-prone nodes (those with the highest values of λ_i) will execute tasks only when better nodes are busy.

This score function has been designed with the aim of reducing the number of re-executions, because tasks will be executed more likely by the most stable nodes, so that the probability of completing a task at the first attempt should be high. However this score alone does not guarantee a perfect distribution of tasks among the most suitable nodes, because the behaviour of the system depends on its composition (number and reliability of the nodes), the characteristics of the tasks being submitted (mainly short tasks vs. mainly long tasks, or a balanced mixture) and even the order in which tasks are submitted. For example, in an extremely good scenario of very stable nodes running short tasks, no improvement can be expected from competition-based scheduling algorithms, regardless of the selected score.

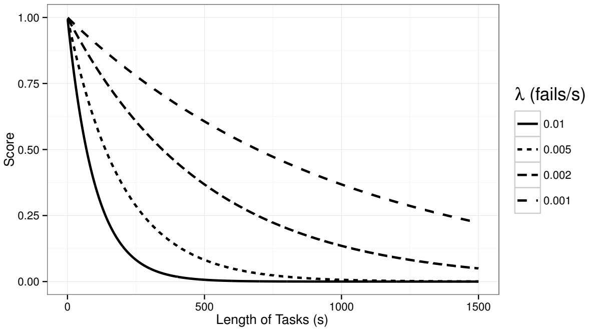


Fig. 4.1: Values of score f_1 based on the expected survival time of nodes (different node failure rates).

4.2.2 Based on the fitness of the duration of a task and the expected survival time of a node

Although the previous score function apparently fulfils our requirement of reducing re-executions, it is not good enough. It leads to a task assignment criterion based only on the stability of the nodes, independently of the lengths of the tasks. Now we present a new score function that not only determines if a node is suitable to complete a given task, but also if the task length suits the expected survival time of the node. What we want is to favour the execution of long tasks in stable nodes, using the unstable ones for short tasks, as a way to increase node utilization and system throughput.

Besides the probability of a node i being alive enough time to complete task j, we also take into account the (normalized) gap between l_j (the length of task j) and the expected lifetime of the node:

$$D(i,j) = \frac{|l_j - E[X_i]|}{E[X_i]} \tag{4.5}$$

where $E[X_i]$ is the expected lifetime of node i. The smaller D, the better the fit of the task into the survival time of the node.

The second score we propose, f_2, combines f_1 with D: it is directly proportional to the node's probability of completing the task and inversely proportional to the normalized gap. From Equations 4.1 and 4.5 we can express this score as:

$$f_2(i,j) = \frac{P(X_i > l_j + t | X_i > t)}{D(i,j)} \tag{4.6}$$

If the lifetime of a node is modelled using the exponential distribution, the expected lifetime of node i is:

$$E[X_i] = \frac{1}{\lambda_i} \tag{4.7}$$

Therefore, the normalized gap D can be expressed as:

$$D(i,j) = |\lambda_i * l_j - 1| \tag{4.8}$$

Finally, from Equation 4.6:

$$f_2(i,j) = \begin{cases} \text{MAX\_SCORE} & \text{if } E[X_i] = l_j, \\ \frac{e^{-\lambda_i l_j}}{|\lambda_i * l_j - 1|} & \text{otherwise.} \end{cases} \tag{4.9}$$

Where *MAX_SCORE* is a certain value considered as the maximum possible score[1].

In Figure 4.2 we can see the scores provided by f_2 for different failure rates and task lengths. As can be seen, the highest score is obtained when the length of the task matches perfectly the expected lifetime of a node. A competition scheduling using score f_1 favors the use of the most stable nodes (from the set of available ones). Score f_2 helps selecting the most suitable node-task pair, assigning short tasks to unstable nodes while leaving stable nodes available for longer tasks. The expected result is an improved utilization of the system, although tasks sent to unstable nodes may require a higher number of re-executions.

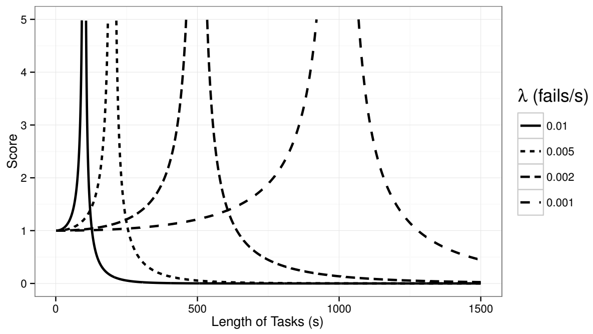


Fig. 4.2: Values of score f_2 measuring node-to-task fitness, for different node failure rates and task lengths.

4.3 Other failure-aware scheduling algorithms

In order to assess the quality of our proposals, in the evaluation section we use as baseline the distributed FCFS scheduling algorithm, but we will also take

[1] $3.40282347 * 10^{38}$ in our implementation

into consideration other failure-aware algorithms. In particular, a failure-aware modification of WorkQueue with Replication/Fault Tolerant [98], and the algorithm discussed in [99] by Amoon (FR onwards). This puts our proposals at disadvantage, because these two algorithms are implemented in a centralized way, which means that they suffer lower overheads in terms of scheduling delays and coordination efforts (they do not wait any time between scheduling attempts or between steps of the algorithm). We have considered the option of re-implementing the competitor algorithms in a distributed fashion, but we estimate that our evaluation approach is fair and shows the potential of distributed, failure-aware scheduling algorithms.

4.3.1 Failure-aware WorkQueue with Replication/Fault-Tolerant scheduling

WorkQueue with replication (WQR) [100] is a centralized scheduling algorithm for bags-of-tasks that uses replication to avoid the effects of differences in performance among system nodes. In WQR, the scheduler sends tasks to randomly selected idle nodes, until the queue is empty. Then, if idle nodes are available, some tasks are replicated in these nodes. The system sets a maximum number of replicas per task. When one of the replicas finishes, the remaining ones are cancelled. This algorithm is depicted in Algorithm 4. Q is the waiting queue, while R is a list with the running tasks; I is a list of idle nodes, and $MAX\_REPLICAS$ is the maximum number of per-task replicas. In our tests, we have used values 2 and 4 for this parameter.

Algorithm 4: WorkQueue with Replication (WQR)

```
while true do
    if size(Q) > 0 then
        w := dequeue(Q);
        n := get_random_node(I);
        w.num_replicas = 1;
        enqueue(R, w);
        execute_in_node(w, n);
    else
        while size(Q) = 0 and size(R) > 0 and size(I) > 0 do
            w := dequeue(R);
            if w.num_replicas < MAX_REPLICAS then
                n := get_random_node(I);
                execute_in_node(w, n);
                w.num_replicas++;
                enqueue(R, w);
```

WQR Fault-Tolerant [101](WQR-FT onwards) adds fault tolerance to WQR using checkpointing and automatic restart. In [98], Anglano et al. introduced a failure-aware version of WQR-FT, WQR-FA onwards, in which the node to execute the task is not selected randomly. Instead, the scheduler computes a score for each idle node and then the best idle node (that with the best score) is selected; note the similarity with our competition-based proposal, but in a centralized environment. The score function used in WQR-FA is based on the predicting binomial method described in [102] that estimates the lifetime of the nodes. For each node i, the algorithm considers $x^{(i)}$, an ordered list storing the past n on-line times of the node, a level of confidence C and X_q (the q^{th} quantile of the distribution of the lifetime of the node). Using these parameters, the binomial method calculates the largest k for which the following equation holds:

$$\sum_{j=0}^{k} \binom{n}{j} (1-q)^{n-j} q^{j} \leq 1 - C \tag{4.10}$$

With the computed value of k we can obtain $x_k^{(i)}$ and a C lower bound for X_q, which is the score used by WQR-FA to select the best idle node among the available ones. The rest of the WQR-FA algorithm is similar to WQR-FT. This algorithm requires as parameters the confidence level, C, and quantile, q. In their paper [98], Anglano et al. use $C = 0.98$ and $q = 0.05$, so these are the values that we will use in our tests. With respect to the maximum number of per-task replicas, we have used 2 and 4.

Although WQR-FA, like WQR-FT, uses checkpointing, we have not included this feature in the comparison tests carried out in this chapter, in order to make a fair comparison between algorithms. Therefore, in all cases, an aborted task restarts from the beginning. Note that checkpointing could be easily integrated into our distributed competition-based scheduling mechanisms.

4.3.2 A fault tolerant scheduling system for computational grids (FR)

In the failure-aware scheduling algorithm proposed by Amoon in [99] (FR), a centralized scheduler uses a score to select the most suitable node to run a task. The per-node score is based on the failure rate of the node (this explains the acronym given to the algorithm). It is computed by considering the number of times a node has successfully completed the assigned tasks, as well as the total number of executions (both successful and aborted) performed by the node. Given all the per-node scores, the scheduler selects the task at the head of the queue and assigns it to the best node. This is like our distributed competition-based algorithm, but with a different score. We see it sketched in Algorithm 5. N_f^i is the number of times a task has been aborted in node i,

N_s^i is the number of successfully completed executions, l_w is the length of the current task w, and R_i is the speed of node i.

Algorithm 5: FR scheduling

```
while true do
    if size(Q) > 0 then
        w := pop(Q);
        best_score := $\infty$;
        best_node := 0;
        for i in I do
            $fr := \frac{N_f^i}{N_s^i + N_f^i}$;
            $T_{exe} := \frac{l_w}{R_i}$;
            score:=$T_{exe} * (1 + fr)$;
            if score $\leq$ best_score then
                best_score := score;
                best_node := i;
        execute_in_node(w, best_node);
    else
        sleep($\tau_s$);
```

Note that FR uses the user-provided estimation of the task length, l_w, and adapts it to a heterogeneous system by performing a correction based on R_i, as we suggested previously. In our tests we consider homogeneous nodes in terms of performance (but not in terms of stability), therefore using $R_i = 1$. Also, note that the score used by FR is very similar to f_1: the most stable node from the free set (the one with the best availability history) will win. Unstable nodes will lose the competition, unless they are the only options.

4.4 Experimental environment

In order to assess our scheduling policies and metrics, a custom-made event-driven simulator of the scheduling process has been developed. It is based on the event-driven engine used in [103], which implements a variation of the calendar queue presented in [104].

Simulated nodes access a single scheduling queue used to store and retrieve the tasks to be executed. While a node is alive, it executes tasks. If a node fails during the execution of a task, the task is reinserted at the head of the queue for a retry. The experiment finishes when all the tasks in the queue have been executed.

Each experiment is repeated 20 times with different seeds for the random number generator (used to generate the workloads and to cause failure and recovery events in nodes). The results shown in figures and tables are the average values of these 20 repetitions.

4.4.1 Scenarios under test

These are the main characteristics of our simulations and the parameter sets used in the experimentation:

- With respect to nodes:
  - We simulate an HTC-P2P system with $n = 1000$ nodes.
  - We consider two types of nodes, called *stable* and *unstable*. Stable nodes fail rarely and recover quickly: the failure rate is several orders of magnitude smaller than the recovery rate [73]. Unstable nodes fail frequently, with a recovery rate similar to the failure rate. In the simulation, the behaviour of each node is managed by two exponential distributions with parameters λ_i (failure rate) and μ_i (recovery rate). In particular:
    - · Stable nodes: $\lambda_i = 10^{-6}$ fails/s and $\mu_i = 10^{-4}$ recoveries/s.
    - · Unstable nodes: $\lambda_i = 10^{-4}$ fails/s and $\mu_i = 10^{-3}$ recoveries/s.
  - We simulate three different system types, with different proportions of stable and unstable nodes:
    - · Stable system (majority of stable nodes): a system composed of 90% of stable nodes and 10% of unstable nodes.
    - · Mixed system: a system composed of 50% of stable nodes and 50% of unstable nodes.
    - · Unstable system (majority of unstable nodes): a system composed of 10% of stable nodes and 90% of unstable nodes.
  - Each node stores a log of its on-line periods, used to continuously update the estimate of the $\widehat{\lambda_i}$ of the node. $\widehat{\lambda_i}$ is bootstrapped at the beginning of the simulation to a very low failure rate, 10^{-8} failures/second, for all the nodes in the system. Once the first failure happens, the value of $\widehat{\lambda_i}$ is recalculated using the information gathered in the log.
  - Each node has a parameter, τ_s, to control the time between consecutive scheduling attempts. This parameter has been set to $\tau_s = 10$ seconds.
  - For the policies involving competition, nodes wait for τ_c seconds from the beginning to the end of the competition. This parameter has been set to $\tau_c = 10$ seconds.
- With respect to tasks:
  - In each experiment, the simulator generates an ordered collection of tasks, constituting a *workload*. All the tasks in the workload are inserted into the queue at the beginning of the experiment in order to test each scheduler in a situation of load saturation. Tasks are independent.

- Tasks are characterized by an execution time or length. This length is sampled from different uniform distributions, yielding three types of tasks:
  - Small (S): $U(1s, 1500s)$.
  - Medium (M): $U(1500s, 6000s)$.
  - Large (L): $U(6000s, 25000s)$.
- We have designed three different workload types, depending on the mixture of tasks constituting the workload:
  - Small workload: formed by 80% of tasks of type S, 10% M and 10% L.
  - Medium workload: composed by 80% of tasks of type M, 10% S and 10% L.
  - Large workload: 80% of tasks of type L, 10% S and 10% M.
- All workloads have been designed to have the same total duration (the sum of the lengths of all constituting tasks), $W = 10^9$ seconds. Therefore, each workload has a different number of tasks Num_tasks. For example, a *small* workload has many more tasks than a *large* one.
- When a task is aborted due to a node failure, it is reinserted for execution at the head of the queue. A maximum number of trials (100) has been set in order to avoid situations in which the HTC system is unable to process the workload (yielding never-ending simulations).

We have chosen this parameter set in order to generate a variety of scenarios in terms of types of tasks (that is, task lengths) and nodes (that is, node availability behaviour) in such a way that we can test if our proposals are capable of assigning tasks to nodes in different environments, from very stable ones (e.g., enterprise clusters) to very unstable ones (e.g. volunteer computing networks in which churning is a frequent event). We use the term *scenario* to refer to a particular combination of a system type (unstable, mixed, stable) with a workload type (small, medium, large). Therefore, we evaluate nine different scenarios.

Note that when modelling the behaviour of nodes we are assuming exponentially distributed failure and on-line times, and the metrics we propose in Section 4.2 are also based on this distribution. However, this does not mean that the proposed metrics are valid only for exponentially distributed failures. They can be used independently of the behaviour of the nodes, although if that behaviour is known to follow a particular distribution, the score can be tailored to better reflect the expected nodes' lifetime. In order to check the general validity of scores f_1 and f_2 as defined above (using the properties of the exponential distribution), we have carried out an additional set of experiments in which node failures follow Weibull distributions. In Sections 4.5 and 4.6 we show the results from the experimentation using exponential and Weibull distributions respectively.

4.4.2 Scheduling algorithms and metrics under test

We have tested the following combinations of scheduling policies and metrics:

- FCFS: First Come First Served, as described in Section 4.1.1.
- WQR: WorkQueue with Replication, as described in Section 4.3.1, with a maximum of 2 and 4 replicas per task.
- WQR-FA: the failure-aware version of WQR, as described in Section 4.3.1, with a maximum of 2 and 4 replicas per task, $C = 0.98$ and $q = 0.05$.
- FR: the fault tolerant scheduling based on the failure rate of each node, as described in Section 4.3.2.
- EC: Competition scheduling (see Section 4.1.2) with score f_1 based on the expected survival time of nodes (see Section 4.2.1).
- BFC: Competition scheduling (see Section 4.1.2) with score f_2 based on the best node-to-task fit (see Section 4.2.2).
- EGC: Competition scheduling in groups (see Section 4.1.3) with score f_1 based on the expected survival time of nodes (see Section 4.2.1) and group size $G = 10$.
- BFGC: Competition scheduling in groups (see Section 4.1.3) with score f_2 based on the best node-to-task fit (see Section 4.2.2) and group size $G = 10$.

Note that FCFS, EC, BFC, EGC and BFGC are tested in a P2P setting (each node has its own scheduler), while WQR, WQR-FA and FR are implemented as defined by their authors, using a central scheduler. The reason to choose $G = 10$ in EGC and BFGC is explained in Section 4.4.4.

4.4.3 Gathered metrics

During the experiments we gather the following metrics:

- System metrics. *Make-span*: the time, in seconds, required to execute the complete set of tasks inserted into the queue. The minimum make-span considering zero overheads would be $\frac{W}{n}$. *Throughput*: the number of tasks completed per second. It can be computed as $\frac{Num\_tasks}{Make-span}$.
- Per-node metrics. We dissect *node utilization*, extracting the *idle time* (time spent doing nothing), the *wasted time* (time used for aborted and cancelled executions), the *useful time* (time used for successful executions) and the *off-line time* (time while the node is not part of the system).
- Per-task metrics. We dissect *tasks overheads* (the time spent in the queue by each task), extracting the *waiting time* (time spent while waiting to be scheduled) and the *wasted time* (time spent in aborted executions).

Regarding the waiting time, we want to remark that this time is usually measured since the moment the task is inserted in the queue. However, in our experiments all tasks are inserted simultaneously at the beginning of the simulation. For this reason, we redefine this term, and use it to refer to the time spent by a task *while being at the head of the queue.*

4.4.4 Choosing the group size

In tests involving EGC and BFGC (that is, competition in groups) we need a group size. This has been fixed to 10, but the choice has not been arbitrary. In order to select a good value for this parameter, we ran an experiment with different values of G (from 2 to 100) for a particular scenario: mixed system with medium workload and BFGC scheduling. We measured system throughput and the waiting time of tasks, and plotted the results in Figure 4.3. It can be observed that, for values of G higher than 10, there is almost no improvement in terms of throughput; however, we can see how the waiting time increases with G. We have chosen $G = 10$ because it shows the advantages of scheduling in groups without incurring into excessive scheduling delays. A thorough analysis of the influence of G in the performance of EGC and BFGC is proposed as future work.

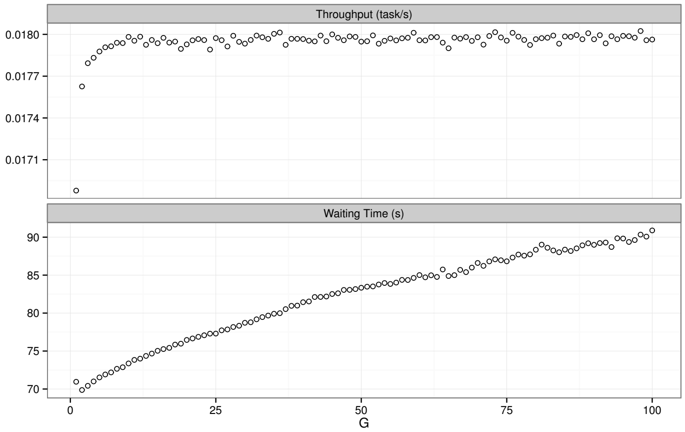


Fig. 4.3: System throughput and task waiting time, for the BFGC algorithm using different values of G, for the mixed system executing a mixed workload.

4.5 Analysis of results with exponentially distributed failures

In this section we analyze the results of the experiments described before with exponentially distributed failures, considering different scenarios and scheduling techniques. The baseline results will be those obtained with FCFS, but we will also compare the results of our proposal against other failure-aware policies.

Note that we do not expect great improvements with any proposal (compared against FCFS) in extreme situations, such as one with a majority of stable nodes to which users submit small tasks: all policies will extract the maximum potential of the system, because task abortion will be a rare event. In the opposite extreme we can envision a very unstable system to which users submit very long tasks. In this case, tasks will be frequently aborted and require re-execution, therefore nodes will spend most of their time performing useless computations. However, we can still try to reduce this waste of resources. In general, our policies are expected to improve system performance by increasing the probability of a correct execution at the first attempt, although some penalties could be expected in BFGC in the form of increased per-task waiting time.

When analyzing simulation results, we focus first on system-level metrics, then on the utilization of nodes and, finally, on the waiting times experienced by tasks – that reflect the perception that a user would have of the HTC system. As some proposals are knowledge-based, we include a subsection that discusses the effects on performance of inaccurate user-provided task length estimations.

4.5.1 System metrics

Figure 4.4 summarizes the main results from the point of view of the system: the make-span for the nine scenarios (of stability and workload), for all the scheduling algorithms under evaluation. As the duration of all workloads is $W = 10^9$ seconds, and the number of nodes in the system is $n = 1000$, the optimum value of make-span (for zero overheads) would be $\frac{W}{n} = 10^6$ seconds.

The first row of the figure corresponds to stable systems. In these, the choice of scheduling policy does not have a significant influence. In fact, the make-span obtained by FCFS is close to the minimum. However, BFGC is capable of squeezing some improvements: the differences between both policies are 1.37%, 1.41% and 3.33% for small, medium and large workloads respectively. The remaining non-trivial policies achieve similar results, although not for all workloads.

For the scenarios where there is enough diversity of nodes and tasks, those in which the proportion of unstable nodes is in the range 50%−90%, we can see that failure-aware policies contribute to shorten considerably the make-span. As expected, these policies enhance the correct distribution of tasks among the different types of nodes, so that the number of re-executions decreases and, thus, make-span improves. We can also see that the best policy is BFGC. Allowing nodes to choose, from a set of tasks, those that better fit into its expected lifetime seems to be a correct strategy from the point of view of system-level task throughput. The improvements over FCFS obtained by the BFGC policy are 16.88%, 16.97% and 15.92% for the mixed scenarios, while for the unstable scenarios the improvements of BFGC are 19.13%, 20.46% and 17.79%.

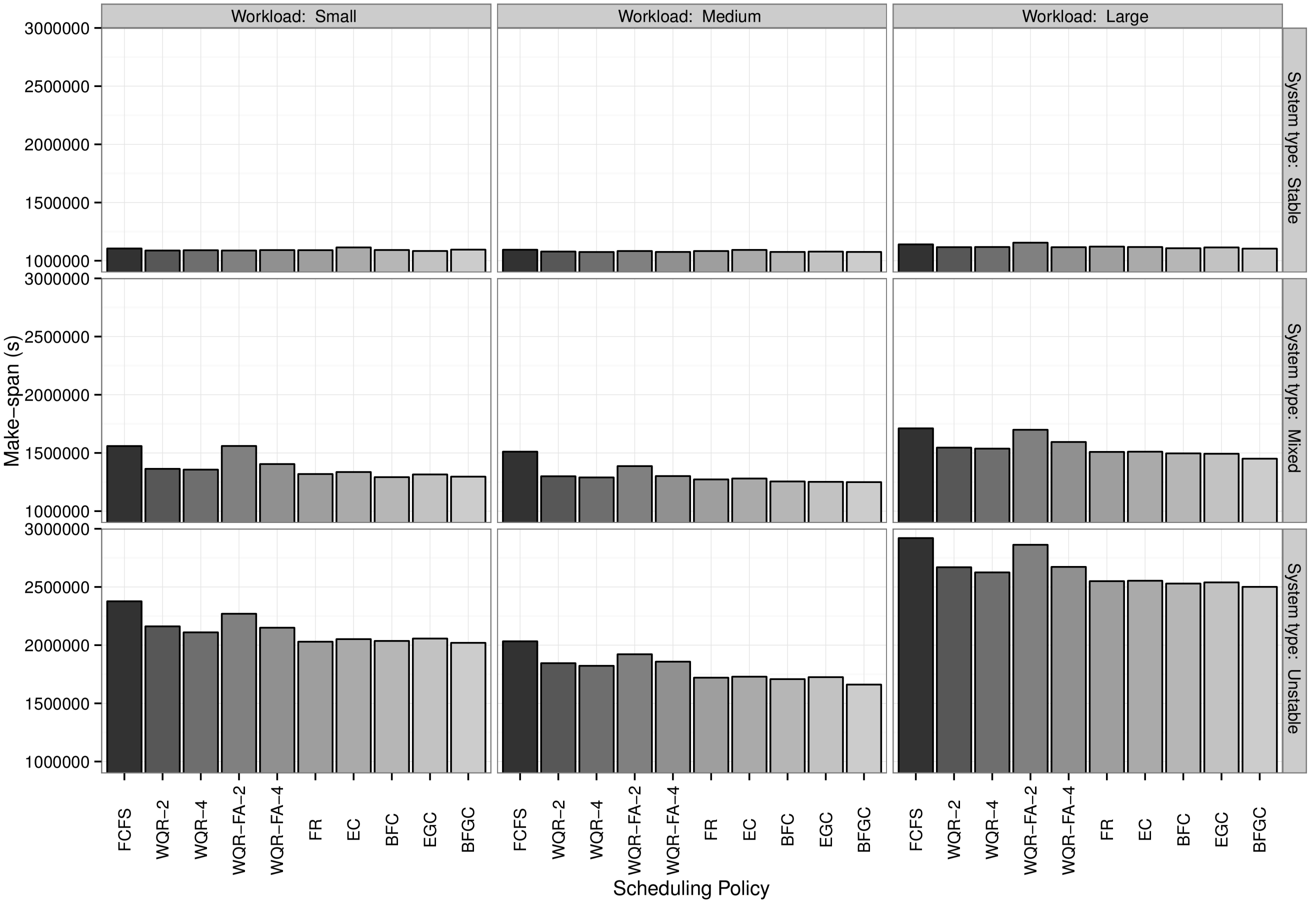


Fig. 4.4: Make-span using different scheduling policies for different scenarios (combinations of node stability and task size). The ideal make-span is 1000000.

As expected, FR and EC exhibit a very similar behaviour, because their purposes and metrics are similar. Scheduling in groups (EGC, BFGC) is better than scheduling for the task at the head of the queue, but only when using the f_2 metric (best node-to-task fit). WQR is not competitive, due to the overheads imposed by replication, and the failure-aware variations (WQR-FA-2, WQR-FA-4) are even worse. This behaviour of WQR scheduling is explained in [98], where authors tested their proposals with different number of tasks in the workload: when the number of tasks per node in the workload is small (under 50), WQR-FA outperforms WQR. However, when this ratio increases, WQR is relatively better. In our experiments, the number of tasks per node in the workloads vary from 77 to 395, which are bad settings for WQR-FA.

4.5.2 Node utilization

We have plotted in Figures 4.5 and 4.6 the results about node utilization. In order to simplify graphs and explanations, we have removed the data points corresponding to WQR-2 (which are worse than those of WQR-4), WQR-FA-2 (which are worse than those of WQR-FA-4), and all of our policies except BFGC (because the remaining three perform worse than this one).

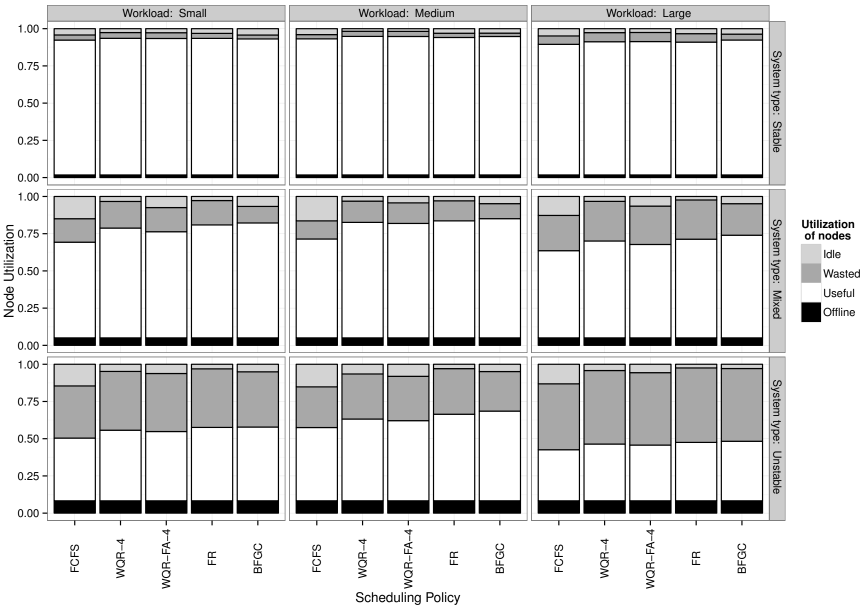
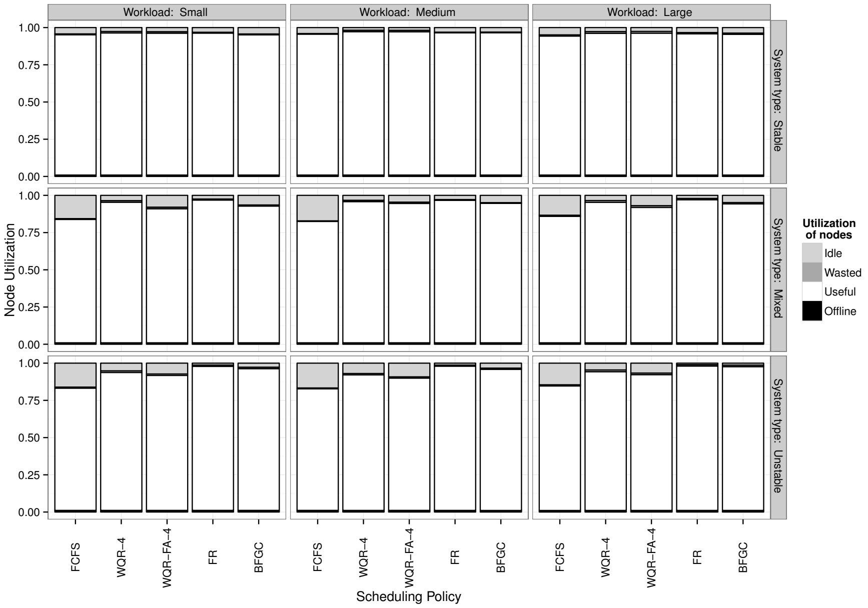
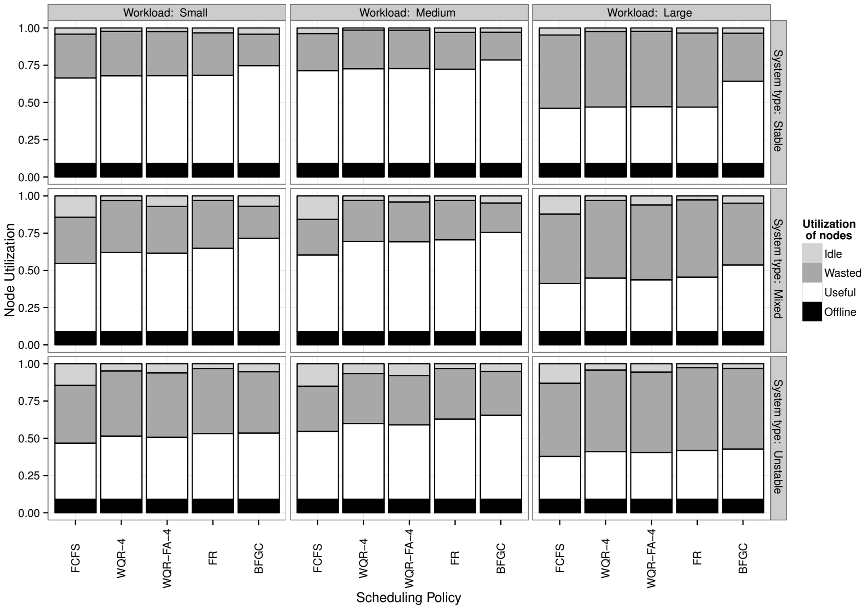


Fig. 4.5: Utilization of nodes for different scenarios (combinations of node stability and task size). Average for all nodes.



(a) Only stable nodes



(b) Only unstable nodes

Fig. 4.6: Utilization of nodes for different scenarios (combinations of node stability and task size) for stable (a) and unstable nodes (b).

In Figure 4.5 we can see a dissection of how nodes spend the time, averaged over all nodes; time is split into useful time, idle time, wasted time and off-line time. Then, Figure 4.6(a) considers only the stable nodes in the system, and Figure 4.6(b) focuses on the unstable nodes.

In general, the utilization of failure-aware scheduling policies results in an increment of the node's useful time for all the scenarios, even in the most stable ones. BFGC is the policy achieving the highest ratios of useful time, and the lowest of wasted time. WQR-4 and FR are respectable runner-ups.

Focusing on stable nodes, see Figure 4.6(a), we can see that FCFS does not maximize their useful time, leaving them empty for sizable periods. All the remaining policies do a better job, reaching a useful time close to 100%. We do not see wasted or off-line periods because these nodes rarely fail.

If we observe Figure 4.6(b), we can see why BFGC is the best policy: its choice of short tasks for unstable nodes results in a significant utilization of these nodes. In the remaining policies, unstable nodes are ignored (idle time) or process tasks that are too long for them, resulting in excessive aborts and re-executions (wasted time). Note, however, that BFGC is a good policy only when there is enough node and task diversity.

The reader may have noticed, when observing Figures 4.4 and 4.5, that the make-span and the per-node useful time in the case of medium workloads (those with a majority of medium tasks) is, for all scheduling policies, better than that obtained with small workloads, but these metrics get worse for large workloads. This non-linear effect requires further examination.

We must take into account the overheads derived from the scheduling process that, although relatively small, are incurred by each scheduled tasks: I/O operations, time between scheduling attempts, etc. But the figures clearly show that the main source of overhead is the wasted time (e.g. re-execution of aborted tasks due to node failures). Notice, too, that the number of tasks in a workload depends on the average task size, because the total duration of all workloads is fixed.

In Table 4.1 we have summarized some metrics for each type of workload, only for the unstable system with the FCFS policy (this has been done for illustrative purposes, as the numbers for other scheduling policies follow the same pattern). We can see the number of jobs per workload (fewer jobs means less scheduling overhead), the measured number of executions per task (the closer to one the better, because all the excess comes from re-executions) and the global make-span for consuming the whole workload (the closer to 10^6 the better). Medium workloads result in fewer tasks being scheduled, compared to small workloads, but the number of re-executions does not increase drastically, yielding a better overall behaviour. However, although the number of tasks in the large workload is small, the number of re-executions increases drastically (because many tasks are too long, given the on-line periods of the nodes, and they are rarely completed at the first attempt). This explains why the make-span for this workload is substantially longer.

	Small	Medium	Large
Number of tasks	395895	216637	77797
Mean number of executions per task	1.413	1.641	3.633
Make-span (s)	2375664	2018959	3036418

Table 4.1: Number of tasks, make-span and mean number of executions per tasks for each workload, for the unstable scenario under the FCFS scheduling.

4.5.3 Task overheads

We have measured and dissected the overheads suffered by tasks when scheduled using different policies, plotting the results in Figure 4.7. We can see how the waiting times are almost negligible, compared with wasted times due to re-executions. Note that as waiting time we only measure the time spent by tasks at the head of the queue. Also, in group-based scheduling policies (such as BFGC) the waiting time is measured as zero for those tasks executed in advance of their turn.

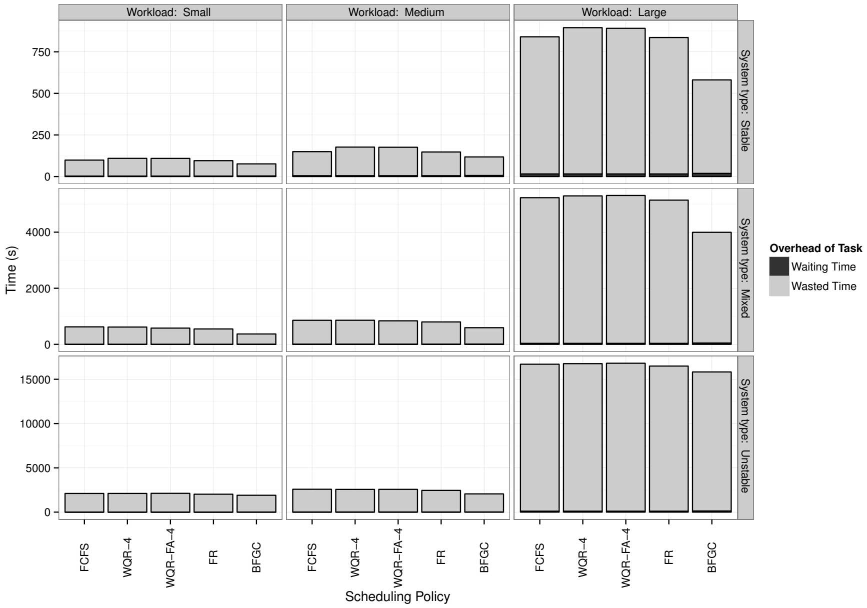


Fig. 4.7: Overheads of tasks for different scenarios (combinations of node stability and task size). Note the different y-axis scale for each row

We can observe that replication-based techniques, namely WQR and WQR-FA, cause increased wasted times. This is because, with these policies, tasks can be aborted not only because of node failures, but also because when a replica finishes, the remaining ones are cancelled. FR and, especially, BFGC waste less resources. In scenarios with enough stable nodes to execute the large tasks (see upper and middle row of Figure 4.7), the benefits of BFGC compared with the remaining policies are worth noticing.

Although this is not visible in the graph, BFGC can cause an increase of waiting time in some scenarios, in particular the one with mixed nodes and a majority of large tasks. This happens because, with this policy, the task waiting at the head of Q may be skipped in favour of other tasks better suiting the characteristics of the available nodes. However, the reduction of wasted time induced by this policy more than compensates this drawback. To illustrate this issue, we have summarized in Table 4.2 the number of non-delayed tasks (those that wait at the head of Q less than τ_s seconds), together with the average waiting time and the average wasted time, for policies FCFS and BFGC. We also include in the table the standard deviation (σ) of both metrics. This information corresponds to the execution of a medium workload in a mixed system. The data is averaged for all the tasks forming the workload, and also dissected for the different types of tasks of the workload. We can observe how the reordering of tasks performed by BFGC results in longer waiting times, but mainly for medium and small tasks, because large tasks are prioritized in the stable nodes. This reordering also results in higher values of wasted time in small (20.8 seconds for FCFS vs. 34.5 seconds for BFGC) and medium (428.3 vs. 486.5) tasks, but much lower wasted time for long tasks (4952.2 vs. 1378.6). Averaging all tasks, the wasted time drops from 837.6 to 529.9 seconds, as reflected in Figure 4.7. The difference is useful time in BFGC, explaining the globally better make-span.

As a summary of this and the previous subsections, we conclude that in those non-homogeneous scenarios composed of a variety of nodes and tasks, failure-aware scheduling policies result in improved task throughput. They try to avoid sending tasks to nodes not capable of completing them. This is the basis of FR, which is a good option despite its simplicity. However, BFGC goes a step further and assigns tasks to nodes looking for the best fit between the expected lifetime of the node and the task length, and experiments have proven that this is a successful approach, as wasted time is drastically reduced. Policies based on replication (WQR-based) are worse than FR and BFGC, as they cause excessive wasted time.

4.6 Experimentation with non-exponentially distributed failures

When modelling the on-line and off-line times of nodes in a distributed system, two main approaches can be found in the literature. Often it is assumed that the failure/reparation events can be represented using exponential distributions, but Weibull distributions are also commonly used. In the previous

Small tasks (10 %)					
Policy	Non-delayed tasks	Waiting time (s)		Wasted time (s)	
		Mean	σ	Mean	σ
FCFS	83.105%	6.054	7.478	20.761	125.656
BFGC	46.177%	11.920	11.549	34.464	166.042
Medium tasks (80 %)					
Policy	Non-delayed tasks	Waiting time (s)		Wasted time (s)	
		Mean	σ	Mean	σ
FCFS	78.456%	7.097	8.521	428.313	1239.137
BFGC	63.569%	10.693	17.888	486.502	1406.110
Large tasks (10 %)					
Policy	Non-delayed tasks	Waiting time (s)		Wasted time (s)	
		Mean	σ	Mean	σ
FCFS	65.000%	10.653	11.999	4952.180	9328.210
BFGC	81.600%	6.166	16.971	1378.569	4011.973
All tasks					
Policy	Non-delayed tasks	Waiting time (s)		Wasted time (s)	
		Mean	σ	Mean	σ
FCFS	77.584%	7.346	8.908	837.591	3431.536
BFGC	63.617%	10.366	17.324	529.869	1812.231

Table 4.2: Percentage of non-delayed tasks (waiting time $\leq \tau_s$), waiting time (average and σ) and wasted time (average and σ) for the mixed system with medium workload. The bottom block gathers the results considering all the tasks in the workload, while the other ones consider only a particular class of tasks.

section we have discussed the results from a set of experiments where our choice was to use exponential distributions. In this section we repeat those experiments, using Weibull distributions to control the failure and reparation behaviour of the (simulated) nodes. These additional experiments show that our competition-based failure-aware scheduling proposal also performs successfully under the Weibull distribution used to model nodes' behaviour.

A Weibull distribution has two parameters: *scale* and *shape*. Regarding the shape parameter, Javadi et al. analysed in [79] several failure traces from real systems and concluded that, when describing the failure distribution of a certain node, this parameter takes values in the range $(0.33 - 0.85)$, which implies that the failure rate decreases over time. Thus, we have decided to fix this parameter to 0.7 for both, stable and unstable, types of nodes. With respect to the shape parameter of the repair distribution, in the same study they concluded that it takes values in the range $(0.35 - 0.65)$, so we have fixed this parameter to 0.5 for both types of nodes.

As regards to the scale parameters of the Weibull distributions used, we have followed this approach to select them. Given a target, expected on-line, or off-line, time $E[X]$ for a certain node, the λ parameter, *rate*, of an exponential with that mean is simply:

$$E[X] = \frac{1}{\lambda} \tag{4.11}$$

However, for the Weibull distribution with parameters λ (scale) and k (shape), the expression is:

$$E[X] = \lambda \mathit{\Gamma} \left(1 + \frac{1}{k}\right) \tag{4.12}$$

Consequently, as the shapes have been already fixed, we can compute the scale of a Weibull distribution with the target expected mean as:

$$\lambda = \frac{E[X]}{\mathit{\Gamma}\left(1 + \frac{1}{k}\right)} \tag{4.13}$$

Using this equation, and in order to achieve the same target failure and reparation times used in the experiments with exponential distributions, the parameters selected to model the nodes using Weibull distributions are:

- Stable nodes:
  - Failures:
    - $\mathit{scale} = 789999.5$.
    - $\mathit{shape} = 0.7$.
  - Reparations:
    - $\mathit{scale} = 5000$.
    - $\mathit{shape} = 0.5$.
- Unstable nodes:
  - Failures:
    - $\mathit{scale} = 7899.995$.
    - $\mathit{shape} = 0.7$.
  - Reparations:
    - $\mathit{scale} = 500$.
    - $\mathit{shape} = 0.5$.

The remaining parameters used to run the experiment are those described in Section 4.4.

The results of the experiments comparing different scheduling algorithms, including distributed and failure-aware ones, are summarized in Figures 4.8, 4.9, 4.10 and 4.11. As can be seen, results are very similar to those presented in Section 4.5, although the differences between algorithms are now narrower. In some cases, FR is slightly better (0.2%) than our proposals in terms of make-span. However, note that FR is a centralized algorithm with lower overheads than our distributed proposals. Therefore, this set of experiments confirm the good behaviour of our algorithms. In particular, BFGC is globally the best of the tested options.

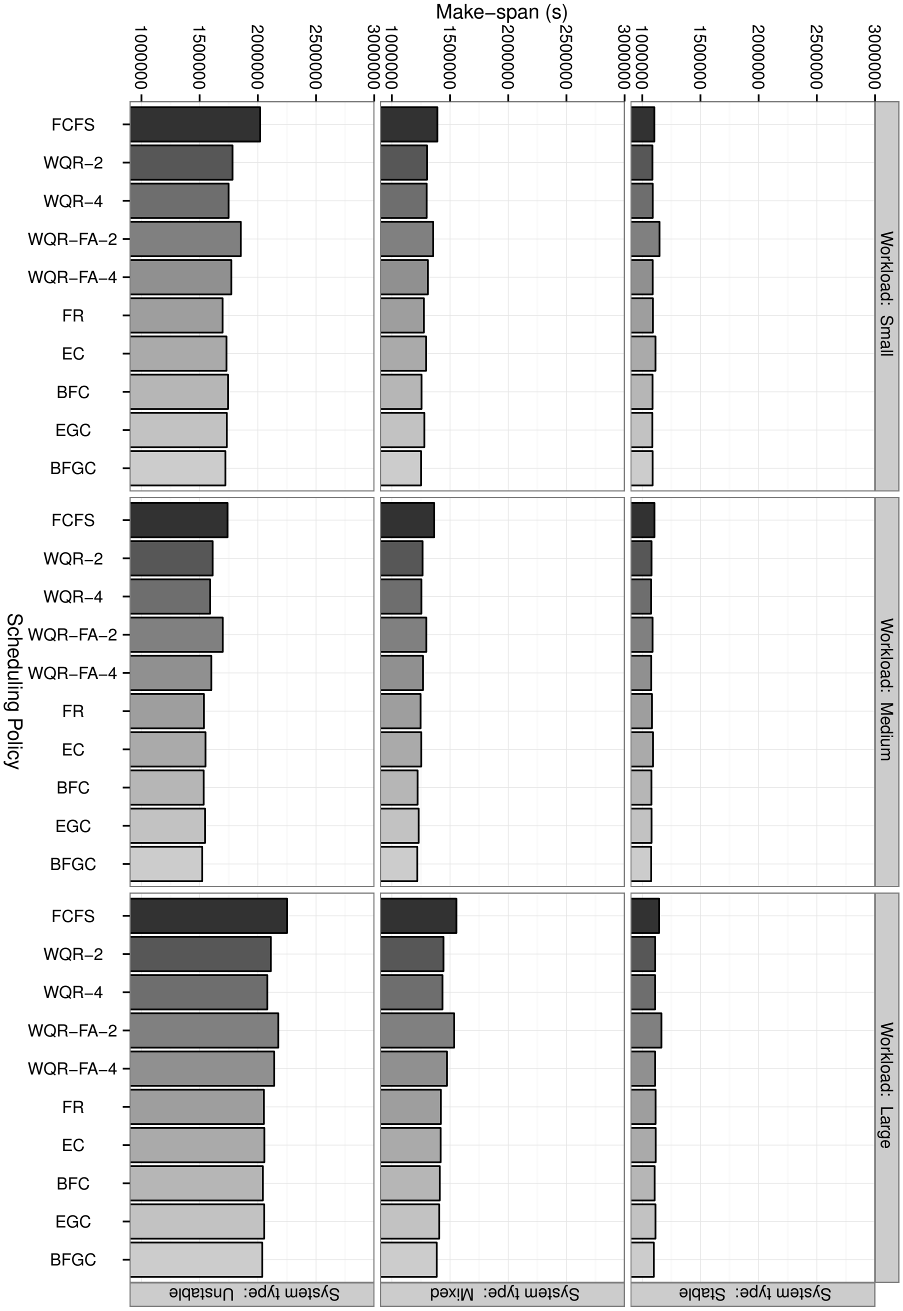


Fig. 4.8: Make-span using different scheduling policies for different scenarios (combinations of node stability and task size). The ideal make-span is 1000000. Failures and reparations follow Weibull distributions.

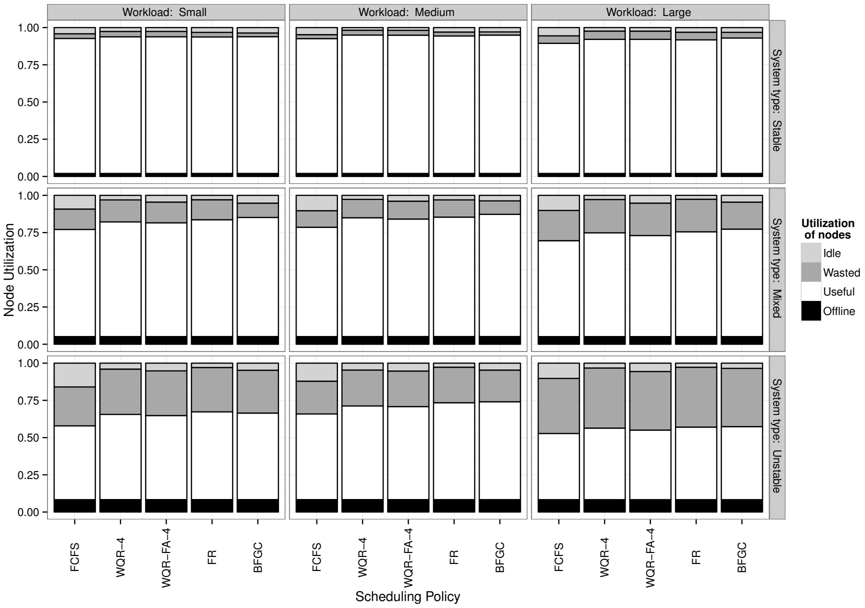


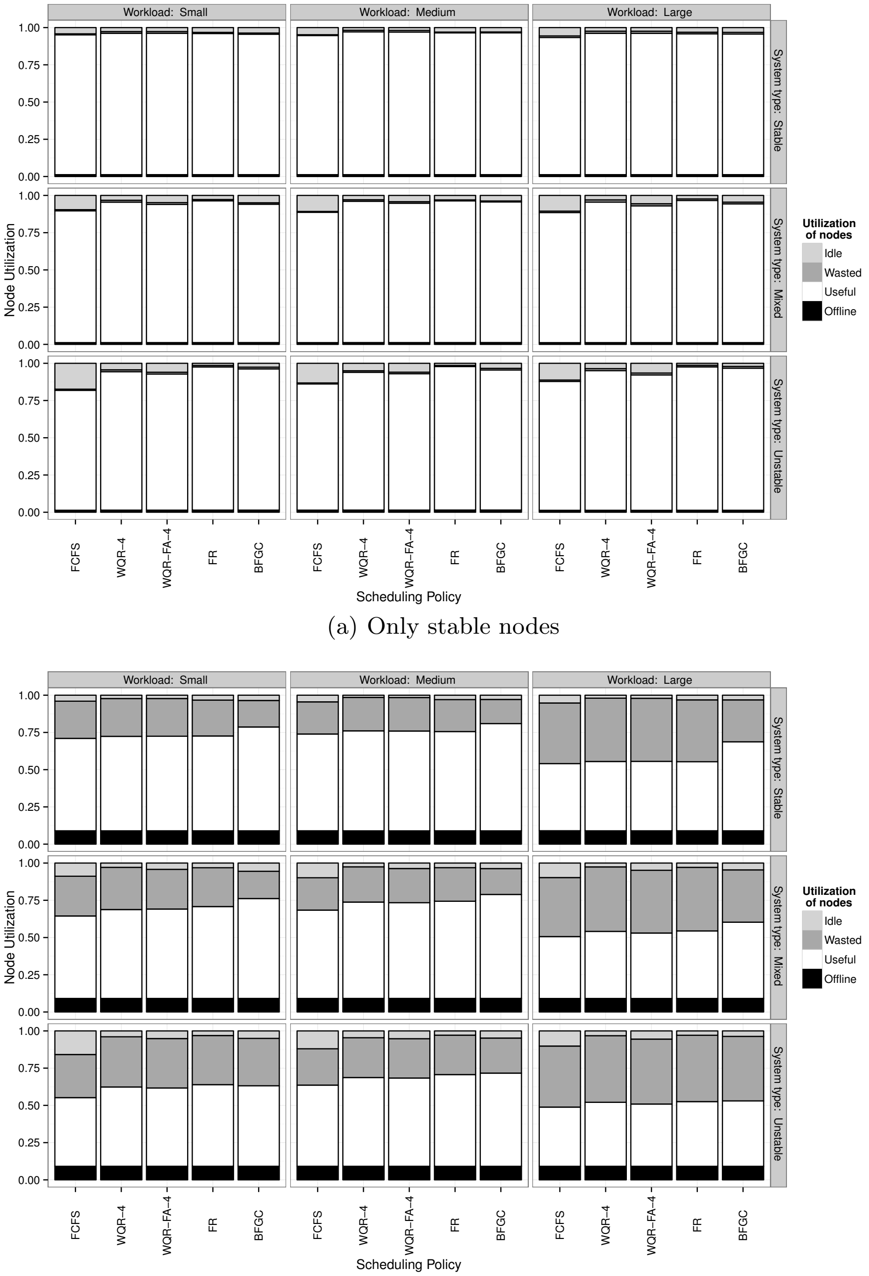
Fig. 4.9: Utilization of nodes for different scenarios (combinations of node stability and task size). Average for all nodes. Failures and reparations follow Weibull distributions.

4.6.1 Dealing with inaccurate estimations of task durations

Note that the most effective policies tested in this chapter are knowledge-based, that is, they use the user-provided estimation of the execution time (length) of the tasks submitted to the HTC system. In our simulation, we have used a task's length as the *exact* run time, that is, the time while a worker node is busy executing the task. We know, however, that these estimations may not be accurate, and real run times may be completely different.

It is known that users tend to overestimates the tasks' runtime in order to avoid having the task killed before completion [105], a common practice in scheduling systems for supercomputers. However, we have not considered this option: in the experiments, all tasks run until completion – unless they fail after 100 execution attempts, something that we consider a pathological situation.

In BFGC, if the user's estimate for a task exceeds its actual run time, the effects will not be negative: the assigned resource will be released sooner than planned. Note, thought, that if the task was assigned to a stable node, the scheduler could have found a better match with a less stable node. In contrast, if a task with a short predicted run time, assigned to an unstable



(a) Only stable nodes

(b) Only unstable nodes

Fig. 4.10: Utilization of nodes for different scenarios for stable and unstable nodes. Failures and reparations follow Weibull distributions.

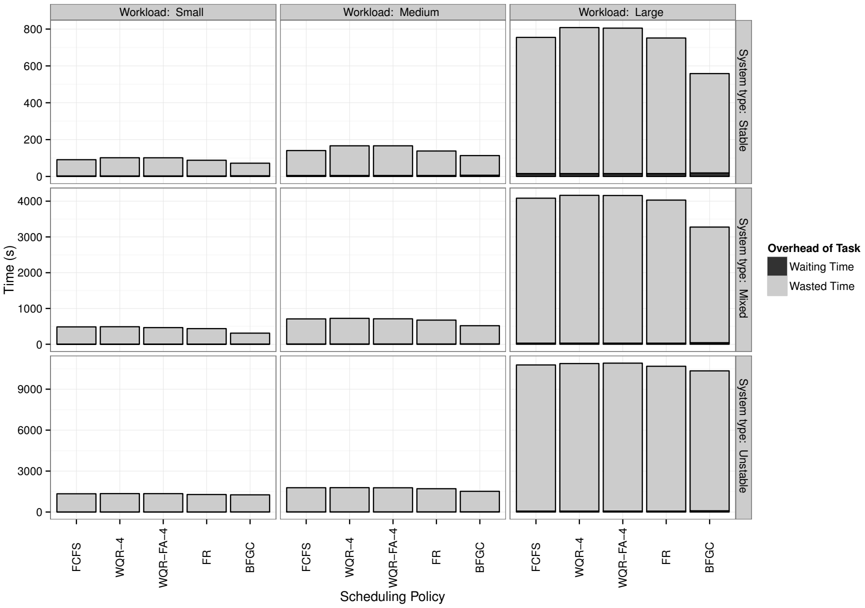


Fig. 4.11: Overheads of tasks for different scenarios (combinations of node stability and task size). Note the different y-axis scale for each row. Failures and reparations follow Weibull distributions.

node, runs longer than expected (user underestimation), it may be aborted and need re-execution.

We wanted to assess the effects of the inaccuracy of user-provided length estimations in the effectiveness of the scheduling algorithms analysed in this chapter. To do so, we introduce in our simulation-based experiments an inaccuracy factor k. Execution times used by tasks are no longer the lengths declared in the workload; instead, they are recomputed as follows: for each task i of length l_i, the run time used in the simulation, r_i, is chosen uniformly at random from the interval $[l_i/k, l_i * k]$. In the experiments, k is varied from 1 (accurate estimation) to 3 (actual run times may be up to three times shorter / longer than predicted).

In Figure 4.12 we show the make-span obtained by FCFS, WQR-4, WQR-FA-4, FR and BFGC for three representative scenarios (stable system with a majority of short tasks, mixed system with a majority of large tasks, and unstable system with a majority of medium tasks). As a reference, for these three scenarios with accurate predictions ($k = 1$), the make-span reductions of BFGC over FCFS are 1.37%, 15.92% and 20.46% respectively. Note that we have included in the plots a variation of the BFGC, BFGCE, that will be explained later. We have also included the ideal make-span $\frac{W}{n}$, where W is the sum of all tasks' run times. This value increases with k because, on average, tasks will take longer run times than predicted.

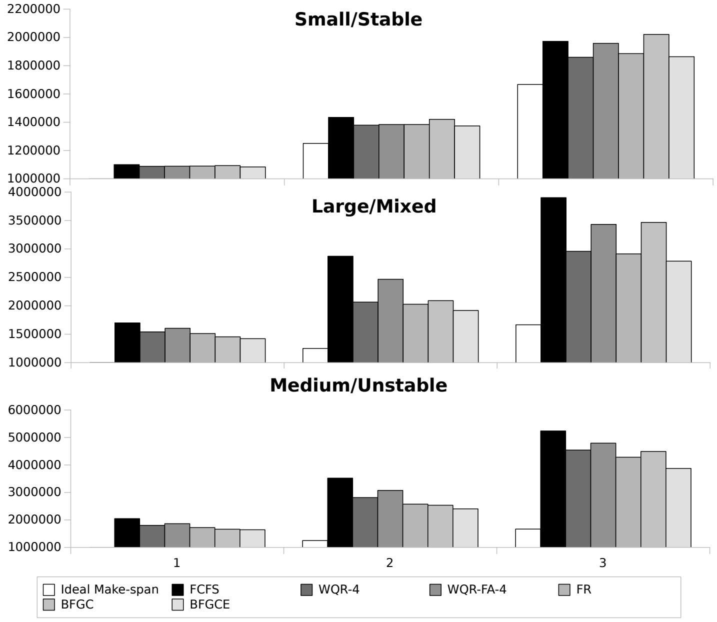


Fig. 4.12: Make-span for several scheduling algorithms for different scenarios and inaccuracy factors (k). Time computed after all tasks in the workload have been completed (or dropped after 100 unsuccessful trials). The ideal make-span for $k = 1$ is 1000000.

For the second and third scenarios, failure-aware policies fall somewhere in between FCFS and the ideal make-span. However, the behaviour of BFGC deteriorates clearly, when compared against other policies, for large values of k. In the first scenario, BFGC can be much worse than the plain FCFS and the remaining policies.

It is not difficult to explain this behaviour: BFGC tries to find a good task-to-node match, but the *actual* task length can be very inadequate, given the stability characteristics of the chosen node. Then, re-executions will be a frequent event. In fact, we have observed that, for large values of k, some long tasks are finally dropped after 100 attempts, and this happens regardless of the scheduling policy. We have gathered in Table 4.3 the minimum value of k at which the scheduler starts dropping tasks, together with the total number of dropped tasks in the corresponding experiment. Note that only the BFGCE algorithm (will be discussed later) can complete all tasks in all scenarios, regardless of k (at least in the considered $[1, 3]$ range).

Algorithms	Small/Stable		Large/Mixed		Medium/Unstable	
	k	Dropped tasks	k	Dropped tasks	k	Dropped tasks
FCFS	-	-	1.5	4	1.2	7
WQR-4	-	-	-	-	1.7	1
WQR-FA-4	3.0	1	1.7	6	1.3	11
FR	-	-	-	-	1.6	1
BFGC	-	-	2.5	1	2.2	1
BFGCE	-	-	-	-	-	-

Table 4.3: Minimum inaccuracy factors for which the scheduler starts dropping tasks for different workload/system scenarios, together with the number of dropped tasks.

Dropped tasks is a symptom of a bad behaving system: very long tasks require many execution attempts to be completed or dropped, distorting the metrics when considering the majority of successfully executed tasks. In order to reduce this distortion, we have plotted in Figure 4.13 the make-span of the first 99.9% completed tasks. The remaining 0.1% includes all the dropped ones, in all the experiments. Plots are now much more clear: BFGC and its variation are the best options, even with severely bad inaccuracy factors.

Nevertheless, we still need to deal with that small percentage of dropped tasks. A good scheduling policy must be able to find the right nodes to execute them, even when the knowledge they have about the tasks' lengths is inaccurate. Now we introduce *BFGCE*, which is BFGC with a correction of the user-provided estimation of the length of a task. BFGCE operates exactly like BFGC but, when a task is aborted, the user-provided length is corrected (actually, increased using a factor e_c), and this corrected value is used in further scheduling attempts. The system assumes that the length prediction was inaccurate, and that, next time, the task should be assigned to a more stable node. After some preliminary tests, we have set $e_c = 10\%$, although we plan to make a deeper analysis of the effects of this parameter in future works (maybe considering a different correction per user). The figures and table show that BFGCE succeeds in completing all tasks and results in the shortest make-spans for all scenarios and inaccuracy factors.

4.7 Related work

In the literature we can find several works that study the scheduling problem in HTC systems in the presence of failures, trying to maximize the fault tolerance of the system. Authors of [106] propose several resource provisioning techniques for cloud environments that use checkpointing to minimize the effects of failures in applications running in supervised clouds. In [101], Anglano et al. propose WQR-FT, a fault tolerant variation of the WorkQueue with Replication (WQR) scheduling algorithm for HTC systems [100] that,

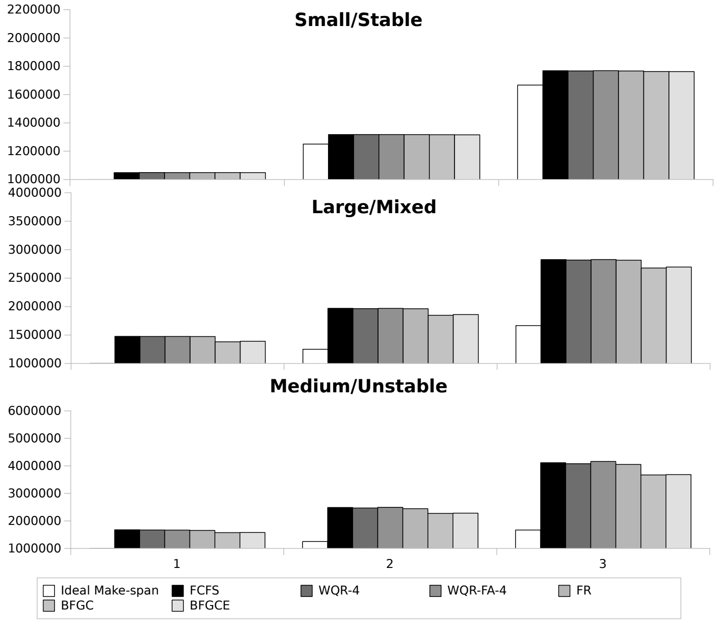


Fig. 4.13: Make-span for several scheduling algorithms for different scenarios and inaccuracy factors (k). Time computed after 99.9% of the tasks in the workload have been completed. None of the tasks has been dropped. The ideal make-span for $k = 1$ is 1000000.

using replication and checkpointing, aims to reduce the effects of failures. Also, in [107] Bansal et al. propose a modification of WQR-FT where the number of replicas of each task is selected depending on the ratio of tasks successfully executed in the system: if most of the tasks are completed, less replicas are launched per task.

Note that the proposals described in the previous paragraph are designed to deal with the consequences of a failure, and can be classified as *fault tolerant* scheduling techniques. Our focus is on *failure-aware* techniques that try to minimize the number of aborted tasks derived from inadequate scheduling decisions. These approaches can complement each other.

Supercomputers for high performance computing are large-scale systems, managed by a central scheduler, in which worker nodes can fail. Authors of [108] and [109] present and evaluate scheduling proposals in which the system partition (collection of nodes) in which a parallel task will run is selected by taking into consideration the node's *resilience*, computed from failure models of nodes. A limitation of these works is that the experiments are based on failure logs, and *future* information is used to compute resilience. Additionally, they do not consider scheduling in groups.

Several works describing HTC systems for grids, including desktop grids, propose failure-aware schedulers, but they differ in the way the reliability of nodes is modeled. In [99] each node is assigned a failure rate computed by measuring the number of tasks successfully completed. Authors of [98] propose a modification of WQR-FT that builds a per-node failure model using the past on-line times, together with a prediction method described in [102]. The desktop grid system described in [110] characterizes the cyclic behaviour of participating nodes (availability) using Markov models. Finally, several works [111, 112, 113] model nodes' behaviour using histograms. The information provided by these models is used to avoid sending tasks to nodes that may not complete them, but the possibility of scheduling in groups to find good task-to-node matches is not part of these proposals.

There is a substantial body of literature analysing knowledge-based scheduling for HTC with focus on fault tolerance, which includes some works cited before: [99, 79, 108, 109, 110, 111, 112, 113]. With respect to knowledge-free algorithms, examples of this kind of algorithms are WQR [100] and its variations [101, 98, 107].

The weakest point of knowledge-based techniques is precisely the need of user-provided *estimations* about the resources required by submitted tasks, namely, the expected duration (we use the term "length"). These estimations can be very imprecise in some contexts: a user can hardly known *a priory* the length of a task whose behaviour depends on the nature of the input files and parameters, not to mention the characteristics of the particular node in which it will run. In this chapter we have ignored the latter effect, assuming that all nodes are homogeneous, or that it is possible to apply a sort of "adjustment factor". Although it is well known that users do not provide accurate estimates of the length of their tasks, authors of [114] argue that there is a strong correlation between user estimations and actual lengths. Therefore, tasks with longer estimated duration should, as a general rule, be assigned to the most stable nodes of the system. We have discussed in Section 4.5 to what extent the accuracy of user-provided estimations affects the performance of our knowledge-based, failure-aware scheduling proposals.

4.8 Conclusions

In this chapter we have presented several policies that can be used in an HTC system in order to improve the scheduling process in the presence of failures. These techniques have been proposed and evaluated in an HTC-P2P environment, but could be used in other platforms, such as desktop/grid computing systems or supercomputers. Moreover, they can be combined with other mechanisms for fault tolerance, such as checkpointing/restarting and replication. The utilization of the information about previous failures together with the expected duration of tasks can be of help for nodes when selecting a task to be executed from those waiting in the scheduling queue. Taking into account this information reduces the number of failures while executing a task and, thus, reduces the number of re-executions per task, so the efficient utilization of nodes improves and the overheads suffered by tasks are reduced.

In order to test our proposals, we have simulated the scheduling process in an HTC system where each node executes its own scheduler so it can make its own decisions about which task to execute. We have also implemented, for comparison purposes, other scheduling algorithms from the literature. Experimental results show that our failure-aware proposals do a good job finding appropriate task-to-node fits, decreasing wasted time and increasing system throughput. This is particularly true for BFGC. It is to be noted that these *distributed* schedulers perform better than the competitor, *centralized* approaches.

As our proposals are knowledge-based, we have also tested their behaviour when dealing with inaccurate estimations of user-provided task durations. Results state that, even with severe inaccuracy factors (up to $k = 3$), a minor modification of BFGC (namely, BFGCE, which corrects the estimation of the duration of a task when it needs to be re-executed) performs much better than the remaining policies tested in this work.

As future work, we aim to implement and test these techniques in a real HTC system; in particular, in the HTC-P2P system presented in Chapter 2. Then, we plan to dig further into these aspects:

- Competition-based scheduling must be complemented with adequate score functions. In this chapter we propose two, based on the properties of the exponential distribution, but others are valid. We could use different distributions (such as Weibull) or, as other researchers have done, characterize nodes using Markov models and histograms. It could be even possible to analyse the evolution of the performance of the system, tuning dynamically some parameters of the score functions in order to choose the values that maximize performance.
- BFGC and BFGCE schedule in groups, but we must improve our study about the impact of group size on performance. We could even consider the possibility of varying this value dynamically, taking into consideration the observed performance.
- Group scheduling could go a step further. Currently, a node only competes for the task that better fits its characteristics. However, it could compete also for the second best task in the group, or even for all the tasks in the group.
- The way BFGCE corrects user-provided estimations must be explored further. As hinted before, BFGCE could create a per-user accuracy model based on his/her previous records, adapting the correction factor through this model.
- The failure-aware policies could be complemented with a replication mechanism, in which the number of replicas would depend on estimations of the

average number of re-executions per task. This mechanism should improve the response time perceived by users.

5

Conclusions and future work

In this chapter we present the main conclusions of this dissertation together with specific directions for additional research. The end of the chapter is devoted to presenting the publications that have resulted from the work carried out during the development of this thesis.

5.1 Conclusions

This dissertation has focused on one important paradigm of distributed computing, High Throughput Computing, its requirements and how to address them using Peer-to-Peer technologies. To that extent we have (1) presented a novel HTC architecture totally distributed based on a P2P storage system, (2) analysed the fault-tolerance of this HTC-P2P system by modelling the availability provided by the underlying P2P storage system and (3) worked on different scheduling policies that could increase the efficiency of any HTC system, including HTC-P2P, by implementing failure-aware scheduling.

Particularly, in Chapter 2 we have addressed the concept of HTC system by distributing the responsibilities of the central manager among all the workers of the system. This HTC-P2P system has these characteristics: (1) lack of central management points, (2) disconnected operation and (3) flexible scheduling with support for (non-strict) FCFS execution order of tasks. The proposed system has been built on top of a P2P data-storage, Cassandra, executed by all the workers on the system. Experimentation using real clusters has been carried out in order to test the validity and scalability of our proposal, showing that the HTC-P2P proposal works and scales properly.

The fault-tolerance of our system has been addressed in Chapter 3. The availability of the HTC-P2P system depends on the availability provided by Cassandra. In order to analyse it, we have proposed two models of Cassandra's availability under different failure situations: (1) transient failures, those in which the node is off-line for some period of time without losing any information, and (2) memory-less failures, those in which a node completely loses the information stored in it. The models we propose are based on the stochastic modelling of replica groups using Markov chains. We have validated our models via experimentation with real Cassandra clusters. Results show that our models are quite accurate descriptors of the availability of Cassandra. By using these models, we can not only predict the availability of any Cassandra deployment, but also obtain hints to select the best possible configuration of Cassandra in terms of availability for any situation.

In existing HTC systems, when a worker node fails while executing some task, the failed task is usually reinserted into the system for re-execution. This causes a waste of resources and is usually solved by using fault-tolerance techniques, namely checkpointing and replication, that do not entirely solve the problem and can be the cause of additional misuse of resources. In Chapter 4 we have addressed this problem by testing alternative solutions as failure-aware scheduling, where each node considers a model of its reliability together with an estimation of each task's length to search for a good matching between tasks and nodes. Two different scheduling algorithms are proposed (competition scheduling and competition scheduling in tasks groups) and are used in combination with two proposed score functions. In order to test our approaches, we have performed simulation-based experimentation in which a comparison with state-of-the-art algorithms is made. Results show that our proposals improve the results obtained not only by the baseline FCFS but also by other scheduling failure-aware algorithms. System performance and the correct utilization of nodes increases by 20% in the best case just by making better scheduling decisions.

5.2 Future work

This thesis has opened multiple lines for further research. The HTC-P2P system that we present in Chapter 2 can be improved in several ways. We plan to enhance the system by allowing users to submit not only individual tasks, but also more complex jobs, including acyclic data flows or multi-pass applications. We want also to include fault-tolerance mechanisms such as checkpointing and replication.

With respect to the collisions problem of the HTC-P2P system related to the eventual consistency provided by Cassandra, we have proposed a mechanism to reduce it to manageable levels ($0.5-6\%$ depending on the system). We have plans to test other possible solutions, such as the inclusion of an external locking mechanism (Zookeeper) or the use of a novel mechanism included in more recent versions of Cassandra, lightweight transactions, that could serve as a locking mechanism as it implements Paxos, a consensus algorithm. For the moment we have not tested these possible solutions but we plan to do so in future releases of our HTC-P2P system.

With respect to the availability models of Cassandra presented in Chapter 3, while the transient failures model can be considered a good descriptor of Cassandra's behaviour, the memory-less failures model must be further improved. The β parameter (the Anti-Entropy repair rate) is currently estimated using the replica size and the number of objects, but it should also consider the size of the objects to be repaired. With respect to the ReadRepair rate (α in the model) we consider that this process is instantaneous, but it requires reading the target object from all the replicas, updating the stale ones. The cost of this process depends on the replica size and should take into account the size of the objects stored in the system. Finally, we have considered two different failure models for Cassandra but, in a real environment, actual failures can be of any of these types. It would be possible to build a model combining transient and memory-less failures by adding new states and transitions to the memory-less failure model. However, the necessary effort is not trivial as it implies a much larger set of states and transitions to describe all the possible situations in which the nodes could be.

With respect to the failure-aware algorithms presented in Chapter 4, we aim to implement them in the HTC-P2P system that we presented in this dissertation. The different proposals can be still improved. We must conduct a deeper study about the impact of the group size in the BFGC and BFGCE policies. Group scheduling can be improved if, instead of considering only one task for the competition, more tasks are considered for scheduling at the same time. With the BFGCE policy we have presented a way to correct user-provided estimations, but this proposal must be explored further. For example, per-user accuracy models designed, or models present in the literature to predict execution times could be incorporated.

In this dissertation we have proposed using failure-aware scheduling in the context of an HTC-P2P system. However, this type of scheduling could be also used in other kinds of HTC systems. The information generated and stored by our society is growing exponentially as devices capable of gathering data from their surroundings are rapidly becoming ubiquitous. This harsh increase in the volume of available information, and our interest in extracting valuable knowledge from it, have produced a new set of "big data" tools that share many properties with HTC distributed systems, and can, therefore, take advantage of the scheduling policies we have presented. We aim to test these techniques in actual big data frameworks, such as Apache Hadoop [115] or Apache Spark [116].

5.3 Publications

In this section we list all the scientific works published or submitted during the development of this dissertation. The papers directly related with the work presented in this dissertation are highlighted.

5.3.1 International Journals

- **C. Pérez-Miguel, J. Miguel-Alonso and A. Mendiburu. High throughput computing over peer-to-peer networks. *Future Generation Computer Systems.* Volume 29, Issue 1, January 2013, Pages 352-360.**
- **C. Pérez-Miguel, A. Mendiburu and J. Miguel-Alonso. Modelling the availability of Cassandra. *Submitted* to *Journal of Parallel and Distributed Computing.***
- **C. Pérez-Miguel, A. Mendiburu and J. Miguel-Alonso. Competition-based failure-aware scheduling for high-throughput computing systems on peer-to-peer networks. *Submitted* to *Journal of Cluster Computing.***
- C. Pérez-Miguel, J. Miguel-Alonso and A. Mendiburu (2010) Porting Estimation of Distribution Algorithms to the Cell Broadband Engine. *Parallel Computing.* Volume 36, Issues 10-11, October-November 2010, Pages 618-634, Parallel Architectures and Bioinspired Algorithms.

5.3.2 International Conferences

- C. Pérez-Miguel, J. Miguel-Alonso and A. Mendiburu. Porting Estimation of Distribution Algorithms to the Cell Broadband Engine. *Workshop on Parallel Architectures and Bioinspired Algorithms (WPABA) in conjunction with PACT 2009.* Raleigh, North Carolina. September 12-16, 2009.
- C. Pérez-Miguel, J. Miguel-Alonso and A. Mendiburu. Evaluating the Cell Broadband Engine as a Platform to Run Estimation of Distribution Algorithms. *CIGPU-2009 (GECCO 2009 Workshop, Tutorial and Competition on Computational Intelligence on Consumer Games and Graphics Hardware).* July 8-12, Montreal (Canada).

5.3.3 National Conferences

- **C. Pérez-Miguel, J. Miguel-Alonso y A. Mendiburu. Sistemas HTC sobre redes P2P. *XXI Jornadas de Paralelismo. CEDI* 2010, Valencia, Septiembre 8-10, 2010.**
- C. Pérez-Miguel, J. Miguel-Alonso and A. Mendiburu. Evaluation of the Cell Broadband Engine running Continuous Estimation of Distribution Algorithms. *XX Jornadas de Paralelismo.* 16-18 Septiembre, 2009,A Coruña.

5.3.4 Technical Reports

- **C. Pérez-Miguel, J. Miguel-Alonso y A. Mendiburu. Informe sobre Sistemas de Computación en Redes P2P. Informe interno EHU-KAT-IK-04-09.**

- C. Pérez-Miguel y J. Miguel-Alonso. Programación de sistemas basados en el Cell. Informe Interno EHU-KAT-IK-03-09.
- C. Pérez-Miguel y J. Miguel-Alonso. Programación SIMD para x86, AltiVec y Cell. Informe Interno EHU-KAT-IK-02-09.

A

Configuration of the ColumnFamilies required by the proposed HTC-P2P

```
CREATE KEYSPACE ks WITH strategy_class = 'SimpleStrategy'
  AND strategy_options:replication_factor = '3';

USE ks;

CREATE TABLE        users (
user_id             uuid,
username            text,
name                text,
email               text,
password_digest     blob,
created_at          timestamp,
updated_at          timestamp,
PRIMARY KEY         (user_id));

CREATE TABLE    queue (
bucket          int,
task_id         timeuuid,
user_id         uuid,
proc            text,
mem             text,
so              text,
disc            float,
libs            text,
tins            timestamp,
PRIMARY KEY     (bucket, task_id));

CREATE   INDEX   queue_proc   ON   queue   (proc);
CREATE   INDEX   queue_mem    ON   queue   (mem);
CREATE   INDEX   queue_so     ON   queue   (so);
CREATE INDEX queue_disc  ON queue (disc);
CREATE INDEX queue_libs  ON queue (libs);
CREATE INDEX queue_user  ON queue (user_id);

CREATE TABLE  tasks (
task_id      timeuuid,
user_id      uuid,
state        text,
proc         text,
mem           text,
so           text,
disc         float,
libs         text,
errormsg     text,
tins         timestamp,
texe         timestamp,
tend         timestamp,
modify_time  timestamp,
PRIMARY KEY   (task_id));

CREATE INDEX tasks_proc  ON tasks (proc);
CREATE INDEX tasks_mem   ON tasks (mem);
CREATE INDEX tasks_so    ON tasks (so);
CREATE INDEX tasks_disc  ON tasks (disc);
CREATE INDEX tasks_libs  ON tasks (libs);
CREATE INDEX tasks_user  ON tasks (user_id);
CREATE INDEX tasks_state ON tasks (state);

CREATE TABLE   workers (
worker_id     uuid,
user_id       uuid,
hostname      text,
proc          text,
mem            float,
so            text,
disc          float,
libs          text,
heartbeat     timestamp,
PRIMARY KEY    (worker_id));

CREATE INDEX workers_proc  ON workers (proc);
CREATE INDEX workers_mem   ON workers (mem);
CREATE INDEX workers_so    ON workers (so);
CREATE INDEX workers_disc  ON workers (disc);
CREATE INDEX workers_user  ON workers (user_id);
CREATE INDEX workers_libs ON workers (libs);

CREATE TABLE    files (
file_id        uuid,
user_id        uuid,
name           text,
basename       text,
parent_path    text,
size           int,
nchunks        int,
PRIMARY KEY     (file_id));

CREATE INDEX files_user ON files (user_id);

CREATE TABLE    chunks (
chunk_id       uuid,
file_id        uuid,
size           int,
sha1sum        blob,
data           blob,
PRIMARY KEY     (chunk_id));

CREATE TABLE    file_chunks (
task_id        uuid,
file_id        uuid,
PRIMARY KEY     (file_id, chunk_id));

CREATE TABLE    task_files (
task_id        uuid,
file_id        uuid,
filetype       text,
PRIMARY KEY     (task_id, file_id));

CREATE TABLE    task_workers (
task_id        uuid,
score          timeuuid,
worker_id      uuid,
PRIMARY KEY     (task_id, score, worker_id));

CREATE TABLE    blacklist (
user_id        uuid,
blocked        uuid,
PRIMARY KEY     (user_id, blocked));
```

References

1. C. resources. Torque. `http://www.clusterresources.com/torque`, December 2010.
2. Oracle. `http://www.sun.com/sge`, December 2010.
3. M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.
4. D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@ home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.
5. D. P. Anderson, "BOINC: A system for public-resource computing and storage," in *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004.*, pp. 4–10, IEEE, 2004.
6. E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *Communications Surveys & Tutorials, IEEE*, vol. 7, no. 2, pp. 72–93, 2005.
7. The Gnutella Protocol Specification. `http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`, june 2002.
8. I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, "Freenet: A Distributed Anonymous information storage and retrieval system," in *International Workshop on Designing Privacy Enhancing Technologies: Design Issues in Anonymity and Unobservability*, pp. 46–66, Springer-Verlag New York, Inc., 2001.
9. D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web," in *Proceedings of the 29th annual ACM symposium on Theory of computing*, pp. 654–663, ACM, 1997.
10. I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.
11. C. G. Plaxton, R. Rajaraman, and A. W. Richa, "Accessing Nearby Copies of Replicated Objects in a Distributed Environment," in *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pp. 311–320, ACM, 1997.

12. B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in *Proceedings of the 23rd Annual Joint Conference of the IEEE Computer and Communications Societies INFOCOM 2004.*, vol. 4, pp. 2253–2262, IEEE, 2004.
13. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing," Tech. Rep. UCB/CSD-01-1141, EECS Department, University of California, Berkeley, Apr 2001.
14. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, *A scalable content-addressable network*, vol. 31. ACM, 2001.
15. P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," in *Peer-to-Peer Systems*, pp. 53–65, Springer, 2002.
16. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 205–220, ACM, 2007.
17. A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," in *ACM SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, ACM, 2010.
18. F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with CFS," in *ACM SIGOPS Operating Systems Review*, vol. 35, pp. 202–215, ACM, 2001.
19. P. Knezevic, A. Wombacher, and T. Risse, "Enabling high data availability in a DHT," in *Proceedings of the 16th International Workshop on Database and Expert Systems Applications, 2005.*, pp. 363–367, IEEE, 2005.
20. B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems," in *Proceedings of the 3rd conference on Networked Systems Design & Implementation-Volume 3*, pp. 4–4, USENIX Association, 2006.
21. K. Kyungbaek and P. Daeyeon, "Reducing replication overhead for data durability in DHT based P2P system," *IEICE transactions on information and systems*, vol. 90, no. 9, pp. 1452–1455, 2007.
22. J. Zhao, H. Yu, K. Zhang, W. Zheng, J. Wu, and J. Hu, "Achieving reliability through replication in a wide-area network DHT storage system," in *International Conference on Parallel Processing, 2007. ICPP 2007.*, pp. 29–29, IEEE, 2007.
23. S. Bessa, M. Correia, and P. Brandao, "Storage and retrieval on P2P networks: A DHT based protocol," in *Proceedings of the 12th IEEE Symposium on Computers and Communications, 2007. ISCC 2007.*, pp. 623–629, July 2007.
24. K. Kyungbaek and P. Daeyeon, "Reducing replication overhead for data durability in DHT based P2P system," *IEICE transactions on information and systems*, vol. 90, no. 9, pp. 1452–1455, 2007.
25. W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, 2008.
26. S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.
27. R. Klophaus, "Riak core: building distributed applications without shared state," in *ACM SIGPLAN Commercial Users of Functional Programming*, p. 14, ACM, 2010.

28. T. Schütt, F. Schintke, and A. Reinefeld, "Scalaris: reliable transactional P2P key/value store," in *Proceedings of the 7th ACM SIGPLAN workshop on ERLANG 2008*, (New York, NY, USA), pp. 41–48, ACM, 2008.
29. J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB: The Definitive Guide*. O'Reilly Media, Inc., 2010.
30. D. Borthakur, *The Hadoop Distributed File System: Architecture and Design*. The Apache Software Foundation, 2007.
31. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, (New York, NY, USA), pp. 143–154, ACM, 2010.
32. J. Katajainen, T. Pasanen, and J. Teuhola, "Practical in-place mergesort," *Nordic J. of Computing*, vol. 3, pp. 27–40, Mar. 1996.
33. Cassandra Ruby client library. `http://github.com/fauna/cassandra`, 2011.
34. The Apache Software Foundation, "CQL 3.0." http://cassandra.apache.org/doc/cql3/CQL.html.
35. P. Leach, M. Mealling, and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace." `http://tools.ietf.org/html/rfc4122`.
36. P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.
37. U. Lublin and D. G. Feitelson, "The workload on parallel supercomputers: modeling the characteristics of rigid jobs," *Journal of Parallel and Distributed Computing*, vol. 63, no. 11, pp. 1105–1122, 2003.
38. S. Toyoshima, S. Yamaguchi, and M. Oguchi, "Storage access optimization with virtual machine migration and basic performance analysis of Amazon EC2," in *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications Workshops*, pp. 905–910, IEEE, 2010.
39. R. Gupta, V. Sekhri, and A. K. Somani, "CompuP2P: An architecture for internet computing using peer-to-peer networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 11, pp. 1306–1320, 2006.
40. G. Chmaj and K. Walkowiak, "A P2P computing system for overlay networks," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 242–249, 2013.
41. A. Legout, G. Urvoy-keller, and P. Michiardi, "Understanding BitTorrent: An Experimental Perspective," in *Proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 2005*, pp. 961–969, ACM Press, 2005.
42. J.-C. Charr, R. Couturier, and D. Laiymani, "JACEP2P-V2: A fully decentralized and fault tolerant environment for executing parallel iterative asynchronous applications on volatile distributed architectures," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 606–613, 2011.
43. C. Lucchese, C. Mastroianni, S. Orlando, and D. Talia, "Mining@Home: Toward a public-resource computing framework for distributed data mining," *Concurrency and Computation: Practice and Experience*, vol. 22, pp. 658–682, Apr. 2010.
44. D. Castellà, J. Rius, I. Barri, F. Giné, and F. Solsona, "A new reliable proposal to manage dynamic resources in a computing P2P system," in *Proceedings of the 17th Euromicro International Conference on Parallel, Distributed and*

Network-based Processing. 2009 (D. E. Baz, F. Spies, and T. Gross, eds.), pp. 323–329, IEEE Computer Society, 2009.

45. V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao, “Cluster computing on the fly: P2P scheduling of idle cycles in the internet,” in *Peer-to-Peer Systems III*, pp. 227–236, Springer, 2005.
46. J. Cao, O. M. Kwong, X. Wang, and W. Cai, “A peer-to-peer approach to task scheduling in computation grid,” in *Grid and Cooperative Computing*, pp. 316–323, Springer, 2004.
47. R. Ranjan, A. Harwood, and R. Buyya, “Coordinated load management in Peer-to-Peer coupled federated grid systems,” *The Journal of Supercomputing*, pp. 1–25, 2010.
48. D. Milano and N. Stojnić, “Shepherd: node monitors for fault-tolerant distributed process execution in osiris,” in *Proceedings of the 5th International Workshop on Enhanced Web Service Technologies*, WEWST ’10, (New York, NY, USA), pp. 26–35, ACM, 2010.
49. M. Sánchez-Artigas and P. García-López, “eSciGrid: A P2P-based e-science Grid for scalable and efficient data sharing,” *Future Generation Computer Systems*, vol. 26, pp. 704–719, May 2010.
50. H. Zhang, H. Jin, and Q. Zhang, “Scheduling Strategy of P2P Based High performance computing platform base on session time prediction,” in *Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, GPC ’09, (Berlin, Heidelberg), pp. 364–375, Springer-Verlag, 2009.
51. J.-S. Kim, B. Nam, P. Keleher, M. Marsh, B. Bhattacharjee, and A. Sussman, “Trade-offs in matching jobs and balancing load for distributed desktop grids,” *Future Generation Computer Systems*, vol. 24, pp. 415–424, May 2008.
52. A. S. Cheema, M. Muhammad, and I. Gupta, “Peer-to-Peer Discovery of Computational resources for grid applications,” in *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID ’05, (Washington, DC, USA), pp. 179–185, IEEE Computer Society, 2005.
53. M. Cai, M. Frank, J. Chen, and P. Szekely, “MAAN: A multi-attribute addressable network for grid information services,” in *Proceedings of the 4th International Workshop on Grid Computing*, GRID ’03, (Washington, DC, USA), pp. 184–, IEEE Computer Society, 2003.
54. A. R. Bharambe, M. Agrawal, and S. Seshan, “Mercury: supporting scalable multi-attribute range queries,” in *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM ’04, (New York, NY, USA), pp. 353–366, ACM, 2004.
55. A. Andrzejak and Z. Xu, “Scalable, efficient range queries for grid information services,” in *Proceedings of the Second International Conference on Peer-to-Peer Computing*, P2P ’02, (Washington, DC, USA), pp. 33–40, IEEE Computer Society, 2002.
56. J. Albrecht, D. Oppenheimer, A. Vahdat, and D. A. Patterson, “Design and implementation trade-offs for wide-area resource discovery,” *ACM Transactions on Internet Technology*, vol. 8, pp. 18:1–18:44, October 2008.
57. L. Lamport, “The Part-time Parliament,” *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.
58. A. A. Markov, “Investigation of a noteworthy case of dependent trials,” *Izv Akad Nauk Ser Biol*, vol. 1, 1907.
59. S. I. Resnick, *Adventures in Stochastic Processes.* Springer, 1992.

60. R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, “Serving large-scale batch computed data with Project Voldemort,” in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, pp. 18–18, USENIX Association, 2012.
61. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” in *In proceedings of the 7th Conference on USENIX Symposium on operating systems design and implementation - Volume 7*, pp. 205–218, 2006.
62. P. Bailis, S. Venkataraman, M. Franklin, J. Hellerstein, and I. Stoica, “Quantifying eventual consistency with PBS,” *The Very Large Data Bases Journal*, vol. 23, no. 2, pp. 279–302, 2014.
63. R. van Renesse, D. Dumitriu, V. Gough, and C. Thomas, “Efficient reconciliation and flow control for anti-entropy protocols,” in *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS ’08, (New York, NY, USA), pp. 6:1–6:7, ACM, 2008.
64. A. D. Birrell, R. Levin, M. D. Schroeder, and R. M. Needham, “Grapevine: an exercise in distributed computing,” *Communications of the ACM*, vol. 25, pp. 260–274, Apr. 1982.
65. R. Karp, C. Schindelhauer, S. Shenker, and B. Vocking, “Randomized rumor spreading,” in *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS ’00, (Washington, DC, USA), pp. 565–, IEEE Computer Society, 2000.
66. N. Hayashibara, X. Defago, R. Yared, and T. Katayama, “The φ accrual failure detector,” in *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pp. 66–78, IEEE, 2004.
67. W. Kuo and Z. Ming, *Optimal Reliability Modeling: Principles and Applications.* Wiley, 2002.
68. A. A. Hagin, “Reliability evaluation of a repairable network with limited capacity and structural redundancy,” *Microelectronics Reliability*, vol. 37, no. 2, pp. 341 – 347, 1997.
69. P. Pulat, “Network reliability with arc failures and repairs,” *IEEE Transactions on Reliability*, vol. 37, pp. 268 –273, aug 1988.
70. H. Li and G. Chen, “Data persistence in structured P2P networks with redundancy schemes,” in *Proceedings of the 6th International Conference on Grid and Cooperative Computing*, GCC ’07, (Washington, DC, USA), pp. 542–549, IEEE Computer Society, 2007.
71. J. Tian, Z. Yang, and Y. Dai, “A data placement scheme with time-related model for p2p storages,” in *Proceedings of the IEEE International Conference on Peer-to-Peer Computing*, vol. 0, (Los Alamitos, CA, USA), pp. 151–158, IEEE Computer Society, 2007.
72. S. Ross, *Stochastic processes.* Wiley series in probability and mathematical statistics. Probability and mathematical statistics, Wiley, 1983.
73. D. Ford, F. Labelle, F. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, “Availability in globally distributed storage systems,” in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.
74. J. B. Dugan and K. S. Trivedi, “Coverage modeling for dependability analysis of fault-tolerant systems,” *IEEE Transactions on Computers*, vol. 38, pp. 775–787, June 1989.

75. S. Akhtar, “Reliability of k-out-of-n:G systems with imperfect fault-coverage,” *IEEE Transactions on Reliability*, vol. 43, pp. 101 –106, mar 1994.
76. M. Marzolla, “The `qnetworks` toolbox: A software package for queueing networks analysis,” in *Analytical and Stochastic Modeling Techniques and Applications, 17th International Conference, ASMTA 2010, Cardiff, UK, Proceedings* (K. Al-Begain, D. Fiems, and W. J. Knottenbelt, eds.), vol. 6148 of *Lecture Notes in Computer Science*, pp. 102–116, Springer, June14–16 2010.
77. R. A. Sahner, K. S. Trivedi, and A. Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the Sharpe Software Package.* Kluwer Academic Publishers, 1996.
78. K. S. Trivedi, M. Malhotra, and R. M. Fricks, “Markov reward approach to performability and reliability analysis,” in *Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1994. MASCOTS'94.*, pp. 7–11, IEEE, 1994.
79. B. Javadi, J. Abawajy, and R. Buyya, “Failure-aware resource provisioning for hybrid cloud infrastructure,” *Journal of Parallel and Distributed Computing*, vol. 72, pp. 1318–1331, Oct. 2012.
80. S. Genaud, E. Jeannot, and C. Rattanapoka, “Fault management in P2P-MPI,” in *In proceedings of International Conference on Grid and Pervasive Computing, GPC'07, Lecture Notes in Computer Science*, Springer, 2007.
81. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, vol. 1. MIT press, 1999.
82. D. Carra and E. W. Biersack, “Building a reliable P2P system out of unreliable P2P clients: the case of KAD,” in *Proceedings of the 3rd conference on emerging Networking EXperiments and Technologies (CoNEXT)*, (New York, NY, USA), pp. 1–12, ACM, 2007.
83. M. Steiner, T. En-Najjary, and E. W. Biersack, “A global view of KAD,” in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, (New York, NY, USA), pp. 117–122, ACM, 2007.
84. Y. Houri, B. Amann, and T. Fuhrmann, “A quantitative analysis of redundancy schemes for peer-to-peer storage systems,” in *Proceedings of the 12th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'10, (Berlin, Heidelberg), pp. 519–530, Springer-Verlag, 2010.
85. R. Rodrigues and B. Liskov, “High availability in DHTs: Erasure coding vs. replication,” in *Revised Selected Papers of Peer-to-Peer Systems IV: 4th International Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005*, vol. 4, p. 226, Springer Science & Business Media, 2005.
86. H. Weatherspoon and J. Kubiatowicz, “Erasure coding vs. replication: A quantitative comparison,” in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pp. 328–338.
87. G. Utard and A. Vernois, “Data durability in peer to peer storage systems,” in *Proceedings of the 2004 IEEE International Symposium on Cluster Computing and the Grid*, CCGRID '04, (Washington, DC, USA), pp. 90–97, IEEE Computer Society, 2004.
88. J. Li, G. Xu, and H. Zhang, “Performance comparison of erasure codes for different churn models in p2p storage systems,” in *Advanced Intelligent Computing Theories and Applications. With Aspects of Artificial Intelligence* (D.-S. Huang, X. Zhang, C. Reyes García, and L. Zhang, eds.), vol. 6216 of *Lecture Notes in Computer Science*, pp. 410–417, Springer Berlin / Heidelberg.

89. J. Tian and Y. Dai, "Understanding the dynamic of peer-to-peer systems," in *Proceedings of the 6th International Workshop on Peer-To-Peer Systems, IPTPS 2007*, 2007.
90. R. Bhagwan, S. Savage, and G. M. Voelker, "Understanding availability," in *Peer-to-Peer Systems II, Second International Workshop, IPTPS 2003, Berkeley, CA, USA, February 21-22,2003, Revised Papers*, pp. 256–267, Springer, 2003.
91. W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer, "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, pp. 34–43, ACM, 2000.
92. S. Saroiu, P. K. Gummadi, and S. D. Gribble, "Measurement study of peer-to-peer file sharing systems," in *Electronic Imaging 2002*, pp. 156–170, International Society for Optics and Photonics, 2001.
93. K. Kim and D. Park, "Reducing replication overhead for data durability in DHT based P2P system," *IEICE transactions on information and systems*, vol. E90-D, pp. 1452–1455, Sept. 2007.
94. F. M. Cuenca-Acuna, R. P. Martin, and T. D. Nguyen, "Autonomous replication for high availability in unstructured P2P systems," in *Proceedings of the Symposium on Reliable Distributed Systems (SRDS)*, 2003.
95. Z. Yao, D. Leonard, X. Wang, and D. Loguinov, "Modeling heterogeneous user churn and local resilience of unstructured p2p networks," in *Proceedings of the 2006 14th IEEE International Conference on Network Protocols, 2006. ICNP'06.*, pp. 32–41, IEEE, 2006.
96. W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky, "Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics," *Transactions on Storage*, vol. 4, pp. 7:1–7:25, Nov. 2008.
97. B. Schroeder and G. A. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?," in *Proceedings of the 5th USENIX conference on File and Storage Technologies*, FAST '07, (Berkeley, CA, USA), USENIX Association, 2007.
98. C. Anglano, J. Brevik, M. Canonico, D. Nurmi, and R. Wolski, "Fault-aware scheduling for Bag-of-Tasks applications on desktop grids," in *Proceedings of the 7th IEEE/ACM International Conference on Grid Computing*, pp. 56–63, IEEE, 2006.
99. M. Amoon, "A fault-tolerant scheduling system for computational grids," *Computers & Electrical Engineering*, vol. 38, no. 2, pp. 399 – 412, 2012.
100. W. Cirne, D. Paranhos, L. Costa, E. Santos-Neto, F. Brasileiro, J. Sauve, F. A. B. Silva, C. Barros, and C. Silveira, "Running Bag-of-Tasks applications on computational grids: the MyGrid approach," in *Proceedings of the 2003 International Conference on Parallel Processing, 2003.*, pp. 407–416, Oct 2003.
101. C. Anglano and M. Canonico, "Fault-tolerant scheduling for Bag-of-Tasks grid applications," in *Advances in Grid Computing - EGC 2005, European Grid Conference, Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers* (P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, eds.), vol. 3470 of *Lecture Notes in Computer Science*, pp. 630–639, Springer, 2005.
102. J. Brevik, D. Nurmi, and R. Wolski, "Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems," in *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid, 2004. CCGrid 2004.*, pp. 190–199, April 2004.

103. M. M. Khan, J. Navaridas, A. Rast, X. Jin, L. Plana, M. Lujan, J. V. Woods, J. Miguel-Alonso, and S. Furber, "Event-driven configuration of a neural network cmp system over a homogeneous interconnect fabric," in *Proceedings of the 8th International Symposium on Parallel and Distributed Computing, 2009. ISPDC '09.*, pp. 54–61, June 2009.
104. R. Brown, "Calendar queues: A fast 0(1) priority queue implementation for the simulation event set problem," *Communications of the ACM*, vol. 31, pp. 1220–1227, Oct. 1988.
105. W. Tang, N. Desai, D. Buettner, and Z. Lan, "Analyzing and adjusting user runtime estimates to improve job scheduling on the blue gene/p," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS), 2010*, pp. 1–11, IEEE, 2010.
106. B. Javadi, J. Abawajy, and R. Buyya, "Failure-aware resource provisioning for hybrid cloud infrastructure," *Journal of Parallel and Distributed Computing*, vol. 72, pp. 1318–1331, Oct. 2012.
107. J. Bansal, S. Rani, and P. Singh, "The WorkQueue with dynamic replication-fault tolerant scheduler in desktop grid environment," *International Journal of Computers & Technology*, vol. 11, no. 4, pp. 2446–2451, 2013.
108. A. J. Oliner, R. K. Sahoo, J. E. Moreira, M. Gupta, and A. Sivasubramaniam, "Fault-aware job scheduling for bluegene/l systems," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 64, IEEE, 2004.
109. Y. Li, Z. Lan, P. Gujrati, and X.-H. Sun, "Fault-aware runtime strategies for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 4, pp. 460–473, 2009.
110. E. Byun, S. Choi, M. Baik, J.-M. Gil, C. Y. Park, and C.-S. Hwang, "MJSA: Markov job scheduler based on availability in desktop grid computing environment.," *Future Generation Computer Systems*, vol. 23, no. 4, pp. 616–622, 2007.
111. K. Ramachandran, H. Lutfiyya, and M. Perry, "Decentralized approach to resource availability prediction using group availability in a P2P desktop grid.," *Future Generation Computer Systems*, vol. 28, no. 6, pp. 854–860, 2012.
112. H. Xiaoping, W. Zhijiang, W. Congming, W. yu, C. Yongshang, and S. Ling, "Availability-based task monitoring and adaptation mechanism in desktop grid system," in *Proceedings of the Sixth International Conference on Grid and Cooperative Computing, 2007. GCC 2007.*, pp. 444–450, Aug 2007.
113. J.-H. Hyun, "An effective scheduling method for more reliable execution on desktop grids," in *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications (HPCC), 2010*, pp. 172–179, IEEE, 2010.
114. H. Li, D. Groep, and L. Wolters, "Workload characteristics of a multi-cluster supercomputer," in *Job Scheduling Strategies for Parallel Processing* (D. Feitelson, L. Rudolph, and U. Schwiegelshohn, eds.), vol. 3277 of *Lecture Notes in Computer Science*, pp. 176–193, Springer Berlin Heidelberg, 2005.
115. T. White, *Hadoop: the definitive guide: the definitive guide.* " O'Reilly Media, Inc.", 2009.
116. M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pp. 10–10, 2010.