PC-PeerSim: Simulação Paralelizada de Overlays Peer-to-Peer em Clusters

[Extended Abstract]

João Manuel Parreira Neto
Instituto Superior Técnico

ABSTRACT

This article resumes the development of a solution for simulating peer-to-peer overlays in the Peersim system.
This project and its motivation in the imposed limits of the current version of peersim on not being able to take advantage of multi core processor and is severly limited by the available physical memory of the machine it's running in when defining the size of the simulation.
This document is comprised of an introduction to the concepts of grid computing and their uses and benefits, to the concept of the Terracotta distributed virtual machine and finally to the concept of a peer-to-peer overlay. It contains also a description of several current projects from the mentioned areas of expertise, filed and ranked according to evaluation criteria.
Here comprised is also a detailed description of the architecture and implementation of the developed solution, beginning with the changes made so that the simulator may maintain coherence and synchronization when running in the multithreaded mode, explaining the new protocol API and the memory sharing strategies used in the distributed execution. Finally, this dissertation ends with a detailed description of the architecture and implementation of the developed solution and and evaluation of the obtained results that prove the validity and usefulness of this project.

Keywords

Terracotta, Peer-to-peer, Grid, Simulation, Multithreading, Overlay, Distributed Computing

1. INTRODUCTION

The objective of this work consists in developing a system that allows and application designed to be based in a peer-to-peer overlay can be executing in a simulated environment and additionally that this simulation is able to run in a parallel execution environment (as a multicore/multicpu computer) or in a multi machine cluster (as opposed to single machine sequential execution).

1.1 Grid Computing

A Grid system is defined as a infrastructure that permits and facilitates the rapid sharing of computational resources ina a large scale so that they can be available to distributed computing applications. This infrastructure must encompass the necessary tools to manage the shared data between the various participants, all scheduling and planingm and all the requests of operations over the same shared data [8].
All work must be coordinated by an orchestrating entity that coordinates all activity in the grid.

1.2 Peer-to-peer

A peer-to-peer network can be defined as a network infrastructure that allows an easy way to distributed data or resources to a group of various machines (nodes or peers) associated together in a common context. This infrastructure does note require a mandatory central organization unit or server.
The peer-to-peer networks are nowadays associated to the online file sharing movement, proving this way the viability of a distributed system with a enfasys on the efficient data search and transmission, but there are also other applications for this tecnology, being the example of this distributed storage software and chat applications.
A peer-to-peer network allows in this way the construction of an organized system for data sharing and transmission (or other resources of computational interest) between various heterogeneous machines (nodes) without the necessity of a central control, in which all nodes can actively participate, these being able to resist to failures in their peers and maintain connectivity without the necessity of specialized intervention by a coordinating agent.

1.3 Terracotta

Terracotta appears as an essential tool for the construction of the distributed layer of the application. With the emergence of these Grid and Peer-to-Peer technologies and the prominence of Java as the leading free and open source programming language (that is also implemented in a variety of th operative systems and hardware), some applications began to emerge in the last years a systems being able to run Java applications in a distributed fashion over a cluster of various machines. Terracotta is defined by a distributed Java virtual machine based on a client/server architecture in which is possible to execute programs, originally coded to single machine execution, in a cluster, without the need to alter its implementation or add any modules.

1.4 Peersim

A peer-to-peer simulator consists in a system able to simulate a peer-to-peer overlay with an arbitrary number of nodes, and arbitrary protocols and node operations, in a single machine. This software allows the emulation of algorithm and code structure behavior in a computer network, without having actually to install physical machines in a physical network. Peersim is a peer-to-peer overlay simulator written in Java that supplies some protocols and easily allows it's enhancement with newer or custom made ones. Besides the greater apparent scalability it offers (compared with other Java based simulators) with it's ability to simulate with reasonable efficiency an overlay composed of a million (10^6) simulated nodes [15][16].

2. RELATED WORK

2.1 Peer-to-Peer Network

The various different peer-to-peer network protocols may be classified according to it's degree of centralization and being structured (or not).

A structured peer-to-peer network maintains, during it's execution, a structure similar to a distributed Hash Table where it keeps information about all the participants in the network, so that it can maintain their location int the network always available, and also all the keys for quick search times in the overlay.

A key represents something for which a node is responsible (in a data-sharing application a key bay represent a file or a data block, in a DNS server it would represent a name/address pair, in a library it would represent an article, etc.). The way this structure is implemented depends on the protocol itself, as the communication and search mechanics vary in each one of them, although normally a specific node in the overlay will maintain a limited number of other nodes (it's neighbors). The overlay assigns an unique identifier to each node and to each key. This allows the construction of a virtual graph of the overlay in which we can easily identify the node responsible for each key.

The balance and maintenance of a structured network has a high degree of complexity, making it difficult and inefficient to keep the overlay coherent when there's a high volume of nodes entering and exiting the network with it's set of keys. This makes unstructured and centralized overlays as the popular implementations of choice when building data sharing networks.

The centralization degree of a peer-to-peer overlay is defined by the need of a central semi-coordination in the network. Semi-coordination because if we build a network in which a centralized server does all the work, it can no longer be called a peer-to-peer network, as it violates its main goal of work sharing.

A peer-to-peer overlay may be:

- **Completely decentralized.** All the nodes have the same statute in the network. This alternative has the advantage of not maintaining a complete control over the total state of the overlay.
- **Partially decentralized or hybrid**. Some of the nodes in the network are called super-peers, and together they coordinate all entries and exits of the network, although each node by itself only has knowledge about a subset of the entire network.
- **Completely centralized.** The network has the need of a central server that controls all the entries and exits of the network and it may or may not have knowledge of all the objects or data blocks stored in the network. The obvious drawback is the central server being a major fail point in the entire network, as all the work depends on it's availability. [14]

2.1.1 Structured Networks

The prime examples of Structured networks would be **Chord**[20], **Kademlia**[13] and **CAN**[**?**].

Chord is a peer-to-peer overlay where the nodes are disposed in a virtual ring in which each node only knows about a small subset of nodes located all around the ring. The nodes are responsible to all keys whose identifier is less than it's own identifier and bigger than the identifier of the predecessor. When a node needs a key, it sends a message to the closest node to the and the request is then relayed forward if needed.

Tapestry makes use of a technique similar to Plaxton ([18]) to locate a key from a certain node. Each node keeps a address table of it's neighbors with a number of levels equal to the number of digits of the highest possible identifier. Each level stores x entries (x being the the base of the addresses. $x = 10$ for decimal, $x = 2$ for binary, etc.). Inspired by Tapestry, comes the Pastry overlay [19] in which the table is a more complex structure using the number of bits required to store the addresses and using address prefix based searches.

CAN makes use of a pseudo geographical organization of the nodes, meaning the nodes are disposed in areas mapped by a cartesian structure in which each node is defined by it's ID and coordinates in a virtual space (with N dimensions). The neighboring is defined by adjacent areas in the virtual space.

Other interesting examples of structures networks would be **Kademlia**[13], based in a node distance metric, **Viceroy**[12] with it's multi level ring based network, **Koorde**[10] based on Chord but in which each node has various neighbors and **Cyclone**[4] also based on Chord with multiple rings.

Structured protocols where not designed with a single specific purpose but from the need of building a data sharing and trasmission system, completely decentralized a capable of support a high degree of node traffic without interfering in the lookup process.

In the cited examples, the lookup times are similar, being in the majority defined by $O(logn)$ or slight variations of it with the exception of Koorde, which uses a De Bruijn graph for speedup, and CAN that provides a lookup time of $O((d * N) * (1/d))$, d being the number of dimensions of the virtual space.

2.1.2 Unstructured Networks

The first and most famous example of an unstructured network would be **Napster**[3]. Napster was one of the earliest file sharing network, alongside **Gnutella**[3] and **Kazaa**[9].

Napster used a central server that indexed all files in the system alongside it's location. Highly centralized, Napster would hardly be a good example of a peer-to-peer network nowadays although it's was the first example of a public distributed file storage in a very large scale.

Reducing the centralization degree, comes **Kazaa** and it's

definition of super-peers. Not having a completely central server, each super-peer indexes a subset of the nodes and it's keys. A search is propagated from a node to it's super-peers and the rest of the super-peers making the search much more efficient than in **Gnutella** that provides a completely decentralized system where the search is propagated in flood through all the neighbors.
Some good examples of unstructured peer-to-peer networks with other objectives than file sharing we have **Freenet**[6] and **FreeHaven**[7] offering services of anonymous data publication.
Although centralization offers a good performance and ease of data and location indexation, big problems come over the high dependence of the network in it's super-peers or central servers.

2.2 Grid Systems

The concept of Grid System encompasses a vast lot of very different projects with radically different methods and objectives, final purpose or even the resource sharing process. On the subject of Desktop and Cooperative Grids, the prime examples would be **SETI@Home**[2] and **Folding@Home**[11]. A Cooperative Grid is an system open to whoever is interested to offer it's resources to a common goal, in an approach similar to a peer-to-peer network. Both these examples run as Desktop Grids, which means they usually run in a background process so that the user can share only part of the available resources (being able to run only in idle time for example).
SETI@Home and **Folding@Home** are both scientific projects in which the goal to attain would require a computational cost too high to be viable. The Grid project consists in the processing of data coming from the radiotelescope in Arecibo, Puerto Rico. The goal of the project is to collect all kinds of radio signals from outer space in search of an artificial source that may come from an extra-terrestrial civilization. Using **SETI@Home** each user runs an application that fetches data blocks from the central server, processes the same data and returns it to the server. In a similar fashion, the project **Folding@Home** consists in the processing of Protein Folding experiments for medical purposes, mainly cancer investigation.
Both these projects are built over the **BOINC**[1] framework. This toolkit was designed specifically so that a scientist or a researcher would be able do build a Grid system in the same fashion with little work.
Another good example of Grid systems would be the marketplace systems. The purpose of these systems is not to participate directly in a specific project but to be able simply to share resources so that other people may use them for their own purposes. The prime example of this application is the **Grid4All** project[21]. This project was part of an initiative in the European Union to create a commercial market of resource sharing where prices are regulated according to a market system of request and offer ratio.
These examples illustrate the multitude of final purposes for a Grid system, although always with a common goal of creating an environment where tasks with enourmous computational need may be accomplished.

2.3 Peer-to-Peer Simulators

Peer-to-Peer simulators is a very specific area so there are not many relevant projects to discuss. Besides **Peersim**, the main projects in network simulation are **GPS**[22], **P2Psim**, and **OverSim**[5].
GPS was peer-to-peer overlay simulator based in Java that's currently inactive. It was based in an event engine and was extensible with custom made protocols. Also almost abandoned is the **P2Psim** project. This project was also based in an event engine and also allowed the use of custom made protocols, unfortunately because of scarce documentation, published benchmarks showed some difficulty in running simulations with more than 3000 nodes[17].
OverSim is a very interesting project as it provides a very complete package with statics module and a graphical interface with an overlay visualizer. This software is more directed to the simulation of actual TCP/IP networks and it's dynamics.
At last, **Peersim**[17] is the leading Peer-to-Peer simulator with it's Java based simulator, providing both a cycle-driven and event-driven engine and high scalability and extensibility. Also, some of the most common peer-to-peer protocols are already available for peersim.

3. ARCHITECTURE

In order to attain the propose objective, the current version of Peersim would have to be changed t o abandoned the main restriction it presents today: it's purely sequential and single-threaded execution. Although Terracotta provides a solution out of the box to connect several Java virtual machines (VM) without being necessary to modify the code of the program, this would only serve Peersim as having several machines in communication but running independent simulations. The simulation would be faster neither would it have more memory at it's disposal.
The approach to the objectvie was made in two main phases::

- **Parallelization** - Modifying the peersim simulation engine so that it can be able to execute the simulation in several threads.
- **Distribution** - Extend the same simulation engine so that it can be able to use the shared memory platform Terracotta.

3.1 Parallelization

As it was said, in it's original version, Peersim provides two independent simulation engines. A cycle-driven engine and an event-driven engine. In it's cycle-drive version, Peersim runs a defined number of cycles. In each one, it cycles through the entire list of nodes forming the overlay, and one by one, activates all the protocols defined to run at each cycle. (Fig. 1)

In it's event-driven version, the simulator initializes an ordered queue with events representing node activation alongside simulation control events. This queue is consumed in an infinite loop, in which the events after activated are recycled to the end of the queue. When the simulation reaches the time limit (defined as a total of events to process), the recycling stops, and when the queue is empty, the simulation ends. (Fig. 2)

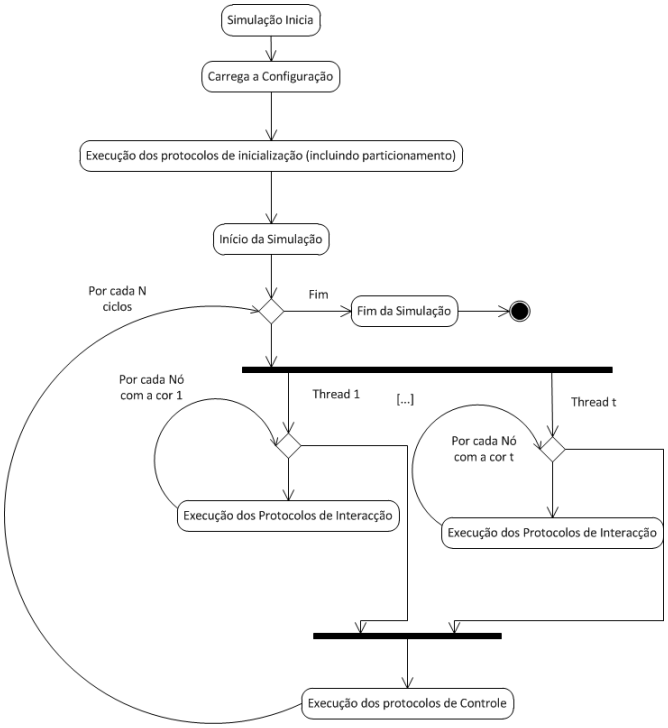

Figure 1: *Cycle-Driven* execution

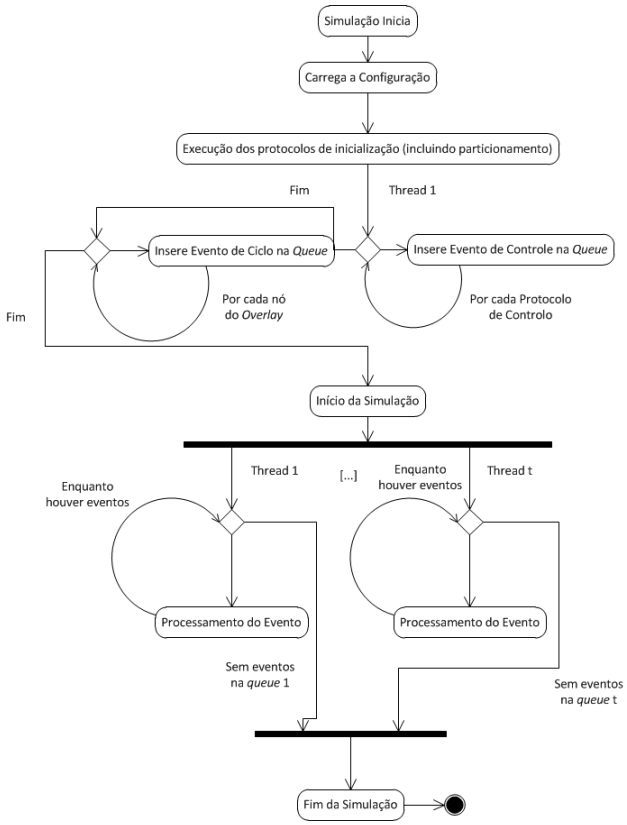

Figure 2: *Event-Driven* execution

For the simulation to be executed in various threads simultaneously, the entire set of nodes must be shared by them. So, in a simple approach, in both cases, the solution to the problem will be to have the various threads feeding of the node list or queue and synchronizing all activity made in each node.
This solution, although simple, lacks efficiency because of the need to sincronize all the code pertaining to node interaction. This inefficiency comes for the fact that sincronized code will always take more time to run than unsafe code. As all threads eventually feed from a global set of nodes, it's not possible to determine which thread will activate node x, so it's necessary to synchronize all interactions.
From this problem we form the hipothesys of assigning only a fixed set o the overlay to each thread permitting the simulator to synchronize only some interactions between concurrent threads.

3.1.1 Overlay partitioning

So that it is possible to define and know which nodes will be simulated by which thread, we added an overlay partitioning mechanism to peersim. This network partitioning, similar to graph coloring, is determined by an algorithm that assigns to each node a color that determines which thread will activate this node.
To maximize the efficiency of the coloring process, this should be made according to the neighboring relations between the nodes. This will effectively maximizer the interactions between nodes of the same color.

3.2 Distribution

After solving the parallelization problem, comes the problem associated to memory limits of the simulation. This problem is what effectively limits peersim of it's potential because it imposes a hard limit to the size of the overlay to simulate. Aditionally, the complexity of the protocols to use in the simulation, may augment the cost of memory occupation associated with each simulated node.
Terracotta and it's capability of easily sharing objects between various VMs, without the need to change the original code, comes as almost a miracle remedy to reach the objective, there are though some complications that need solution.
Although Terracotta allows the easy connection between VMs, when a participant loads an object, or a set of objects, this load is not shared through the rest of the participants, only the machine who loaded it will have it's memory occupied by the object. As the majority of the simulation load will be the node information itself, this task will have to be shared by all the participants so that it may be possible to increase the maximum simulation size.
When using the coloring process here, we can assign each participant with a specific color so that each participant may do only part of the work thus distributing the processing load by the entire cluster.

3.3 Architecture

Figure 3 represents the general architecture of PC-Peersim. There is described in a simplified fashion how the virtual overlay is simulated by a physical cluster. On top of the figure we have the peersim entry interface where the simulation parameters are defined. In the base we have the interac-

tions between nodes. The Terracotta middleware comes as a wrapper between the program and the Java virtual machine.

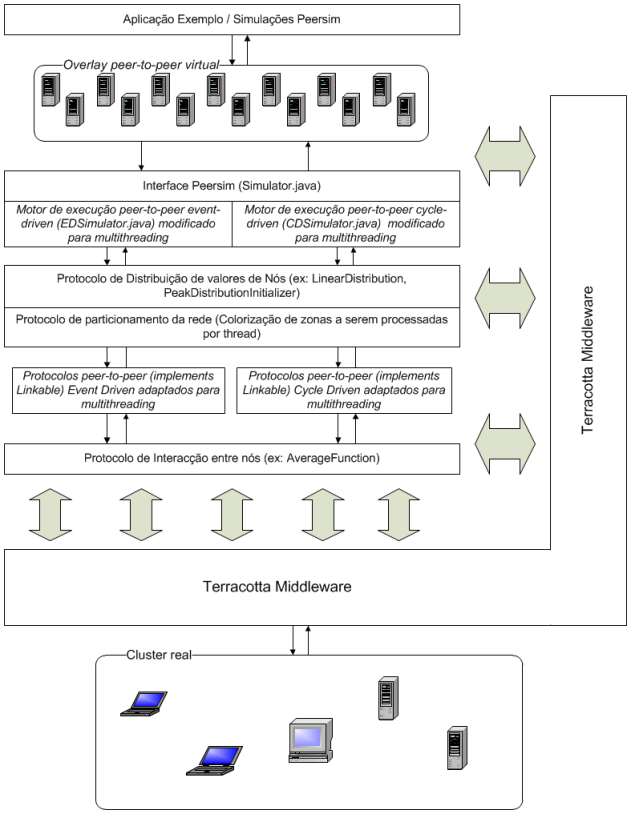

Figure 3: PC-Peersim Architecture

In closer detail:

- **Peersim Simulations** - The configuration of simulations remains unaltered compared to the original version. This configuration defines parameters like overlay size, number of cycles and which protocols will be executed, including initialization protocols (which define how the node values are inititialized and now, which partitioning algorithm to use)
- **Peersim Interface** - The main changes come now, as the configuration stage no makes the full or partial loading of the overlay. As referred, the partitioning is defined as initialization protocol so it runs in the initial stage of the simulation. To support these changes, the code structures that represent nodes and the overlay now have to maintain information about node coloring.
- **Execution Engine** - The execution engine is the component that effectively runs through all the nodes in each cycle (or event consumption) and executes the configured operations. According to the engine and mode of execution, it will do different things:
  - **Original mode** - The cycle driven engine will work with the whole overlay and will runt all cycles to the end. The event-driven engine works with a global queue of events configured with all the nodes in the overlay until a time limit is attained.
  - **Distributed mode** - The cycle driven engine will work over a partition of the overlay and remains blocked until all participants end the same cycle. The event-driven engine is not available in the distributed mode.
  - **Multithreaded mode** - The cycle driven engine launches a thread each cyle and each thread works with a partition of the overlay associated with it's color. The event-driven engine launches a thread for each color also, and each of them works with and independent queue. Each thread waits after x events so that none of them becomes too much ahead.
- **Node Distribution Protocol** - This protocol defines how the node values are initialized. When storing values in a node, it's impractical to define specific values to each node, so algorithms are used to do this.
- **Partitioning protocol** - This protocol contains the necessary implementation to make the correct and effective color distribution through the network.
- **Peer-to-Peer Protocols** - This protocol defines the actual peer-to-peer overlay in use, with the search algorithms and neighbor relations.
- **Interaction protocols** - This last component defines all executions to be made for each node.

4. IMPLEMENTATION

4.1 Partitioning

Several approaches where made to partitioning algorithms so that the simulator produces an overlay with cohesive colored areas. Let's assume as an example we have an overlay with 37 nodes and 62 neighbor connections and we are going to make a simulation with 4 threads/colors. In the ideal case, after the partitioning, the overlay would be divided in 4 areas (Fig. 4). In this example we would have a high connection ratio between nodes of the same color (73.4 %, 13 inter partition connections). In a normal situation, the partitioning would be more fragmented as in Figure 5, lowering the cohesion ration to 55.4

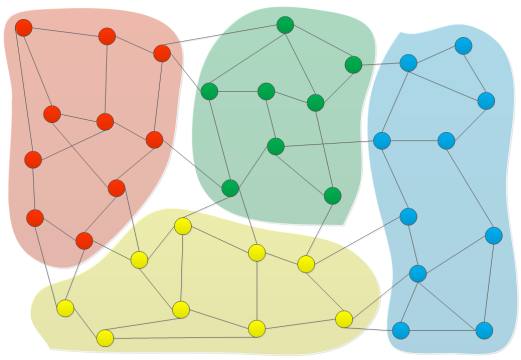

Figure 4: Ideal Partitioning

In the end, we reached two versions of the algorithm with good results:

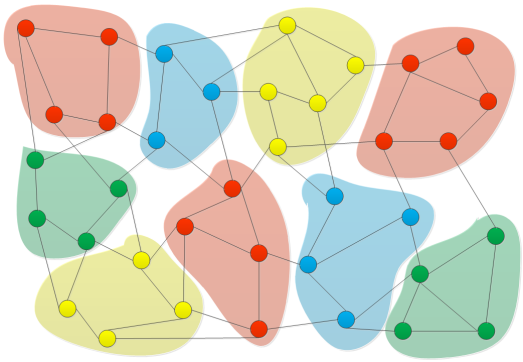

Figure 5: Fragmented Partitioning

- Breadth First Single Threaded
- Random

4.1.1 *Breadth First Single Threaded*

This algorithm tries to create the biggest possible area of a color then passes to the next color when it reaches a number o nodes equal to the size of the overlay divided by the number of colors. The execution is based around a list of nodes being fed with uncolored nodes. Each time a node is consumed it is colored, and it's neighbors are inserted at the head of the list.
If after cycling all colors, there are still uncolored nodes, this situation is corrected by adding the color corresponding to the one in majority in its neighbors.
In an overlay configure with a number of neighbors equal to $\log_{10} N$ being N it's size, this algorithm reaches a cohesion rate approaching 65

4.1.2 *Random*

A hypothesis that runs almost instantaneously is to assign to each node a random color from the set of available colors. This disregards completely the neighbor relations but has no impact in the total simulation time.
In an overlay configure with a number of neighbors equal to $\log_{10} N$ being N it's size, this algorithm reaches a cohesion rate approaching 50

4.2 Multithreaded overlay processing

4.2.1 *Cycle-driven Engine*

Instead of processing the entire overlay each cycle, the simulator now launches new threads each cycle that feed of a new structure representing the overlay. Instead of storing the overlays in a simple array, Peersim now manages a set of lists for each region, stored in a hash map.(Fig. 6)

This solution of launching new threads at each cycle has an associated cost so this is only effective when the execution of the interaction protocols represents the majority of the cpu load of the simulation.
In able to maintain the functionality of dynamically adding and removing nodes during the simulation without a considerable performance drop it was necessary to use unordered lists (Set) to represent the nodes. In doing this PC-Peersim

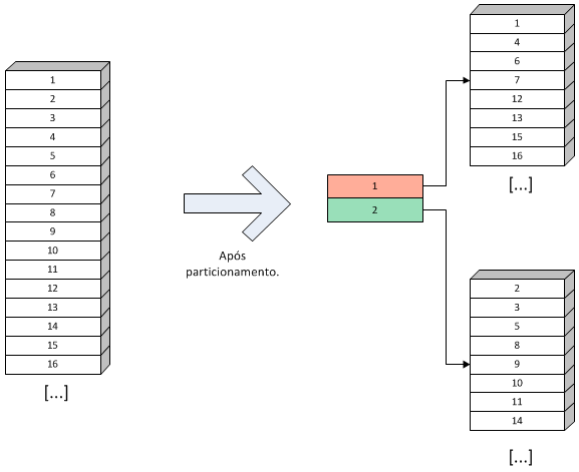

Figure 6: Overlay separated by color

lost the ability to cycle through the nodes in a specific order, only in a pseudo random fashion.

4.2.2 *Event-driven Engine*

In order to change this engine to a multithreaded version it was necessary to change que event queue to a structure containing mini-queues, where for each color. The control events are stored in one of the threads only so that they are executed only once. In this fashion, the engine may have the multiple threads running from the beginning of the simulation consuming their assigned events. So that no thread would become too much ahead of the others, after simulating the interactions between a number of nodes the thread stops at a cyclic barrier waiting for the other ones. The Figure 7 we can observe how the control events are only consumed by on thread only, and each thread keeps cycling the node events.

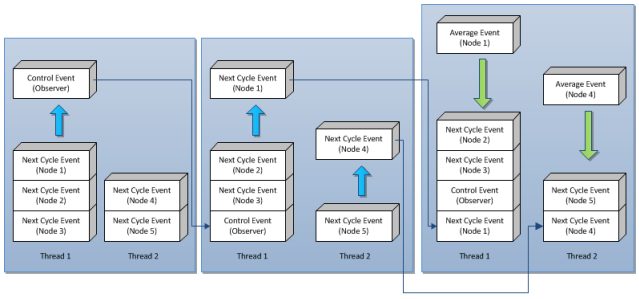

Figure 7: *Queue* modified to support *multithreading*

4.3 Node activity synchronization

As Peersim uses independent protocols to make the interactions between nodes, we could not simply alter the same protocols to effect synchronization because when other protcols are to be used or adapted, they would simply not work in PC-Peersim. To circumvent this, a strong API was defined as the new structure to use in the protocols. It defines new mandatory methods where to place specific node interaction code and defined new final methods that universally decide if they need to synchronize the execution and then invoke the interaction method.

With these minimal changes, it is possible to convert a protocol made for Peersim to use in PC-Peersim with the correct synchronization.

4.4 Distributed Execution

When running the simulation in the distributed mode (over terracotta), each participant will only be responsible for onde color so that the work may be simplified without sacrificing the main goals. For the same reason, only the cycle-driven engine was implemented over Terracotta.
The first participant will be the coordinator of the simulator, performing all initializations and coordinating the shared load. To be able to synchronize this execution and coordinate the shared load, terracota must share the node and overlay representing structures and also a set of locking objects and barriers.
To perform the object sharing Terracotta provides an XML based configuration engine that allows to define shared objects called roots. Any change made to an object reachable by a root object is propagated to all participants. It is not possible to change the root itself. Also, in order to protect the access to these objects, Terracotta requires mandatory synchronization in all possible access points to them. To be possible to do this without being necessary to change the code, it is also possible to configure automatic synchronization for all methods matching a regular expression, in the same configuration, as expressed in this example:

```
<named-lock>
   <method-expression>
        * example.aggregation.AverageFunction.*(..)
   </method-expression>
   <lock-level>write</lock-level>
   <lock-name>activateNodeInteraction</lock-name>
</named-lock>
```

5. EVALUATION

The tests presented here were made using two different configurations for simulation time extension for a configured overlay. In both engines, tests were made using the Peersim base protocols, more specifically based in the example simulation number 2 included in the original Peersim distribution. This example uses the internal protocol WireKOut and and interaction protocol name AverageFuncion that maintains a numeric value fore each node and for each interaction it calculates the average between the values in both nodes and assigns the new value to the nodes.
Besides these examples tests were made with a custom Chord and Pastry implementation and finally with the same example number 2 but running in the distributed mode over Terracotta. The distributed example was made both using two participants in a single machine and using the same two participants in two separate machines over a 100 Mbits/s LAN.
The test results are defined by the simulation time.

5.1 Event-driven results, Example 2

The event-driven engine revealed an interesting performance gain. Contrary to the cycle-driven version, this engine maintains the configured number of threads in permanent execution, feeding of the event set available in the queues. This fact permits the simuation to be able to be equally efficient in

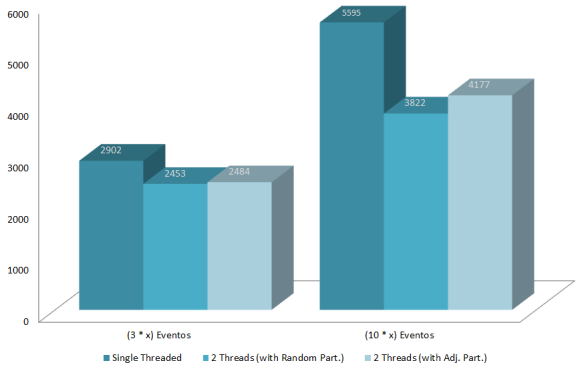

Figure 8: Execution time comparison for the Simulation Example 2 in the *Event-Driven* engine for 250.000 nodes

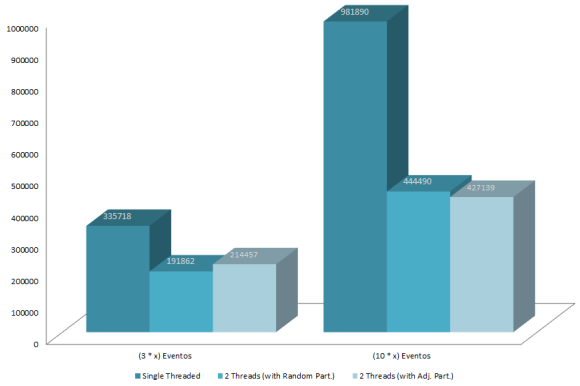

Figure 9: Execution time comparison for the Simulation Example 2 in the *Event-Driven* engine for 2.500.000 nodes

overlays of lower(Fig. 8) and great dimension (Fig. 9). The higher the computational cost for each cycle and protocol, the higher will be the possibility of the threads occupying 100% of CPU time. As for partitioning, the Random version revealed itself more efficient in the majority of cases, although the adjacency based version may be effiecient in big simulations.

5.2 Event-driven results, Chord Example

In the tests made with the Chord simulation example, the results were very interesting, as we can see an excellent performance gain in Fig. (10). One should notice that both this simulation and Pastry simulation carry a big computational cost when executing the interaction protocols, fairly larger than the simple average calculation in example 2. This factor makes the simulator spend more CPU time executing the protocols themselves instead of synchronization and event rotation.

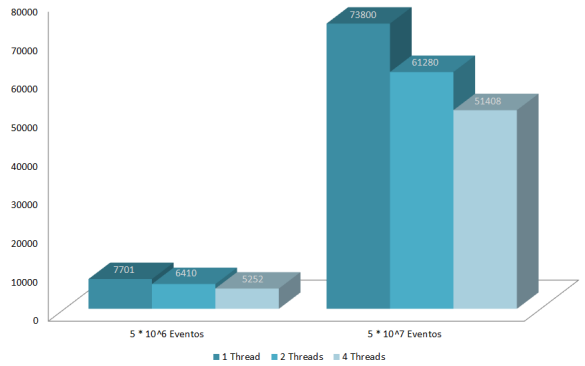

Figure 10: Execution time comparison for the Chord Example in the *Event-Driven* engine for 250.000 nodes

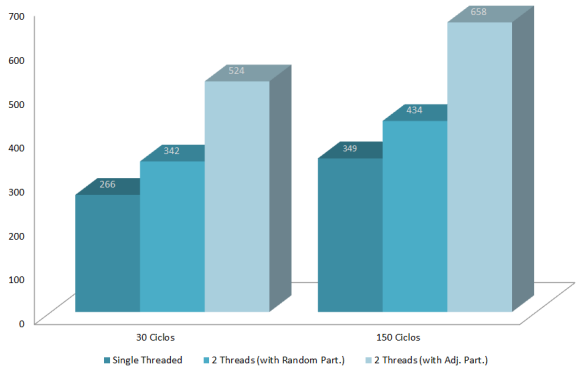

Figure 12: Execution time comparison for the Simulation Example 2 in the *Cycle-Driven* engine for 250.000 nodes

5.3 Event-driven results, Pastry Example

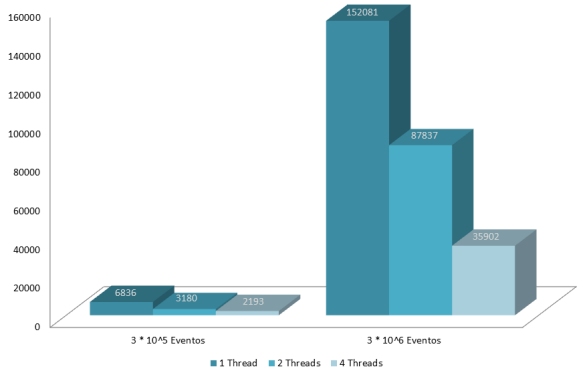

Figure 11: Execution time comparison for the Pastry Example in the *Event-Driven* engine for 5.000 nodes

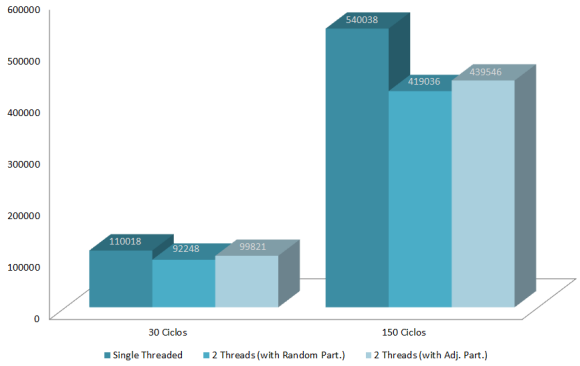

Figure 13: Execution time comparison for the Simulation Example 2 in the *Cycle-Driven* engine for 2.500.000 nodes

Similarly to the tests made with the Chord simulation, the same performance gain was observed in the Pastry simulation (11). Observing the time needed to perform the simulation over an overlay of 5000 nodes we can conclude the high cpu requirements of this protocol.

5.4 Cycle-driven results, Example 2

The cycle driven engine revealed itself less efficient than expected. The implementation of this engine passed through two versions during it's development. Originally, the engine would start a fixed number of threads as in the event-driven version. These threads would execute the configured cycles, and synchronized themselves at the end of each cycle. At the end, we found that it would be more efficient to start these same threads each cycle eliminating the need to synchronize at the end of each cycle. Someway, the cost to launch a new Thread by the VM was smaller than the inefficiencies shower by the required synchronization. This gain is only relevant when simulating bigger overlays. As we can see in Fig. 12 there is a drop in performance largely overshadowed by the gain in performance when running the bigger cycle necessary to simulate a larger overlay (Fig. 13).

5.5 Distributed simulation results

Contrary to the simulation running in a single machine, the distributed version over Terracotta revealed itself extremely inefficient and slow. In the almost the same time required to run 30 cycles in an overlay with 2.500.000 nodes in a single machine we can run 60 cycles in an overlay with 50 nodes in a distributed version.
This huge overhead associated with the use of Terracotta may be explained with several factors:

- Peersim makes use of an object array to represent the set of nodes of the overlay. This array is one of the objects configured as root in Terracotta and we may conjecture that Terracotta isn't able to share in an efficient

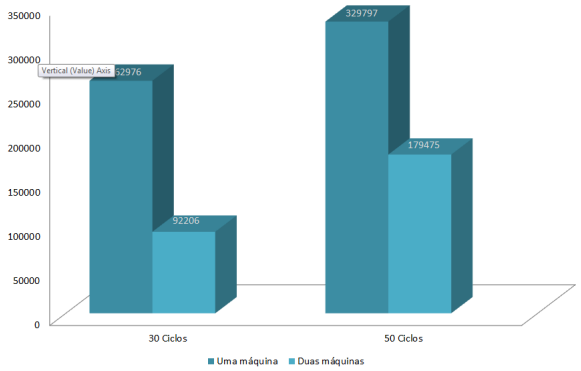

Figure 14: Execution time comparison for the Simulation Example 2 in the *Cycle-Driven* engine running over Terracotta for 500 nodes

way an array of objects without having to replicate it to each participant maintaining total synchronization over the same in all participants. This synchronization is maintained by the Terracotta server which apparently also maintains the objects in cache.
This theory is supported by the fact that the processes of all participants and of the Terracotta coordinator present a bigger memory footprint than a simulation of the same size running directly in a single VM. Although we cannot conclude imediately that this memory is associated with the same objects, the conclusions obtained by other Terracotta users in sharing the same type of objects points in that direction.

- Since the nature of the simulation is to promote interaction between various nodes that may "belong" to different participants we can conclude that all data and object exchanges between participants over the TCP/IP layer will be a lot slower than the direct accesses in the original simulation.

5.6 Final Analysis

The first conclusion to take from these performance tests is that there was effectively a strong success in accomplishing the objectives of this thesis. With the exception of the distributed simulation, all tests showe an improvement in the execution time necessary to complete the experiments.
The cycle-driven engine revealed itself less efficient than expected although it showed some improvement for bigger simulations. The event-driven engine showe a bigger performance gain mainly when using protocols with larger cpu requirements. The distributed simulation emerged as much inneficient in terms of both memory footprint and execution time and it served only to prove the ease of sharing objects between VMs using Terracotta and which problems one should avoid in a future development of this distributed version.

6. CONCLUSIONS

After analysing the results, we can safely say that the amin objectives were achieved with great success. More precisely we are able to affirm that it is possible to perform and obtain simulation results in a more efficient and fast way when using PC-Peersim. In almost all examples, there was very satisfactory results in terms of execution times:

- The cycle-driven engine showed an improvement to around 80% of the original execution time. This improvement however is only significant using a large number of nodes in the simulated overlay. This small gain in performance should not be overlooked as this version of the engine is the least flexible of the two, being this an interesting result overall. The neighboring partitioning algorithm was an underachievement, particularly using a small amount of nodes in the overlay and a small number of cycles, because it consistently added a delay to the overall simulation time making the test run longer than using the random partitioner.

- The event-driven engine presents a much more interesting improvement, around 50% of the original performance. Here lies the true victory of this project as this version of the engine is the most widely used for the creation of protocols (custom made and simulations of widely known overlays like Chord, Pastry, etc) by the research community in the field of peer-to-peer. The tests made using Chord and Pastry presented excellente results, with an additional batch of results being made on a quad-core machine. This batch of tests continued to show a good performance gain faced with the results already taken using two threads. As when using these examples, the simulation is effectively running protocols with a higher degree of computational charge, we can safely say that the development of the PC-Peersim is definitely an asset to the future.

  In it's distributed version, the PC-Peersim integrated in Terracotta presented very poor results. The memory overhead that Terracotta adds to the normal execution of the program added to the necessity of having to run an external coordinator server and the problems in Terracotta being able efficiently share some tipe of objects, the tests revealed that, without some deeper changes to the implementation, PC-Peersim will use as much memory for each participant as it is necessary for a single machine to simulate an overlay 10 times bigger. As the nature of Peersim requires a high load of information exchange (mainly Node objects), where will always be a big bottleneck in performance as the participants must always be exchanging objects with others, making the initial memory load sharing quite useless and ineffective. Besides the high memory footprint, this simulation had a big problem of execution speed. The objective of simulation very large overlays that would be impossible to load in a single machine was, therefore, not achieved. This exercise was quite useful though in providing the insight to which problems will erupt when using Terracotta, and to formulate possibilities of future work to achieve this same objective.

6.1 Future work

After completing this thesis and the corresponding result and objective analysis, we can conclude that there are big

perspectives of future work to attaining a version of Peersim that can truly be distributed and efficient. The use of Terracotta reveals itself as a wrong approach to the problem, as we linger with the idea that it would be possible to reach the same objective in a much simpler way, just by altering it's implementation. Using a lightweight messaging system it's quite possible to develop a version of Peersim capable of simulate large scale overlays with a good memory efficiency. After analyzing the results we also find that there's the possibility of improvement in the cycle-driven engine by reimplementing some of it's key features, although the long term usefulness of this engine could be questioned.
In the end we consider that there is a lot of potential for the development of a big part of all peer-to-peer protocols that where analyzed in the related work section. The versions of Chord and Pastry included where very useful when analyzing the results, so it would be very interesting to build a bigger and stronger code and knowledge base in this area.

7. ACKNOWLEDGMENTS

I owe my thanks to Prof. Luís Veiga and Prof. João Garcia as they where instrumental to the success of this thesis.

8. REFERENCES

[1] D. P. Anderson. BOINC: a system for Public-Resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10. IEEE Computer Society, 2004.

[2] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[3] S. Androutsellis-theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies, 2004.

[4] M. S. Artigas, P. G. L?pez, J. P. Ahull?, and A. F. G. Skarmeta. Cyclone: A novel design schema for hierarchical DHTs. In *Peer-to-Peer Computing, IEEE International Conference on*, volume 0, pages 49–56, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[5] I. Baumgart, B. Heep, and S. Krause. OverSim: a flexible overlay network simulation framework. In *2007 IEEE Global Internet Symposium*, pages 79–84, Anchorage, AK, USA, 2007.

[6] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66. 2001.

[7] R. Dingledine, M. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Designing Privacy Enhancing Technologies*, pages 67–95. 2001.

[8] I. Foster. The grid: A new infrastructure for 21st century science. *Physics Today*, 55:42, 2002.

[9] J. L. Information and J. Liang. Understanding KaZaA, 2004.

[10] M. Kaashoek and D. Karger. Koorde: A simple Degree-Optimal distributed hash table. In *Peer-to-Peer Systems II*, pages 98–107. 2003.

[11] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande. Folding@Home and Genome@Home: using distributed computing to tackle previously intractable problems in computational biology. http://adsabs.harvard.edu/abs/2009arXiv0901.0866L, Jan. 2009.

[12] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192, Monterey, California, 2002. ACM.

[13] P. Maymounkov and D. MaziÃ¨res. Kademlia: A Peer-to-Peer information system based on the XOR metric. In *Peer-to-Peer Systems*, pages 53–65. 2002.

[14] D. S. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-Peer computing, 2002.

[15] S. Naicken, A. Basu, B. Livingston, and S. Rodhetbhai. A survey of Peer-to-Peer network simulators, 2006.

[16] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2):95–98, 2007.

[17] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The state of peer-to-peer simulators and simulations. *ACM SIGCOMM Computer Communication Review*, 37(2):98, 2007.

[18] C. G. Plaxton. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[19] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for Large-Scale Peer-to-Peer systems. In *Middleware 2001*. 2001.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, page 160, 2001.

[21] G. A. Vouros, A. Papasalouros, K. Kotis, A. Valarakos, K. Tzonas, X. Vilajosana, R. Krishnaswamy, and N. Amara-Hachmi. The Grid4All ontology for the retrieval of traded resources in a market-oriented grid. *International Journal of Web and Grid Services*, 4(4):418 – 439, 2008.

[22] W. Yang and N. Abu-ghazaleh. GPS: a general peer-to-peer simulator and its use for modeling BitTorrent, 2005.