SClob - A load balanced P2P content sharing network

Dimitris V. Michail

Submitted to the
Department of Electronic and Computer Engineering
in Partial Fulfillment of the Requirements for
the Diploma of Electronic and Computer Engineering
at the Technical University of Crete.

Guidance Committee
Associate Professor Manolis Koubarakis (supervisor)
Professor Peter Triantafillou
Assistant Professor Euripides Petrakis

September 2002

Acknowledgments

I would like to thank Prof. Peter Triantafillou, for supervising this effort and for his scientific guidance and support. Moreover I would like to thank Associate Prof. Manolis Koubarakis for participating in my jury and for accepting the typical role of the supervisor from June 2002, when Prof. Triantafillou left from Technical University of Crete. Finally, I would like also to thank Assistant Prof. Euripides Petrakis for participating in my jury.

Abstract

The unpredictable growth of the Internet community as well as the size of information available, have overwhelmed the traditional models of distributed computing. Client/server computing seems unable to cope with the constantly increasing need for larger systems.

The peer-to-peer (P2P) model, although originally conceived much earlier, has recently emerged as a new way to create distributed environments. The concept of peers which play both client and server roles seems very attractive, especially in respect to scalability issues.

In this text, we shall present SC-lob, a scalable P2P content sharing system which provides two fundamental properties, load balancing and short query response times. Our work includes the design as well as the implementation of the system in the Java programming language.

Contents

Chapter 1

Introduction

Over the past few years we have witnessed an unpredictable growth of the Internet community. This growth includes not only the number of users but also the technologies provided and the information available. Perhaps the most impressive is the rate at which new "content" is being added to the web every day. Making more and more information available to a rapidly growing user population has become a challenging problem. As a result, the Internet community constantly develops more efficient solutions regarding content availability and distribution.

1.1 Content distribution

Regarding content distribution, the dominating model during the last decade has been the client-server model. Due to it's simple architecture and relative good performance, this model has managed to take over almost every aspect of the Internet infrastructure. Almost every Internet service is divided into two parts, a server and a client. The server's role is to register well known services, while the client's role is to use these services. Due to its simple architecture, this model not only ensures good performance, but also requires little development effort. However, the requirement for central control of information and processing becomes one of its major disadvantages, when the system's size grows significantly. The servers' delays form bottlenecks and

the provided services break down. As a result, new models for distributed computing have emerged during the last few years.

1.2 The Peer-to-Peer model

The P2P model abolishes the need for roles, like server and client, by forming a network where all nodes (peers) are equivalent. Each peer consists of both a client and a server, called a servant. All underlying operations are performed by the collaboration of various servants, without the need to rely on any well-known dedicated peer. For this reason, the main disadvantages of the client-server model[1] are minimized in P2P, since bottlenecks and single points-of-failure are less possible to exist. This scalability offered is one of the major reasons which made the Internet community target its research towards P2P.

In practice the P2P model has been used for various application domains, from file sharing and instant messaging, to distributed file systems and preservation systems. All these different domains of use exploit different advantages of the P2P model while at the same time tackle different problems. Each application domain needs to be examined according to its specific requirements, therefore each resulting design leads to a different use of the model. As a consequence, various P2P architectures exist, from *pure* to *hybrid* where some peers play different role than others.

1.3 Quality of Service (QoS)

A fundamental issue in order to classify the various P2P systems is the quality of service (QoS) they provide. A distributed system's success is strongly dependent on its efficiency and ability to provide high quality services to the users. More precisely, in a P2P network the efficiency of the system can be

[1] P2P networking is not a recent concept. Even the Internet was originally conceived as a P2P network. However the client/server model managed to undertake its development, and only recently the P2P model managed to show its advantages, especially due to the need for wide-area systems.

summarized as:

- The response times, defined as the time taken to satisfy users' requests.
- The throughput of the system, in other words the number of users (requests) it can support in a time interval.
- The quality of the services offered to the users.

Few P2P systems address the issue of load balancing specifically; they either just distribute the objects uniformly in the naming space, by using an appropriate hash function, or just ignore it. A distributed system should fully utilize all available resources, hence we believe that the problem of load balancing is fundamental in this kind of networks. We study extensively the issue of load balancing in dynamic P2P networks and propose a design which uses some ideas from the theory of parallel processors.

The rest of this dissertation is organized as follows. Chapter 4 discusses recent work on P2P systems and load balancing. Chapter 2 presents the details of our SC-lob P2P system architecture. Chapter 3 presents SC-lob's implementation, and finally, chapter 5 discusses our conclusions and future work.

Chapter 2

Architectural Overview

In this chapter we present our SC-lob P2P architecture. Our intention is to create a scalable P2P content sharing system which provides additional properties which have not been addressed by already existing designs. However, we believe that there is a trade-off between the system's scalability and the quality of services it provides. Our design falls into a middle-sized P2P system which can provide services to thousands of peers, while simultaneously trying to divide these services and the available resources equally for the benefit of the whole system and not only a small subset of it.

SC-lob is a content sharing distributed system which provides load balancing and efficient meta-data queries. As a *hybrid-P2P* system, SC-lob is formed from simple and super peers. As we will see in section 2.1, super peers have some additional responsibilities. However, unlike other systems, this role assignment is not static, but rather done automatically, and redundancy is used to prevent the appearance of single points of failure.

Many aspects of our work, which will be presented further on, are "inherited" by work done on [9] which address the same issues. More precisely, many fundamental concepts are maintained, like clustering, division of load balancing in inter and intra cluster phases and use of fairness index, while at the same time significant changes like formation of clusters and differences in load balancing policies result in a totally different system.

2.1 Clustering

SC-lob categorizes peers and groups them in clusters, logical sets of peers. Consider a system with N peers denoted by $P_1, P_2, ..., P_N$ which can communicate through the TCP/IP protocol suite. These peers are logically divided into M clusters, $C_1, C_2, ..., C_M$, where each peer belongs into only one cluster. Each cluster C_i is represented in the system by a small subset of its member peers, which we call super-peers. In definition, this super-peers selection takes into account mostly the processing capabilities of each peer and the intention of the peer to remain logged on for a long period of time. By providing a self-automated fault-tolerant super-peers selection mechanism, the system guarantees indefectible operation. The use of several super-peers on each cluster provides some redundancy on the access point of each cluster, and therefore reduces the chance for single point-of-failure. Moreover, the super-peers selection takes into account the peers' processing capabilities, is self-adjustable and no critical operation is based on their existence.

Every peer in the system maintains a clusters-repository, where, for each cluster, identification information about the relevant super-peers is kept. In other words, super-peers are the representatives of each cluster to the rest of the system, giving a first access point. Every possible change of the representatives in a cluster, is automatically followed by a report to the rest of the system's clusters, therefore keeping global identification information consistent. The report mechanism used will be discussed in more detail in chapter 3. In sort we could say that first the report is propagated to one super-peer of each cluster (based on a virtual binary tree) and second is again propagated inside each cluster by a super-peer to the other cluster peers (again based on a virtual binary tree topology). One could argue that the super-peers of many clusters could possibly crash simultaneously. We address this issue by introducing a caching mechanism, where each peer keeps information about various peers which are likely to belong to a cluster. These peers may have been discovered from data transfers or by query redirection, issues which will be discussed further on. When the super-peers of a cluster cannot be addressed, then these cached entries are tried instead.

2.1.1 Shared meta-data repository

As in every content sharing system, each peer i possesses a number of $|d_i|$ shared data objects[1] , and for each of these documents a small meta-data description. Each cluster uses redundancy of the meta-data information as a means to reduce the query execution time. More precisely, each peer maintains a meta-data repository, which contains the meta-data descriptions of the whole cluster it belongs to. The meta-data repository is kept synchronized by forcing each peer to announce the existence of its shared objects. Although this operation introduces a small network overhead, the benefits, such as cluster querying in O(1) complexity time, cannot be refuted.

2.1.2 Categories & related metadata

Query execution time is further reduced by SC-lob's categorization of data. All data objects are either automatically, or interactively (by the user), assigned to some predefined categories. This assignment, together with the fact that each cluster announces to the rest of the clusters the categories it possesses, reduces the search space of queries. The user can narrow down the query's search space by specifying the categories that interest him/her, and consequently the query is submitted only to the clusters where such categories are known to exist. This categories-aware mechanism, combined with the clustering, provides not only a fast global view of the system's content, but also guarantees an upper bound to the query execution time. We must point out that this categorization of items is based on the type of its object as used on the WWW (e.g filename extension) and is only used as a speed-up on queries. No specific grouping of peers into cluster is performed based on categories.

[1] Throughout the rest of this text, the terms data object and data item are used interchangeably, unless otherwise stated.

2.1.3 Model & performance issues

References to upper bounds on query execution times as well as complexity issues are directly relevant with the distributed model assumed [14]. Our system is a n-node directed graph $G(V, E)$ formed from peers. Each peer is a node in the graph, and is modeled as an I/O automaton, P_i. P_i has some input and output actions by which it communicates with an external user. In addition, P_i has outputs of the form $send(m)_{i,j}$, where j is an outgoing neighbor of i and m is a message, and inputs of the form $receive(m)_{j,i}$ where j is an incoming neighbor of i and m is a message. We consider one kind of faulty behavior on the part of peers: stopping failures. A stopping failure of P_i is modeled as a state where all tasks of P_i are stopped. We do not currently consider Byzantine failures which are directly related with security matters (they are however on our future work plans). Regarding the communication channels, we model each directed edge (i, j) of G as an I/O automaton $C_{i,j}$. Its external interface consists of inputs of the form $send(m)_{i,j}$ and outputs of the form $receive(m)_{i,j}$. The channel's properties are directly related to properties of TCP/IP, so messages cannot be damaged or lost without the sender being notified. Our distributed model is asynchronous; each peer's actions are directly related only to its current state and not to any notion of time or rounds.

Two measures of complexity are considered for our asynchronous distributed algorithms: time complexity and communication complexity. Time complexity is measured in terms of an upper bound l for each possible action. We usually assume that sending a message through an edge (i, j) takes one time unit and that each action executed on a peer takes also one time unit. It is assumed that link failures resulting on re-transmition of messages add a constant amount of extra time units and don't actually change the complexity result. However, this assumption is only true if there is a maximum number of link or peer failures that can occur during an algorithm's execution. Communication complexity is typically measured in terms of the total number of constant-size messages transmitted. Large messages may be considered as many smaller ones.

2.1.4 Clusters & peers communication

The system can be seen as peers which form two kinds of networks. Inside each cluster the member peers form a complete network, in definition a network where links exist between each pair of peers. The various clusters form a second complete network through their representatives (super peers). Although they both are complete networks, the messages exchanged between peers or between clusters (by super peers) exploit some form of virtual topology. Various algorithms, like the synchronization of each peer's meta-data repository, use this virtual topology inside a cluster, as a route to forward messages to all peers. This virtual topology routing mechanism was designed while keeping in mind the scalability of the system, and therefore should have logarithmic time complexity. Our implemented algorithm uses a virtual binary tree in order to route messages, which results in $O(log_2(N))$ time where N is the number of peers in a cluster, or the number of clusters in the system. Details about the routing algorithm and implementation issues will be discussed in section 3.4.

2.1.5 Cluster formation

We shall now discuss how clusters are formed. The login procedure for a peer P_i is just a matter of finding, by out-of-band means, another peer's address, P_j, which is already in the network, and submitting a login request. The new peer automatically becomes a member of the cluster of which peer P_j is already a member. Although all peers are accepted in a cluster, there is a maximum number of members in each cluster due to scalability issues[2]. When a cluster reaches the maximum number of members, a cluster split event occurs, resulting in two new clusters formed by the members of the original one. The splitting follows various criteria based mostly on load distribution and will be discussed in more detail in chapter 3. Similar to the maximum peers there is a minimum number of peers that can be in a cluster at any one time. When the number of members reaches this minimum number, the

[2]The performance of intra-cluster actions can become an overhead for the cluster's operation.

cluster members try to log into some other cluster with a valid number of member peers. Except from the clusters participating in the join, the rest of the system's clusters are notified about any change through the message routing mechanism, in order to keep the global identification information consistent.

2.2 Load balancing

One of the most important issues on the efficiency of a content sharing system is that of load balancing. When we refer to load balancing in a content sharing system we mean that all peers should have equal normalized workloads, according to their ability to serve. The notion of workload or popularity of a resource is directly relevant with the probability of that resource being requested. When we say that a resource has popularity equal to 10%, we mean that 10% of the requests on the system will be pointed to this particular resource. In practice, the services provided by a load-balanced system are highly available to all users without any exceptions, while at the same time the possibility of malfunction caused by an overloaded peer is minimized. It is a fact that the system's fault-tolerance and robustness are highly dependent on the distribution of the workload across all peers forming the network.

The use of clustering allows us to divide the problem of load balancing into two subproblems:

- *Inter-cluster load balancing*: In other words all clusters should have equal workloads.
- *Intra-cluster load balancing*: All peers inside a cluster should have equal normalized workloads. All requests for documents hitting the cluster, either data-transfers or queries, should be equally divided between all member peers, according to their ability to serve.

The metric we use for evaluating the system's balanced state is "fairness", discussed in [15] and used in [9] for P2P load balancing. The fairness index

is defined as follows. Let a system allocate resources to n users, such that the i^{th} user receives an allocation x_i. Then the fairness[3] index is

$$f(x) = \frac{(\sum_{i=1}^{n} x_i)^2}{\sum_{i=1}^{n} x_i} \tag{2.1}$$

We use equation 2.1 to evaluate the load distribution of clusters and peers, when solving inter and intra-cluster load balancing respectively. Possible values are always between 0 and 1. The closer the value to 1, the fairer the load distribution becomes.

In sections 2.2.1 and 2.2.2 we first present various properties and mechanisms of the system which are basic in our load-balance design, which is then presented in 2.3 and in 2.4.

2.2.1 Dynamic cluster behavior

Our design follows directly the dynamic nature of our P2P network. We assume that peers will enter or leave the system, either willingly or due to failures and, at the same time, content will be added or deleted. This dynamic nature constantly changes the workload distribution and requires a monitoring mechanism. The various fundamental operations which can occur in the network are summarized as follows:

- A peer can enter the system, by logging into a cluster.
- A peer can leave the system, either willingly or due to failure.
- Content is added by a peer.
- Content is deleted by a peer, either willingly or due to failure.

Except from these fundamental operations, our system makes decisions in order to change the workload distribution and to maintain it's scalability. For this reason we introduce some other possible dynamic actions, which are:

[3]The fairness index can be interpreted as the percentage of users that share the used resources equally.

- A peer can change clusters, by logging out from one and logging into another.
- Content can be replicated between peers of the same cluster.
- A cluster can split or join with other clusters if the amount of member peers exceeds some predefined threshold values.

We shall discuss these actions in the next sections. However, we must point out that decisions about such actions are directly related with the monitoring mechanism.

2.2.2 Popularities monitoring

As already mentioned the system is dynamic, therefore providing load balancing requires monitoring of popularities. In a content sharing system the popularities measurement can be performed by counting the hits of various data items and/or by estimating the future popularities from previous values.

Suppose a peer i has data items $d_1, d_2, ..., d_n$ with hit counts $h_{d_1}, h_{d_2}, ..., h_{d_n}$ respectively. A peer's workload is calculated as the sum of the hits counts of all data item it possesses. The popularity of a peer is the percent of its workload with respect to the total workload of the system. However, since the peers are heterogeneous, we assign to each peer a *processing capacity*, pc_i, either by a benchmarking tool or by the user's selection, and normalize the peer's workload. This allows us to have a unique approach, regarding load balancing, despite the differences and the nature of each peer since the assigned processing capacity reflects the processing capabilities as well as the network capabilities of the peer (maximum possible bandwidth). The resulting workload formula is:

$$W_i = \left(\sum_{j=0}^{n} h_{d_j} \right) / pc_i \tag{2.2}$$

where pc_i is the processing capacity of peer i. The peer popularity formula is W_i divided by the total workload of the system. Subsequently, we can

calculate each cluster's popularity by just summing the popularities of its member peers, as follows from the fact that each peer belongs into only one cluster.

Except from the monitoring mechanism, a reporting mechanism is also used for the popularities. Each peer monitors the hit counts of its data items, and every time a significant change occurs, a popularity report message is forwarded to the other cluster members. Likewise, a popularity report message is sent from the super-peers of each cluster to the other clusters when and if the change of the cluster's popularity is significant. Reporting is not performed for very small changes since we do not wish to introduce network delays by flooding the network with control messages. As of this, each peer in the system is aware of:

- The popularities of the other members of its cluster.
- The popularities of all clusters.

2.3 Inter-cluster load balancing

By the term inter-cluster load balancing we mean that all clusters should have equal workloads. Remember that the workload or popularity of a cluster is measured as a sum of the workloads or popularities of its member peers. As of this, changes in a cluster's popularity can be achieved by moving peers from one cluster to another, an operation which we call "peer migration". We describe next our approach in more detail.

Consider the following abstract distributed load balancing problem. We are given an arbitrary, undirected, connected graph $G(V, E)$ in which each node $v_i \in V$ contains a number l_i of current workload. The goal is to determine a schedule to move an amount of workload across edges so that, finally, the weight on each node is equal. This problem describes inter-cluster load balancing in our distributed system when we associate a node with a cluster, an edge with a communication link of unbounded capacity between two clusters and l_i with the popularity of cluster i. Since the system works on

top of tcp/ip we assume that the graph $G(V, E)$ is complete, at least when load balancing actions take place.

One way to re-balance the load is to repartition the peers into new clusters. However, with this approach it is difficult to ensure that the new partitioning will be "close" to the original partitioning. In case the new partitioning deviates significantly from the old one, the cost of large data transfers becomes unacceptable. An alternative strategy to change the load distribution is to migrate peers (load) among clusters. Peer migration needs only the meta-data of peers to be transmitted and can be considered a low-cost operation, in respect to the amount of data that is moved during the normal operation of the system.

A practical approach to dynamic load balancing is to divide the problem into the following phases [16, 11]:

1. *load evaluation*: The load of each cluster must be known, or estimated, so that an imbalance can be detected. Further actions are initiated if the imbalance is above a threshold, e.g 90%. This imbalance is calculated with the fairness metric and describes the percentage of the clusters which share the resources equally. This means that a fairness of 90% denotes that 90% of the clusters are equally sharing the workload of the system.

2. *"flow" vector calculation*: Based on the measurements taken in the first phase, the ideal load transfers necessary to balance the system are calculated. Note that no actual movement is performed before the final "flow" is calculated, since it may result in more data-migrations than necessary.

3. *load selection*: Peers are selected for transfer or exchange, between clusters, to best fulfill the "flow" vector calculated by the previous step.

4. *load migration*: Once selected, peers migrate from one cluster to another.

2.3.1 Load evaluation

The load evaluation is performed from the monitoring component of the system, as described in section 2.2.2. As mentioned, every peer is aware of all clusters' popularities so no actual extra work is needed for this phase. From the measurements of load, the each cluster (by its super-peers) calculates the current imbalance, by evaluating the fairness metric. However, when a small imbalance exists, the system can waste resources without achieving any significant progress. This is due to the non-negligible cost of peer migration, and to the peers' popularities granularity, which plays a significant role in the maximum balance that can be achieved. Subsequently, a threshold value is introduced, and only if the imbalance is above this threshold value any subsequent load balancing actions are triggered.

2.3.2 "Flow" vector calculation

The flow calculation phase is responsible for the calculation of the amount of load to migrate from each cluster in order to have a uniform load distribution. A local iterative diffusion policy [10] is used, where the load of cluster i from time t to time $t+1$ is modeled by:

$$l_i^{t+1} = l_i^t + \sum_{j \in A(i)} \alpha_{ij}(l_j^t - l_i^t) \quad 1 \leq i \leq n \tag{2.3}$$

$$\alpha_{ij} = \frac{1}{|A(i)| + 1} \tag{2.4}$$

where $A(i)$ is the set of neighbors of cluster i and α_{ij} is called the diffusion parameter of i and j, which determines the amount of load to be exchanged between the two clusters.

One iteration is needed in order for this approach to converge when the graph $G(V, E)$ is complete. Although our clusters' topology is a complete graph, a potential increase in the number of clusters will introduce communication overhead and, at the same time, can result in "flow" calculations which will not be satisfiable. If the number of clusters is large, each cluster may have to send a small amount of load to each of the other clusters, which

is likely to be possible. Thus, in such cases, in order to address scalability issues as well as to perform better load balancing, we choose not to treat the clusters as a complete network. We can use various virtual topologies, like a tree, which result in slower convergence[4] of the "flow" vector calculation but at the same time are more scalable and efficient. The introduction of small neighborhoods of clusters, as the size of the system grows, is also helpful in a highly dynamic environment. When the system behaves highly dynamically, with respect to the peers and content, fast local load balancing actions are preferred [16] to slower global actions.

Furthermore, the possible load movements between clusters are completely dependent on the load of the member peers on each of the clusters. When load granularity is too coarse, it prevents any movement and does not permit the achievement of good balancing. In order to prevent unnecessary time-consuming calculations, with respect to load selection for migration, as well as to minimize the network overhead for peer migrations, we filter the calculated "flow" vector and rule-out very small load movements which will not result in significant load balance improvements.

2.3.3 Load selection

After the "flow" calculation phase each cluster is aware of the amount of load that should be transmitted through each of its outgoing edges. A cluster i can send load to another cluster j by selecting some peers to migrate to cluster j. The selection of the peers which should migrate from a cluster to other clusters, in order to satisfy the "flow" vector, is the work done in this phase of our load balancing algorithm.

Suppose we have two clusters C_i and C_j and that we want to move an amount of load L_{ij} from cluster i to j. Suppose also that cluster's i members is the set $P = \{P_1, P_2, ..., P_N\}$ with load $l_1, l_2, ..., l_N$ respectively and that the two clusters are interconnected with a communication link $C_i \leftrightarrow C_j$. We want to find a subset of P with total load as large as possible but not

[4]Depending on the topology assumed, the second degree diffusion[13] method may be used to increase the convergence rate

```
1 APPROX-SUBSET-SUM( S , t , ϵ )
2   n = |S|;
3   L_0 = (0);
4   for( i = 1; i <= n ; i++ )
5     L_i = MergeLists( L_{i-1} , L_{i-1} + x_i );
6     L_i = Trim ( L_i , ϵ/n );
7     remove from L_i every element greater than t;
8   let z be the largest value in L_n;
9   return z;
```

Figure 2.1: Subset Sum fully-polynomial approximation

larger than l_{ij}. This problem is an instance of the subset-sum problem which has been proved to be NP-complete. Subsequently we cannot use a naive approach with exponential complexity, but rather some faster method with an acceptable error rate. A fully-polynomial approximation scheme [17] is used, in order to solve the problem in acceptable time, especially as the number of peers becomes larger. The scheme used uses "trimming" in order to reduce the search space of the problem. To trim a list L by a parameter δ, where $0 < \delta < 1$, means to remove as many elements from L as possible, in such a way that if L' is the result of trimming L, then for every element y that was removed from L, there is an element $z \leq y$ still in L' such that $(1-\delta)y \leq z \leq y$. In other words we can say that z is "representing" y in the new list L'. Each y is represented by a z such that the relative error of z, with respect to y, is at most δ. Pseudo code for such an algorithm is shown in Figure 2.1.

Actually our load selection problem is a multiple subset-sum, since for each cluster we must select some load to send to more than one outgoing link. The optimal solution in this particular problem requires time, and surely cannot be calculated in an application which requires real-time response. In order to decide which peers will migrate from a cluster to other clusters we use a simple greedy algorithm, extended with simple heuristics[5] which are likely to provide a good solution for this particular instance of the problem.

[5]Example of heuristics is the common categories possessed from the target cluster and the candidate peers, the processing abilities of the final cluster (after the migration), etc.

The various load movements are sorted in descending order and a subset sum approximation is executed for each of them in turn.

2.3.4 Peer migration

The final step of our algorithm is the actual peer migration from one cluster to another. With peer migration, the load distribution of the clusters is changed, resulting in a balanced state. The actual algorithm about peer migration is implementation dependent and will be discussed further in chapter 3.

2.4 Intra-cluster load balancing

The second part of our algorithm is intra-cluster load balancing. Since we have already balanced the workloads of the clusters we now wish to balance the workloads of the peers inside each cluster[6]. Requests hitting a cluster, either for a data object or for queries, should be equally distributed to all member peers, according to the processing abilities of each peer. As discussed in section 2.2.2 the abilities of each peer are represented by a *processing capacity*, pc_i, a measure of the peer's overall performance.

Replication of data objects and redirection of requests are our main intra-cluster load balancing actions. All query requests hitting a cluster through a peer i are redirected or not to another peer according to the processing capacities. All peers in the cluster can handle the query request, due to the shared meta-data repository that is maintained. The probability that a query request is handled by a peer i is

$$p(i) = \frac{pc_i}{\sum_{j=1}^{N} pc_j} \tag{2.5}$$

when pc_i is the processing capacity of the peer i and N is the number of peers in the cluster. A similar redirection is performed when a requested data object from peer i has replicas located on other peers of the same cluster.

[6]Actually the two load balancing subproblems are not exactly independent. This is why we schedule both actions periodically on each peer which results in random execution (the peers are not synchronized).

If $\{P_1, .., P_n\}$ is the set of peers which contain a copy of the requested data object d, then the data object is retrieved from peer i, with probability as shown in equation 2.5.

Each cluster in order to ensure load balancing replicates documents between peers. Each peer of the cluster is authorized to create replicas for the documents it holds. A peer decides where and when to replicate a file, based on knowledge about the workloads of the other members of the cluster, obtained through the popularities' reporting mechanism. Decisions about deletions are also locally made, by peers which possess the respective replicas. This approach ensures that our replica management mechanism is completely decentralized and scalable.

Intra-cluster load balancing, achieved by replicas creation and deletion, is not entirely independent with the inter-cluster load balancing actions. However when a date item is replicated in a cluster, the load is divided between all owners of this item's copies (if a data item is copied due to a user's request, it is not considered a replica and does not take any load), which results in very small change of the workload of the cluster. However, this change is small and we argue that the dynamic nature of the system has surely worse effects. Due to the periodic execution of the load balancing actions and the randomness imposed (each peer may login in different time and no global time is assumed) there is no convergence problem due to a small dependence between the two load balancing actions. Remember that our intention is to load balance a dynamic system, which imposes continuous actions but it's goal is not to reach perfect load balance as this cannot be achieved due to heterogeneously and granularity constrains.

2.4.1 Replicas management

A possible creation of new or the deletion of already existing replicas is an operation which requires some identification information. Data objects' descriptions in the shared meta-data repository, which is kept at each peer, contain explicit information about the identification of replicas, their locations, and popularities. Data objects are assigned into three categories based

on the state of replicas:

1. *Original item*: A data object which was published from a peer, and can be replicated as needed. All data objects which were created from data-transfers initiated by a user belong to this category.

2. *Replica*: A data object which is a replica of an original item, located on a different peer from the original , and through redirection handles a fraction of the requests. Replica chaining, that is creation of replicas of replicas, is not permitted. Only original data objects can be replicated, possibly to more than one peer in the same cluster.

3. *Floating replica*: A replica data object which was obtained from a peer that no longer exists in the same cluster. The originating peer may have disconnected, crashed or moved to another cluster through "peer migration" based on inter-cluster load balancing decisions. These replicas are not automatically deleted when the original item is deleted. A possible deletion of this kind of item occurs by the replica creation/deletion policy based on the intra-cluster load balancing actions.

Each peer in a cluster, based on its load, is authorized to make decisions about replica creation or deletion. Suppose a cluster has average load L_{avg}, and peers have loads $l_1, l_2, ..., l_n$. Each peer i which has more load than the average ($l_i > L_{avg}$) can replicate a local data object to a target peer j which has less load than the average ($l_j < L_{avg}$). The replica is automatically assigned, on creation, a fraction of the load of the other existing copies, based on the processing capacities of the peers holding these copies. On the other hand, a peer can also delete a replica, therefore reducing its load and increasing the load of the peers which are holding copies of the same object. Finally, floating replicas can also be deleted as a means to reduce load, without affecting other peers' load distribution.

Candidate data objects for replication are those who have high popularity and at the same time their size is limited. A peer's replica creation algorithm sorts the available data items in descending order based on the fraction $l_{d_i}/size_{d_i}$ where d_i is a shared data object, l_{d_i} is the load of the

object, and $size_{d_i}$ its size. The resulting greedy algorithm takes also into account that the popularities of the objects follow a zipf distribution meaning that only a small fraction of them are quite popular to be worth of replication. The algorithm tries in turn the first m popular data objects, by testing the resulting fairness of the system for every possible peer which could obtain a replica. While the fairness index increases, new replica items are created.

Unfortunately, some restrictions exist on the maximum load balance that can be achieved locally. These restrictions are based mostly on the available disk space that each peer has for the creation of replicas. Furthermore, due to the heterogenous nature of our system, not every replica can be created on each peer. Peers with slow network connections can only replicate small data objects. A possible solution to this problem, which is in our future work plans, is to be able to replicate just a fraction of a data object.

Chapter 3

Implementation

Our work extends beyond just the design of a load balanced P2P system, into implementation. We have chosen to implement a champion application which will operate as discussed in chapter 2. The system's implementation is done in Java, resulting in a cross-platform application, and provides the core system as well as a GUI front-end. This chapter discusses in more detail various parts of the system's design.

3.1 Clustering

Clustering requires various data structures to maintain information which is crucial to the identification of peers and clusters. Each peer in the system maintains three important repositories:

- **A local cluster members repository**: Here a peer keeps information about the peers which belong to the same cluster. The information kept is related to network communication (IP, port, etc.) as well as processing capacity and free disk space for replicas.
- **A shared data objects repository**: This placeholder holds the meta-data information of the cluster, in which a peer is a member.
- **A remote clusters repository**: For each of the clusters in the system, one entry in this repository contains identification information.

Identification information for a cluster are the UIDs of the super-peers, their IPs, port numbers, etc.

3.1.1 LCM repository - local cluster members

Information about a cluster and its members is kept in this repository. When a peer logs into the system, through a peer which belongs into a cluster, sends a message which instructs all the rest cluster members to add a new entry into this repository. This entry describes the network address of the peer, its processing capacity which is a constant number denoting performance abilities, a 64 bits unique identifier which identifies the new peer, its intention to remain in the system for a long period and the free space for replicas creation.

All peers belonging to a cluster are kept sorted in this repository in descending order, first by their processing abilities, second by their intention to remain in the system and finally by their unique identifiers. This sorting is exactly the same in all peers, thus allowing us to obtain the representatives of the cluster by just selecting the first few peers. When a change of the representatives occurs, a report message is forwarded to the rest of the clusters. This report message reaches all peers in the system, by first reaching the representatives of each cluster and secondly by reaching the peers in each cluster, and is used to update the identification information about the cluster which is stored in the **RC** repository described in section 3.1.3.

3.1.2 SDO repository - shared data objects

Each peer in the system has a file system layer which triggers automatic events when files are either added or deleted. Addition or deletion of data items can occur either by a user requested data-transfer, a automatically created replica from load balancing or by external events on the file system. All new files created are immediately considered shared. For each shared file a meta-data description is kept in this repository. This meta-data description contains various fields like name, title, size, category, hits count, etc. Each shared file can be identified by a unique 32-bits identifier. Furthermore, the

meta-data kept provides information in order to identify replicas and their corresponding original files.

A new-file event fired by the file system layer, results in storing meta-data information into the repository, as well as transmitting this information to the rest of the cluster members. Each peer in a cluster is aware of all the shared objects which exist in various member peers in the cluster. Similarly, when a file is deleted or the user no longer wishes to share it, its entry is deleted from the repository, both locally and at the rest of the cluster members (with the use of delete messages). In this way each peer has synchronized information about the shared objects and can answer queries about the whole cluster.

Each peer keeps a hits count for each shared file which is locally stored. This hits count is updated each time a new download request arrives. The peer regularly checks the total hits count and when a significant change occurs, a popularity report (containing the total hits count) is forwarded to the rest of the cluster's members. This significant change is defined to be a fraction of the last reported value, for example when the hits count changes by more than 5% of the previous forwarded popularity report. This popularity report is also stored in this repository, allowing each peer to be aware of the popularity of each member of the cluster, and consequently to be able to decide about replicas creation or deletion.

The popularity reporting mechanism ensures that each peer has an almost up-to-date view of the popularities of the other cluster peers. However, when a peer crashes and all other cluster peers remove its entry from their repository (by a message arrived which was initially sent from the peer which realized the crash), the hits of all replicas must be transferred to the appropriate items. For this reason, each peer periodically sends not only the total hits count (popularity report), but also the hits of the replicas it possesses.

3.1.3 RC repository - remote clusters

Identification information for all the clusters of the system is kept in this repository. For each cluster a small subset of its members as described in section 3.1.1 is kept, synchronized through reports which are generated when

changes occur. Each cluster's entry is uniquely identified by the cluster's unique identifier which is 32 bits long. Furthermore, other information is also kept for each cluster, as a mean to reduce the query execution time, as well as to perform load balancing actions. We could summarize the information kept about each cluster of the system as follows:

- **Unique identifier**: A unique 32-bits identifier of the cluster.
- **Representative peers**: A small subset of the cluster member peers. These member peers are the most stable and fast peers of the cluster.
- **Categories**: The categories in which the clusters data items can be assigned. This information is used mostly for querying but can also be used for other purposes.
- **Total popularity**: The total popularity of the cluster, which is the sum of the normalized popularities of the cluster's member peers.

Except for the unique identifier which is a constant, all other information is updated when appropriate. Each cluster, through its representatives, sends up-to-date information about representatives, categories and popularities. These updates, through the forwarding mechanisms, eventually reach all the peers of the system.

The **RC** repository keeps all clusters sorted by their unique identifier. This sorting is useful in our message forwarding mechanism which will be discussed in section 3.4.

3.2 Dynamic cluster behavior

The clusters, as discussed in chapter 2, have a maximum and a minimum limit of member peers. When the number of peers exceeds these limits either a cluster split or join occurs.

A cluster split occurs when the number of peers becomes more than allowed. This is done to preserve the scalability of each cluster, since more peers add additional delays, mostly in the forwarding of control messages. A

cluster split is decided by the leader (the first of the super-peers based on their unique identifiers) of the cluster, which is the first peer in the sorted array of peers kept in the **LCM** repository. In order to begin the cluster split, the leader runs a subset sum algorithm (see figure 2.1), trying to divide the cluster in half, based on the peers' popularities. Finally, after the new cluster has been determined, a cluster split message is forwarded to all cluster members.

When a split message arrives, each peer checks whether it should stay at the old cluster or move to the new one (the message contains information which defines which peers should create a new cluster, all other peers stay behind). In case the peer must stay at the old cluster, then a simple operation is performed where the **LCM** repository is updated by deleting the peers of the new cluster. On the other hand if the peer should belong to the new cluster, then new empty data structures (**LCM** and **SDO** repositories) are created and updated with the information of the other peers (also belonging to the new cluster). Furthermore, a new cluster entry is added to the **RC** repository and finally the leader peer announces the new cluster existence to the other clusters. The information of the old cluster is not immediately deleted since various messages about the old cluster could still be traveling and not reach all of the old cluster's members.

Cluster join is a much simpler operation. When the number of peers becomes less than the minimum allowed, one by one the member peers start migrating to other clusters. The peers start to migrate in the reverse order from that of the **LCM** repository. Each peer chooses randomly to migrate to a cluster which has number of peers more than the minimum allowed. In case all cluster have simultaneously less peers than minimum no migration is performed. The number of peers of a cluster is known by the popularity reports which are periodically sent. This prevents two clusters which automatically run cluster join from exchanging members. Finally, the last peer migrating, sends a message to instruct the rest clusters to delete the cluster from their **RC** repository.

3.3 User actions

As in P2P content sharing systems, user actions can be divided into two main categories; querying and data transferring. We will now discuss how these user actions are executed and handled by the system.

3.3.1 Queries

Queries are submitted by a user through the GUI front-end of our application. A user can specify keywords as well as the category which interests him and expects to see the most popular files that match these criteria. Through the **RC** repository, the clusters possessing documents in the query's category are found and a query request is submitted to them in turn. The local cluster can be immediately queried by the peer since the **SDO** repository contains all relevant meta-data descriptions.

A query is submitted to one of the super-peers of each cluster. However these peers do not necessarily handle the query, but instead redirect (or not) the request to another peer based on their processing capacities. The peer which will handle the query, locates the most popular data items which match the query and sends a query response. This redirection is a basic operation of intra-cluster load balancing. Our query mechanism provides a fast global view of the system's content, resulting into $O(M)$ time complexity where M is the number of clusters of the system. As we can see, the number of clusters consists an upper bound on the query's execution time.

3.3.2 Content retrieval

A query can be followed by a request for retrieval of some data objects. Except from identification information about the data objects, information about the peers holding these objects is also included in query responses. As follows, a retrieval request is not addressed to a super-peer but directly to the appropriate owner.

Similarly to the query handling, intra-cluster load balancing is achieved by a redirection mechanism. A peer checks whether replicas of the requested

object exist on the cluster and redirects the request appropriately based on the processing capacities of the object holders. Replicas identification is easily achieved by the **SDO** repository which maintains enough information about their existence as well as their location.

3.4 Control message routing

We have already mentioned that our message forwarding mechanism provides $O(log_2(N))$ time complexity, in both cases of intra and inter-cluster forwarding. First of all, intra-cluster forwarding is the routing of messages from peers inside a cluster. Similarly inter-cluster forwarding is the routing of messages between clusters by their representatives (super-peers).

In both cases we keep a sorted representation of the various possible recipients of messages. For example the **LCM** repository always keeps the member peers of a cluster sorted. The same is also true about the **RC** repository which always maintains a sorted list of the clusters. This sorting is consistent between all peers of the system (at least when each peer has received the same control messages). A peer, when forwarding a message to its cluster or to remote clusters, uses either the **LCM** or the **RC** repository to find the appropriate recipients.

Suppose a cluster has N peers, $P_0, ..., P_i, ..., P_{N-1}$, and that peer i sends a message which should reach all the cluster members. The message is transmitted in a form of binary tree, where peer i is the root node. The set of peers is divided into two subsets where in each subset the same algorithm is executed recursively. This subset division is achieved with the use of a field called *uptoUID* which is present in every message which is transmitted. Figure 3.1 shows two examples for message routing. The routing is accompanied by a duplicate identification mechanism, to avoid handling of an already received message. Furthermore, each message has a hops-count which is reduced by one in every peer. Both duplicate identification and hops mechanisms are used to ensure correct message routing, in cases where multiple peers crash simultaneously. We would like to point out that this message routing mechanism does not always preserve the ordering of the

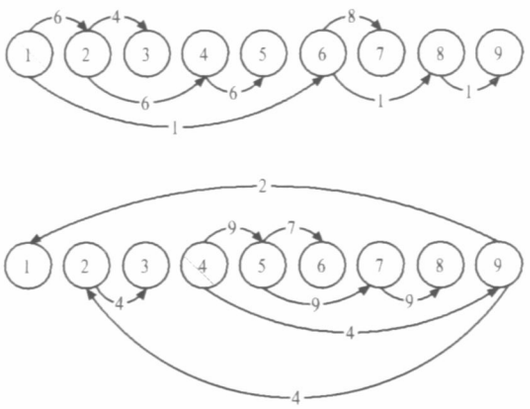

Figure 3.1: **{inter,intra}-cluster routing**. In the upper figure, the initial sender is 1, while at the lower figure, the initial sender is 4. The arcs' numbers are the field *uptoUID* of the message, which denotes up to where the message should be forwarded. Messages are always forwarded to the right in a modulo N manner, when N is the number of peers or clusters.

messages, however our algorithms do not require a total ordering in order to work robustly.

3.5 Fault tolerance

Fault tolerance is achieved by correctly handling network timeouts and retries. Each message transmition is tried several times with a small network timeout (e.g 15 sec) and only after several unsuccessful attempts it is assumed that the target peer has crashed. In case of transmition of messages between clusters, a peer tries to send the message to every super-peer which is known for the cluster, and only after failure in all attempts a cluster is considered crashed. However before we actually consider the cluster as unreachable a peer tries to communicate with some cached peer entries which were obtained either by query redirection or by a data-transfer.

When a peer realizes that either a peer or a whole cluster can not be reached, an update message is immediately sent to the appropriate peers. In case a local cluster peer is down a message is sent to the local cluster

members. In case a whole cluster is down a message is send globally to the other clusters which is in turn transmitted to all peers of the system.

3.6 Login, logout & cluster migration

The login procedure is just a matter of finding another peer which is already in the system (in a cluster), by *out-of-band* means. Subsequently a login request is made, which is immediately followed by a response which describes the current state of the system as well as of the cluster. Moreover, the response contains the various clusters and their representatives, the local cluster members and their popularities and the meta-data descriptions about the shared items of the local cluster. By the end of the login procedure the new peer is totally synchronized with the peer which received the login request.

The routing mechanism used in the cluster (and between clusters), uses two recipients for each message forwarded. The one recipient is the direct right neighbor of each peer, in respect with the sorted representation of peers in the **LCM** repository (see section 3.4). During the login procedure many messages can flow in the network and we would not like the new peer to lose any of them. This is why a login request is redirected from the peer made to another peer in the cluster which is the one who will be the direct left neighbor of the new peer, and therefore will forward all messages to the new peer.

Logout is a much simpler operation than login. Each peer who wishes to logout transmits a logout message to the cluster. Each peer who receives the logout message, updates its repositories to reflect the deletion of the peer, and the possible deletion of replicas or replicated items which existed at the peer.

Cluster migration, a fundamental operation on the achievement of inter-cluster load balancing, is performed by a login followed by a logout procedure. The migrating peer first logs into the target cluster and then logs out from the source cluster. The repositories of the old cluster are not immediately deleted, since various messages could have already been forwarded. The

peer, although not anymore a member of the cluster, continues to forward any messages it receives to ensure correct operation of the forwarding mechanism.

3.7 Load balancing

3.7.1 Inter-cluster

Our inter-cluster load balancing implementation follows directly our design presented in chapter 2. The various steps which were earlier presented are executed one by one, resulting in a load balanced system.

The popularities monitoring and reporting mechanism constantly provides the peers of the system with an almost up-to-date view of the load balanced state of the system. Each peer periodically checks whether an inter-cluster load balancing action should be performed. However since the popularities reporting could be a little bit different from peer to peer (we do not use strong consistency) we choose to use a somewhat centralized approach for the step of the "flow" vector calculation.

The leader peer of each cluster is responsible for the calculation of the load that should be moved to other clusters. The actual size of the system, that is the number of clusters, is checked before choosing the actual topology for the diffusion algorithm. As the number of clusters grows, we reduce the edges of the graph $G(V, E)$ where V are the clusters (nodes) and E are the edges (communication links). After the "flow" vector has been calculated the result is filtered, and very small movements are removed, until a threshold (fairness) is reached (e.g 90%). Finally, each cluster leader follows the diffusion results and tries to select peers to migrate to other clusters in order to satisfy the flow requirements. The selection is performed as outlined in section 2.3.3. The peers selected are informed by a "migration" message to execute a cluster migration procedure.

3.7.2 Intra-cluster

Each peer periodically executes the local load balancer, a thread which is responsible for the creation or deletion of replicas in order to have a uniform load distribution inside each cluster. If and when the calculated load state is below a threshold value (e.g 90%), further intra-cluster load balancing actions are initiated. The load balanced state is measured by the fairness index, discussed in section 2.2.

In case the fairness of the load is below the threshold, each peer checks for whether the resulting fairness will increase by performing some possible actions. Each peer tries in turn the following actions in order to equalize the load inside its cluster:

- *Deletion of floating replicas*: Each peer may possess a replica from a peer which is no longer a member of the cluster. This replica is called "floating" and has no dependency on any other items or peers, thus its deletion affects only the load of the current peer. The peer tries all floating replicas, one by one, sorted ascending by the fraction $\frac{p_i}{size_i}$ where p_i is the popularity while $size_i$ is the size of the replica, and deletes them if the resulting fairness of the cluster is better than the current one. This particular sorting is preferred, since we would like to free additional disk space for other possible replicas.

- *Deletion of non-floating replicas*: A peer may also possess items which are replicas from other peers on the same cluster. For each of these replicas, again sorted ascending by their $\frac{p_i}{size_i}$, the resulting fairness is calculated and, if appropriate, the replica is deleted. Note that, unlike floating replicas, a non-floating replica affects the load of other peers' as well as the current peer's.

- *Creation of replicas*: A peer may replicate a local item to another peer of the cluster. This action will result in reducing the load of the current peer and possibly increasing the load of other cluster peers. The various local items are sorted descending by their $\frac{p_i}{size_i}$, and one by one are checked for replication. Each such data item is checked in

turn and the fairness index is calculated for every possible peer which could hold the new replica. Not all peers can hold every replica, due to space and network bandwidth restrictions. The item is then replicated to a peer, with enough space and bandwidth, for which the maximum fairness was achieved. Regarding space limitations information is kept in the **LCM** repository. More details for the reasons of these actions have already been given in section 2.4.1.

Chapter 4

Related Work

During the last few years the P2P model has been extensively used as a model for collaboration of various peers, forming a distributed network. From simple applications, which began from industrial companies and non-profit organizations, to university research. The most common application domain using P2P is that of content sharing.

4.1 Napster

Napster [2] was one of the first content sharing systems, focusing MP3 music files, which brought the revolution of P2P computing. Napster, although a *hybrid-P2P* system, became the most widely used content sharing application. The actual model on which Napster, and several other similar systems, based their operation, uses central servers in order to direct traffic between individual registered users. The central servers maintain directories about the shared files stored on the respective PCs of registered users of the network, and are responsible for answering query requests and handling updates. The fact that makes this system a peer-to-peer one is that all data routes directly from one peer to another without implication of the central servers. The main advantage of this system is its central index which handles queries quickly and efficiently. On the other hand, the performance depends significantly on the number of servers running and the number of users logged on

to each server. Many servers may be overloaded while simultaneously others may be underloaded, leading to bottlenecks and vulnerability to failures.

4.2 Gnutella & Freenet

On the opposite side reside various systems which follow the *pure-P2P* model and address the same issues in a totally different manner. Gnutella [3], the first such *pure-P2P* system, operates in a completely decentralized way. All peers forming the network have exactly the same responsibilities and execute exactly the same algorithms. Every peer logged on to the system is aware of the existence of various other peers, its neighbors, by maintaining a neighbor list. Messages flowing in the system are routed from peer-to-peer until either the destination or a maximum hops-count is reached. A query in Gnutella is considered a push operation, since the source peer floods the network with messages to all its neighbors. This operation continues recursively until a maximum TTL hops value is reached, and the results are transmitted backwards.

Due to its decentralized architecture Gnutella can scale better than Napster, but at the same time various limitations argue about the efficiency of its design. Gnutella's query mechanism is bounded by the TTL value used, thus exposing a certain locality. All queries reach a maximum network distance from the originating node, equal to the TTL value used. Furthermore, query flooding is quite expensive regarding the fact that in P2P systems querying is the most frequent operation.

Freenet [4] is a P2P system which behaves more like a distributed file system. The basic ideas are similar to the ones used in Gnutella, but Freenet has more sophisticated algorithms regarding querying of information as well as replication. Instead of just routing messages to neighbor nodes, Freenet uses content based routing; each shared data object is assigned a unique ID, obtained by some hash function, and this ID is used in order to route requests from peer to peer. As the system operates, the routing tables of the peers are updated in order to reflect the last known routes to data objects. Furthermore while a data object is transmitted to a peer, all intermediate nodes

store replicas of this object locally. Finally, objects stored on Freenet nodes contain no information about the original owners of the objects, guaranteeing anonymity, a key design issue in Freenet's development.

4.3 Pastry, Tapestry & CAN

Tapestry [5] is one of the second generation systems. Actually, Tapestry is a wide-area location and routing infrastructure, on which P2P systems can be build. Tapestry has an explicit notion of locality, providing location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized services. Location and routing are not handled separately, but rather by an integrated mechanism, thus meeting the goals for wide-scale resilience. Tapestry's inspiration is Plaxton's location and routing mechanism (introduced in [6]), which is extended in order to overcome some problems, like scalability and single points of failure.

Pastry[7] resembles Tapestry, since it is also a location and routing infrastructure. Much like Tapestry, Pastry routes requests for objects to neighbor peers until the closest copy is discovered. Each peer has a unique identifier and its routing table points to peers which have similar unique identifiers (common prefix). Similar to Tapestry, the routing tables of Pastry are not growing proportionally to the system's size, thus providing scalability. A query for an object requires logarithmic complexity on the number of hops in order to reach its destination. Furthermore, Pastry tries to adapt its operations with a locality metric, usually an IP routing hops count, and create appropriate routing tables, thus minimizing the distance between neighbor peers.

CAN [8] resembles much to Tapestry and Pastry because it is also a routing and location layer. The main difference from the above two systems is that CAN does not base its routing infrastructure on Plaxton's algorithm or similar mechanisms. CAN uses a virtual d-dimensional Cartesian coordinate space on a d-torus, and dynamically assigns each node to a unique point. Each node routes messages to the best possible choice from the direct

neighbors in the d-dimensional space. CAN achieves scalability, much like the above systems, by maintaining a routing table whose size does not grow proportionally to the number of nodes. Furthermore CAN uses caching and replication of key pairs in order to balance the load of certain nodes which possess more popular keys than others.

4.4 Load balancing

The concept of load balancing is an open problem, at least in the case of P2P systems. Tapestry and Pastry use cryptographic hash functions in order to distribute the various keys inside their name-spaces as equally as possible. CAN gets a little further and uses caching and replication to locally load-balance the requests hitting a node with its neighbors. The problem of load balancing, however, requires more than a fair distribution of objects since the dynamic nature of these systems results in constant changes of popularities. For this reason, continuous load balancing actions are essential in keeping the load distribution as fair as possible. An approach which actually triggered this work, for maintaining a P2P system in load balanced state, can be seen in [9].

On the other hand, the dynamic load balancing problem has been extensively studied in the field of parallel processing and generally in distributed networks. The diffusion method which we use for inter-cluster load balancing was first discussed from Cybenco[10] and further extended from various authors. Various optimality results for diffusion can be found in [11, 12]. Furthermore the work done by Karagiorgos and Missirlis[13] discussed about the number of iterations and the rate of convergence of local diffusion algorithms on different topologies.

Chapter 5

Conclusions & Future Work

The Peer-to-Peer model has recently emerged as a new way for deployment of distributed systems. Currently, work and research are mainly directed to the scalability of these systems as well as searching and locating shared objects.

In this work we examined whether load balancing can be provided by such large systems, and consequently whether the benefits are worth more than the drawbacks. Furthermore we investigated how replication of data and meta-data can be of aid, especially in achieving short query response times. The use of redundancy is no longer prohibited due to today's computing environments which are extremely capable of high-speed processing and network transmition. Moore's law surely gives a new essence in the design of future distributed computing.

Load balancing has been extensively studied in the context of parallel processors and various ideas seem to be also applicable in distributed systems. However, the biggest problem in achieving load balancing seems to be the minimization of the constraints apposed by the granularity of the load. Depending on the coarseness of the load granularity, the maximum balance that can be achieved may vary. We intend to investigate in our future work, the possibility of dividing large items into smaller ones, thus replicating only fractions of the original items. It is likely that this granularity adjustment will maximize the possible balance that can be achieved.

Until now clustering is performed with no direct relationship between

peers. A possible extension is to provide some locality inside the clusters, meaning that all peers have a minimum distance between them. Since our network is built on top of TCP/IP, a distance could be related to a ping response time. This approach would create clusters which are formed from peers which lie in the same geographical region, and therefore execute faster intra-cluster actions.

Furthermore, until now we have paid little attention to the security of our P2P system. We intend to work more on this subject by examining possible behaviors under attacks as well as Byzantine failures. Our future efforts will also try to develop various privacy related mechanisms, an extremely important aspect in P2P networks.

Finally, in respect to our implementation, we would like to minimize the possibility of peer fragmentation into multiple networks. Also various aspects of our implementation need further work and enhancement. We intend to keep on developing our system, in the form of free software (GPL-like license), maximizing its performance and scalability.

Appendix A

Software Used

The following software systems or libraries were used to implement our champion application. The development was done in Java, resulting in a cross-platform application.

- Java Development Kit v1.3.1 and v1.4
- Jakarta Log4j logging library
- File System API from Netbeans

The choice of this software was based mostly on the following reasons:

- They are based on the Java programming language, and consequently are portable across all platforms for which a Java runtime environment exists.
- They are distributed under the terms and conditions of free software, either the General Public License or modifications, and therefore their source code is freely available to the public domain.
- They have been severely tested under real conditions and are considered reliable for extensive use.

Bibliography

[1] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. *Proceedings of the 27th VLDB Conference*, 2001. Roma, Italy.

[2] Napster Home Page. http://www.napster.com.

[3] Gnutella Home Page. http://gnutella.wego.com.

[4] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet : A distributed anonymous information storage and retrieval system. In LNCS, editor, *Designing Privacy Enhancing Technologies: International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009, New-York, USA, 2001.

[5] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[6] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in distributed environment. *Proceedings of ACM SPAA*, June 1997.

[7] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.

[8] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. Technical Report TR-00-010, Berkeley, CA, 2000.

[9] P. Triantafillou, C. Xiruhaki, M. Koubarakis, and N. Ntarmos. Efficient content management in highly-dynamic p2p content sharing systems. Internal Report, Technical University of Crete, 2002.

[10] G. Cybenco. Load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989.

[11] Y. F. Hu and R. J. Blake. The optimal property of polynomial based diffusion-like algorithms in dynamic load balancing. *Computational Dynamics*, '98. K. D. Papailiou, D. Tsahalis, J. Périaux, D. Knörzer (Eds), John Wiley & Son, (1998).

[12] Y. F. Hu, R. J. Blake, and D. R. Emerson. An optimal migration algorithm for dynamic load balancing. *Concurrency: Practice and Experience*, 10(6):467–483, 1998.

[13] G. Karagiorgos and N. M. Missirlis. Iterative algorithms for distributed load balancing. *OPODIS*, December 2000.

[14] N. A. Lynch. *Distributed Algorithms*, chapter 14. Data Management Systems. Morgan Kaufmann Publishers, Inc., 1996.

[15] R. K. Jain, Dah-Ming W. Chiu, and W. R. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. Technical Report DEC-TR-301, Digital Computer Comporation, 1984.

[16] A. Corradi, L. Leonardi, and F. Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–??, /1999.

[17] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*, chapter 37. The MIT electrical engineering and computer science series. MIT Press, 1990.