PADI@HOME

A distributed cycle-sharing system over the Internet

1. INTRODUCTION

For many years, there has been a need for more computational power to perform more and more demanding tasks. Single processor machines have been replaced by more advanced multi-processor ones, but there are a few times where this is just not enough, and billion-dollar super computers are not available. With the advent of the Internet, there is a world of machines interconnected, and a potential for using the power of this machines in a distributed fashion. Thus, PADI@HOME is a project aiming to use multiple remote machines to perform resource-intensive tasks over the Internet. The project presented in this paper has the purpose of developing a distributed compiling system, composed of multiple volunteer machines performing the work, coordinators to assemble the work performed by the volunteers and a distributed file repository to store intermediate and final compilation results. The remainder of this article describes the main Peer-to-peer solution to this problem and then simplifies it into two architectures of decreasing complexity - A replicated and a centralized one.

2. PEER-TO-PEER ARCHITECTURE

2.1 Underlying architecture based on Chord

Chord, as described in [3] is a protocol that provides just one functionality: given some key, it finds the node responsible for that key. Besides this, Chord also posesses (rather, imposes) some interesting properties that we found useful for some parts of this project. Features such as fast lookup of keys, node entry and exit (arbitrary) are already provided by Chord, but will be discussed in the next subsections. Finally, we'll discuss some additions to the Chord protocol in order to achieve fault tolerance, replication and better performance.

2.1.1 Identifiers, lookups and load-balancing

As stated above, Chord is used to lookup nodes responsible for keys. In this particular implementation, identifiers for Chord are generated using the SHA-1 cryptographic hash function, that results in 160bit keys. Node identifiers are obtained by hashing their respective IP address, the remaining identifiers for keys are generated by hashing logical names (such as file names). Using this hash function allows for load balancing in respect to the distribution of keys among the nodes, since there is a very high probability hat key identifiers are mapped to nodes in a uniform way.
Lookups are performed using the standard algorithm described in [4].

2.1.2 Node entry and exit

Chord supports the ability to have nodes entering and leaving disorderly. The only assumption is that nodes run the `stabilize()`[4] routine ever so often to detect newly joined nodes and nodes that have left disorderly or died (according to [2], only one node may die until the next stabilization). Also, in case of nodes that want to leave by their own accord, a simple algorithm is explained in subsection 2.1.5.

2.1.3 Fault tolerance

In order to provide some degree of fault tolerance using Chord, one must extend the original protocol to support successor lists. This replaces the single successor pointer in a Chord node with a list of its next r successors. Let F be the number of simultaneous node failures that one wants to tolerate, then $r = F + 1$. According to [2], no two nodes in a Chord ring fail at the same time, or before the system stabilizes, that is, $F = 1$. To achieve some sort of fault tolerance with minimal effort, each Chord node must have a successor list of size $r = 2$. If the successor list only contained the immediate successor of a given node n, in the event of the failure of node n's successor, this node would be unable to find another successor in its list - having to rely on the finger table, which is not necessarily accurate.

2.1.4 Replication

One of the requirements for fault tolerance in this system is the replication of data among various nodes. Replication works in the same fashion as the CFS file system [1]. Supposing there are N nodes in the Chord network, and each node has a successor list of size r, let k be a number such that $k \leq r$. If key is to be stored at server n, that key is also stored on the next k successors of n. An example of this is given in Figure 1.

Given that no two nodes fail at the same time, there is only need to replicate the data to the immediate successor of a given node, that is, $k = 1$. It's important to note that

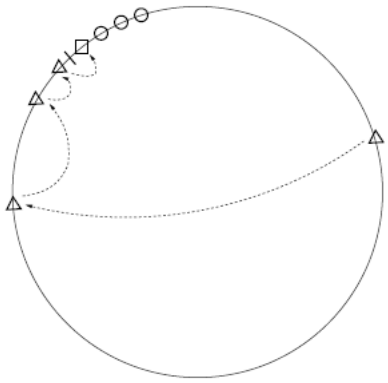

Figure 1: Storage of data replicas. The node in which the data is originally stored is denoted by the square. The circles denote the successors of that node, where the replicas are stored. Triangles are nodes used in the Chord protocol to find the square node.

Chord alone doesn't provide direct support for replication. One needs to build a layer on top of Chord that supplies functions to retrieve and store data on nodes.

void `storeData`(*data*)
data `retrieveData`(*data_key*)

`storeData` uses the Chord protocol to find the node n that will be responsible for holding the *data*. After retrieving this node, it retrieves its successor list and inserts the *data* into n and the first successor in the list.

2.1.5 Chord protocol optimizations

In order to improve the performance of the underlying Chord protocol, some optimizations were made. The first, described in [4], provides a faster way to handle an event that causes a change in the predecessor of some node n. When the predecessor of n changes to some node p', n notifies its previous predecessor p to set its successor to p'. This is faster than waiting for the next stabilization cycle to update p's successor.
Another optimization is related with voluntary node departures from a Chord network. When a node n wants to leave the network, it should transfer its keys to its successor beforehand. After that, n notifies both its predecessor p and successor s about its departure. Node p removes n from the successor list and adds the last successor of n to its own list. Similarly, s updates its predecessor with n's predecessor. It's node n's responsability to send this information to p and s. As is the case with the previous optimization, this allows for a faster update of the Chord cycle, rather thar relying on the slower stabilization process.

2.2 Clients

The client application receives the input from the user. For each file that composes a specific project, the client will obtain each file's unique identifier (UID) from the name server and store it in the repository. After storing all the project's files, the client submits the Makefile to a coordinator, who will be responsible for compiling the project.
If compiled successfully, the client will receive a notification from a coordinator and will then obtain the output files from the repository. If the client can't find at least one output file (because of a repository's node failure, for instance) the whole process will be cancelled and a failure message will be shown to the user. Obviously, if the compilation process fails, a coordinator will notify the client, who will then abort.
Note that the clients are themselves part of the repository (section 2.4), as well as the volunteers. The client application provides the following interface:

void `notifyCompletion()`

`notifyCompletion` is called by a Coordinator when the compilation process ends.

2.3 Volunteers

The volunteers are responsible for completing tasks assigned by the coordinators. When a volunteer node starts executing, it will register itself under a coordinator. Also, if a volunteer willingly leaves the network, it will notify the respective coordinator.
When a new task arrives, the volunteer launches a new thread to accomplish the task, so that it can still do other work related to the Chord protocol (e.g. lookups). After finishing the task, the volunteer will submit the output files to the repository. Obviously, the volunteer has to obtain the files' respective UIDs, determine where to store them and send them to the respective repository node. The volunteer will then notify the coordinator that the task is completed.
The volunteer application provides the following interfaces:

void `ping()`
void `executeTask`(*task*)

`ping` is called by the volunteer's coordinator in order to determine if the volunteer is still executing.
`executeTask` is a service for assigning tasks to volunteers. Note that tasks are Makefile commands that can be executed in a shell.

2.4 File Repository

The file repository has the single purpose of offering a file storage and retrieval service. Therefore its interface is very simple: it consists of a function that stores a file given its UID and a function that retrieves a file given its UID.

void `putFile`(UID, $file\_data$)
$file\_data$ `getFile`(UID)

On the P2P architecture the Repository is a Chord ring where the nodes are the Clients and Volunteers and the keys are the UIDs of the files to be stored in the repository. However this Chord network does not replicate data, as specified in [2].

2.5 Name Server

The Name Server supplies UIDs for every file name in the shared file repository. A different UID is supplied for each file and version. Same files (files with the same name) with different version numbers have different UIDs. Name servers are organized in a Chord ring.

$$\{version, UID\}\ \texttt{getUID}(file\_name)$$

Each Name Server stores its data in the layout presented in Table 1.

name	UID	version
client1/hello.c	1	1
client2/hello.c	2	1
client1/hello.c	3	2
client1/foo.c	4	1

Table 1: File name entries in a name server.

When a client is fetching some file F from the shared repository, it first has to ask a name server for its UID. This lookup done by turning the file name into a key with the hashing mechanism presented in section 2.1.1 and performing a search for the name server which is responsible for that key in the Chord ring. If the newly found name server doesn't already have an entry for the file name we're looking for, it creates this entry and a new UID with version 1. Otherwise, the name server always supplies the UID of the most recent version of the requested file along with the version number. The version number serves the single purpose of making file retrieval more efficient. The client compares the supplied version number with the version number of its local copy of the same file. If both file versions are the same, there is no need to fetch the file from the repository, since the client has the most recent version already.

2.6 Coordinators

The coordinators are responsible for processing the requests from the clients and also for delegating work on the volunteers. The two relevant problems concerning the coordinators are: joining/leaving the network and Makefile replication. Work delegation and load balancing are implementation details, which will be briefly explained further. The coordinators are organized in a simple Chord ring. The following interface is provided by this component:

$$\begin{gathered} void\ \texttt{compile}(Makefile) \\ void\ \texttt{executeTask}(task) \\ void\ \texttt{replicate}(Makefile) \\ void\ \texttt{register}() \\ void\ \texttt{unregister}() \\ void\ \texttt{notify}() \end{gathered}$$

`compile` is the interface clients use to submit their projects. `executeTask` provides a service for other coordinators to assign work to this coordinator. `replicate` is used to replicate Makefiles among coordinators. `register` and `unregister` allow for volunteers to associate themselves with coordinators or leave. `notify` allows volunteers to notify coordinators of task completion.

One of the requirements concerning the coordinators, is that Makefiles can't be lost. We can achieve this by making each coordinator replicate his Makefiles. This replication mechanism is somehow inspired on the Chord protocol, but it is simpler. As stated, the coordinators are organized in a Chord ring. Therefore each coordinator knows who is its successor and its predecessor. When a coordinator receives a Makefile from a client, it registers that itself is the proprietary of the original Makefile and replicates it to its successor, who will register that itself is the proprietary of the replicated Makefile. So there is always an original and a replicated version of the Makefile. Figure 2 shows how the

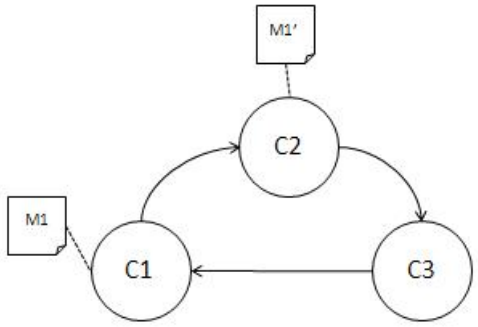

Figure 2: Replication of makefiles between Coordinators.

replication takes place. C_1, the owner of the original version of Makefile M_1, sends a replica M_1' to C_2.The owner of the original version is responsible for processing it. However, the interesting case is when a coordinator fails. Since we only have one replicated version, once a coordinator fails, there will only be the replicated version left. Thus we need to replicate them again, before the system stabilizes. The following example explains how this replication process works. Assume there are three coordinators, C_1 to C_3, organized in a Chord ring. At start, there are no Makefiles. Now let's suppose that each coordinator receives one Makefile from a client - M_1 to M_3 are sent to the respective coordinators. Now each coordinator will send a replicated version of its original Makefile to its successor. Thus, C_1 sends M_1 to C_2 and receives M_3 from C_3.

Table 2 shows the configuration of the coordinator system

	C_1	C_2	C_3
M_1	original	replicated	
M_2		original	replicated
M_3	replicated		original

Table 2: Makefile replication information in Coordinators

after replication. Now let's suppose C_2 crashes. During the stabilization process, we have to guarantee that there are two versions (original and replicated) of each Makefile that C_2 knew about. As the ring stabilizes, C_1 will be informed that C_2 failed. Since C_2 owned all the replicas of the original Makefiles of C_1, C_1 will have to replicate them to its new successor, C_3. Likewise, C_3 will be informed that C_2 failed. Since C_2 owned all the original versions of C_3's repli-

cas, C_3 will now assume its replicas are original versions and will replicate them to its successor, C_1. Note that C_3 now owns new original versions, which it will assume as work to be done. This replication model guarantees that there are always two copies of each Makefile on the network, that can move along the ring.

The following example shows how the whole compilation process takes place, from the point of view of the coordinators. Consider the three coordinators in the previous example. A client sends a Makefile (project) to C_1. C_1 replicates the Makefile to C_2, according to our presented solution. C_1 then determines that this project can be compiled by executing five parallel tasks. Three of these tasks are assigned to the volunteers that C_1 knows. The other two tasks are assigned to its successor, C_2, who will assign them to the respective volunteers. Now there are several options: a) nothing fails; b) a volunteer fails; c) a coordinator fails.
a) If nothing fails, every volunteer will notify the respective coordinator that the task is finished. Each auxiliar coordinator (in this example, C_2) will then notify the coordinator who started the process (in this example, C_1) that the task is finished. Finally, C_1 will notify the client that the project is compiled.
b) If a volunteer fails, the coordinator needs to reassign all the pending tasks to another volunteer/coordinator. The coordinator constantly pings the volunteers it knows, in order to check on wether or not the volunteer is executing. If it is not, the coordinator will reassign all the pending tasks. This ping protocol also allows the coordinators to manage their respective volunteers.
c) If an auxiliar coordinator (in this example, C_2) fails, C_1 will know about this event and will reassign the tasks to another coordinator (C_3). If C_1 fails, C_2 will know about this event. Since C_2 has received the Makefile from C_1 and knows that C_1 failed, C_2 will now replicate the Makefile to its successor (C_3) and will also restart the whole project compilation. In a more advanced solution, C_2 could detect what parts of the project have already been compiled and thus resume the process instead of restarting it.

3. REPLICATED ARCHITECTURE

The replicated architecture is a simplified version of the final peer-to-peer architecture. The Repository consists of a single, centralized entity. Thus the client and volunteers are not organized in a Chord network. However, the Chord infrastructure is already present in this version, so that the name servers provide the desirable services (as described above, in the peer-to-peer architecture).

3.1 Centralized repository

The centralized repository is a single centralized entity that provides the same services as the fully distributed repository, as mentioned in section 2.4. Therefore it has the same interface. The difference is that it isn't supported by an underlying Chord protocol and that we assume no repository failures at this point.

3.1.1 Concurrency on file access

Since work on the various files involved in the compilation process is done by one volunteer at a time, we assume no two volunteers are going to write on the same file (though more than two can read from the same file, and reads are always consistent because no other component writes on the file afterwards). In this case, an optimistic concurrency mechanism is appropriate.

3.2 Coordinators

In this architecture, there are multiple Coordinators, working in the same fashion as in the peer-to-peer architecture (section 2.6. Coordinators are arranged in a ring topology - much similar to a Chord network - to maintain communicability between them without having all Coordinators know each-other. The only difference is that since no Coordinator failures are to be tolerated, there is no need for replication of Makefiles among them, and only one successor pointer is needed for each Coordinator, instead of a successor list.

4. CENTRALIZED ARCHITECTURE

The centralized architecture corresponds to the first version of the PADI@HOME system. On this approach, there is no need for a Chord protocol infrastructure, because there is only a single name server and a centralized repository. Also, because it's assumed that there are no points of failure, there is no need for any fault tolerance. A Client connects directly to the single Coordinator, and the Coordinator has multiple Volunteers to assign work to. Because there is only one Name Server and Repository, there are no problems in maintaining coherence in file UIDs and version numbers.

5. CONCLUSIONS

In this article, we presented a solution for PADI@HOME, with particular emphasis on a Peer-to-Peer approach. The other presented architectures are mere simplifications of the P2P approach. Thus their respective descriptions were not our top priority. Nevertheless these two simplifications must be working properly in order to complete the final, fully distributed architecture
There was given some thought on writing a solution that was rather efficient and economic in memory space, even though it's not always possible to have both these properties. One such example is the replicated architecture. We could have made every server know one another, which would probably have held much better lookup times, but that would lead to issues regarding memory usage, so the Chord-like topology and principles of operation were used even in this version. Finally, we believe there are still many ways in which it could be improved, and we expect to do so in the near future.

6. REFERENCES

[1] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. 2001.

[2] L. Rodrigues, J. Garcia, and L. Veiga. PADI@HOME: Platforms for Distributed Internet Applications Project - 2008/09. 2009.

[3] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *Proceedings of the 2001 SIGCOMM conference*, 31(4):149–160, 2001.

[4] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. 2003.