PADI-FS

Alexandre Henriques
66935
alexandre.henriques@ist.utl.pt
Group 17

Daniel Cardoso
66964
daniel.cardoso@ist.utl.pt
Group 17

Francisco Raposo
66986
francisco.afonso.raposo@ist.utl.pt
Group 17

Abstract

PADI-FS is a distributed file-system to manage sets of files, meaning that it allows access to files from various hosts via a computer network. This makes it possible for multiple users on multiple machines to share files and storage resources. Our implementation includes facilities for replication and fault tolerance, which means that when a limited number of components of our file-system go offline, it keeps working without any data loss.

1. Introduction

Our system has three main components: the Metadata Servers, the Data Servers, and the Clients. The Metadata servers maintain relevant information about each file in the system, such as which data servers store each file, and how many times a file is replicated in different data servers. The Data Servers store the content of files. The Clients can make requests to create, delete, open, close, read and write files in the file-system. There is a fourth component, the Puppet Master, who acts like a centralized controller that can issue commands to other components and is used for testing and debugging purposes. We wont be focusing on it in this paper due to it being irrelevant to the system itself. In the rest of the paper well explain the system architecture and how the components interact between them. Well also explain the implementation and the mechanisms found to address our concerns.

2. Proposed Solution

2.1. Meta-Data Servers behavior

2.1.1 MDS Synchronization

The handling of entry and exit of MDS is constrained by how we define their synchronization. We came up with 2 alternatives for it: 1) either all MDS operate the same way or 2) we set one of them to be an active replica (the one which receives the clients' commands) that is responsible for almost all the processing and the others (passive replicas) merely keep a copy of the state. The option 2) is by forcing the clients to contact the MDS in order of their IDs because the MDS with the lowest ID that is online is always the active replica. Both alternatives have disadvantages. 1) Would require a much more sophisticated (and more difficult to design and implement) synchronization technique between all servers (e.g. to check whether or not a client can delete a file we must check if no MDS has recently received an open command). 2) Could create a bottleneck in the system because all commands would be sent or redirected to one server. If a server goes down in 1), nothing special needs to happen: the server merely asks for the current state of any other server as soon as it recovers. However, in 2) we need to decide which of the remaining MDS is going to be the new active replica (assuming the old active replica is the one that crashed). To solve that issue we simply say that the online MDS with lowest ID (IDs range from 0 to 2) is always active replica. When the old replica server recovers it lets the other replicas know it is now online and if its ID is lower than the others it becomes the active replica. Assume the following scenario: MDS 1 and MDS 2 are passive replicas and MDS 0 is the active replica. MDS 0 crashes and MDS 1 receives a command. Because MDS 1 received a command from the Client he knows he is now the active server. When MD0 recovers it will tell the others MDS he is alive and by knowing that his ID is lower than everyone else's he becomes the active replica. The other two MDS (including the current active replica) become passive replicas because they get the Im Alive from a lower ID MDS. We thought of going ahead with option 2) as it seems way simpler than option 1). Therefore, the following questions need to be answered: How to keep passive replicas up to date? How to recover state after crashing? To keep passive replicas up to date, the active server sends them special commands to update their state when execut-

ing a client command (e.g. create). Its important to notice that the passive replicas can also communicate with clients, though they are not the ones handling the client commands themselves, since the client doesnt know which replica is active, but by the order he calls the MDS he will always communicate with the active MDS. When a MDS is recovering it will ask the MDS (again in order of IDs) for the current state. The MDS that receives this command will send the data and know if it needs to become passive or not.

2.1.2 Commands

To handle the Create command the server just has to check if that file exists in his local storage and if it does, nothing is done, if it doesnt then the file is created. The MDS then communicates with the DS to let them know they have to create the file. If the Data Servera are offline the MDS will keep sending the create command from time to time. To deal with the Close command the server checks if that client has the file open and then it goes the same way as the create command: if the file is not open it does nothing, if it is open he closes it and lets the other two know. We check if that client has the file open so that a client cant send many close commands and close files other clients were using allowing the delete command to be performed. The other two possible commands are Delete and Open. When a Delete command is received, assuming the file exists, the server checks if the file is open by any client, if its not it deletes the file and lets the other two MDS know. The deletion is communicated, as in the Create command, so the DS doesnt have garbage in its state that would later ruin the Load Balancing and Migration algorithms. The Open command just checks if the file exists, if so it saves some information that identifies that that client has the file open and gives the information needed to the client. In the end of any of these commands the server saves in the disk the information so it is not lost in case of a failure.

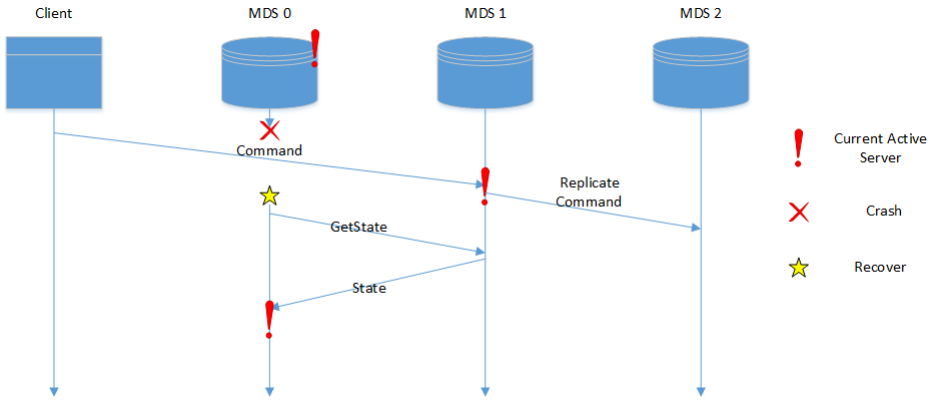

Figure 1. crash situation described in section 2.

2.2. Data Servers behavior

When a DS is created it is empty, so the only thing that needs to be done is to register the server in the MDS. When a DS recovers nothing is done regarding the Read and Write commands issued to it while it was offline. However it will eventually receive pending Delete and Create commands by the active MDS. This works because the quorums are chosen to handle the out of date information. The next Write commands received will update any old versions of the files.

2.3. Load Balancing and Migration strategies

We are going to have two algorithms performing load balance. One of them is executed when creating a file and the other is executed manually. When a MDS receives a Create command it checks the load of every DS by checking the statistics it keeps by asking for them from time to time. It then selects the DS with less activity for the file being created. The activity in a Data Server is calculated by the number of writes, reads and the size of those operations. The increment of the load is calculated in every operation with formulas like this:

WRITE_CONTRIBUTION + NUM_BYTES *

SIZE_OPERATION_CONTRIBUTION

In this case it is a write operation. The contribution of a write, read and the size of that operation can be set in the DS.

The second algorithm runs manually to perform migration of files. First, the primary MDS asks all the Data Servers for their statistics (loads). All Data Servers that are not online or do not respond before the timeout triggers will not be considered for the next step. Then, an algorithm is used to calculate the migrations (only considering the previous Data Servers). After the migrations are calculated, the MDS will send the DS migration commands (asynchronously) to execute. The algorithm to calculate the migrations is parameterized by a threshold (float value in the range [0,1]). It will use that threshold to decide if it pays off performing some migrations or not. The input is a collection of DSLoads. A DSLoad is a structure containing the corresponding DS total load, the DS id and a collection of FileLoads. A FileLoad contains the file id and the file load. Note that the DS total load (considered here) is not always the sum of all its FileLoads because open files are not considered in this calculation. This pseudo-code illustrates the algorithm:

```
CalculateMigrations(loads, threshold) {
        list<Migration> migrations;
        while(loads.count > 1) {
                sort(loads);
                biggestLoad = loads[end];
                if(biggestLoad.loads.count < 2) {
                        loads.remove(biggestLoad);
                        continue;
                }
                sort(biggestLoad.loads);
                foreach(DSLoad load in loads) { //except for the biggest (in increasing order)
                        searchForValidMigration(biggestLoad, load); //will pick the highest fileLoad
                        if(validMigration)                          //and the smallest dsLoad possible
                                break;
                }
                if(!validMigration) {
                        loads.remove(biggestLoad);
                        continue;
                }
                oldDistance = abs(biggestLoad.load - dsLoad.load);
                newDistance = abs((biggestLoad.load - fileLoad.load) - (dsLoad.load + fileLoad.load));
                if(newDistance >= oldDistance)
                        biggestLoad.loads.remove(fileLoad);
                else {
                        oldRatio = dsLoad.load / biggestLoad.load;
                        min = min(biggestLoad.load - fileLoad.load, dsLoad.load + fileLoad.load);
                        max = max(biggestLoad.load - fileLoad.load, dsLoad.load + fileLoad.load);
                        newRatio = min / max;
                        if(abs(newRatio - oldRatio) > theshold) {
                                biggestLoad.loads.remove(fileLoad);
                                biggestLoad.load -= fileLoad.load;
                                dsLoad.loads.add(fileLoad);
                                dsLoad.load += fileLoad.load;
                                migrations.add(new Migration(biggestLoad.id, dsLoad.id, fileLoad.id));
                        }
                        else
                                biggestLoad.loads.remove(fileLoad);
                }
        }
}
```

A valid migration is when the file being migrated does not yet exist in the destination DS (i.e. the destination DS is not a replica of the source DS for that specific file). The function searchForValidMigration will find the best valid migration for the selected biggest load DS (the best migration is the biggest load file being migrated to the smallest load DS). The result of this algorithm is processed by another algorithm that optimizes the migrations by removing multiple migrations of the same file (replica) if they exist. For example, consider the case where after the calculations are done there is a file which is supposed to migrate from DS1 to DS3 and then from DS3 to DS4. Both these migrations would be deleted and another (from DS1 to DS4) is added.

2.4. Read and Write synchronization solutions

To handle the commands that the DS may receive we need to consider that the client has the information about all the DS that contain that file. Here we thought of two options to deal with the problem of synchronization: 1) having a primary DS for each file (to synchronize all DS when performing read/write commands on the files) or 2) having all synchronization done in the client side. Option 1) was based on the algorithm used by the Google File System as seen in [1]. In either case, when executing a write command the client sends the data that needs to be written to every DS returned by the MDS. From this point the two algorithms differ: 1) the client sends the write command to an active DS that is known by the MDS. When the active server receives this command it checks with the other servers to see if they received all the information correctly and then confirms the write. All the servers will reply, to the active, their state related to that file and the active server will send the quorum answer to the client. 2) The client sends the write to all of them and it waits for the confirmation from each server. In this case the client has to deal with the quorum and decide what action he has to take. When executing a read command, we also have the options of the quorums being dealt with in the client or in the active server. In 1) the client only communicates with the active server for that file and that one will send and receive the read command from the others. It then sends the result to the client. In 2) the client sends the read command to all the DS that contain that file and waits for their. Depending on the type of read the client will decide what action he will take from the replies he received. The advantages of 2) are that the DS are dedicated to storing the data and are more efficient in that matter. On the other hand if we move that functionality to the DS, the client becomes a much more simple application that only communicates with a MDS and one DS. However we would have to consider the case when the active DS crashes (and have some protocol defined for a passive DS to take over as active). As we are supposed to tolerate crash faults on any DS we used option 2) as we didnt want to rely too much on a single (the active) DS. To deal with concurrent read and write operations we defined the version of a file as a pair composed by a value, the version per se and a time stamp. When a DS receives two commands with the same version value he will only write the version with the later timestamp.

3. Conclusion

In this paper we have explained the PADI-FS and our solutions to the problems we had while developing it. The first subsection of the Proposed Solution, explains the different alternatives to handle the entry and exit of MDs, later we explained how to keep passive replicas up to date and how to recover state after crashing. In the second subsection we talked about how DS recover from a crash and how they communicate with the MDS. In the last two subsections we explained the data placement algorithms regarding the selection of DSs to hold the contents of files, and the coordination of read and write operations performed with the DS respectively. Overall we think we have presented a complete (but not the only) solution that solves the most relevant issues in PADI-FS.

4. References

[1] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. Spring 2011. http://www.cs.umd.edu/class/spring2011/cmsc818k/Lectures/gfshdfs.pdf