P2P Distributed File System with an API for communicating with a Map/Reduce Framework

R.A.I.D.F.S.

Members:

Jérémy Gotteland, Sven Reber, Pascal Cudré, David Froelicher, Alban Marguet, Valérian Pittet.

Abstract:

Datacenters require huge investments, first to build it, and then powering and cooling the computers require a lot of energy, which ends up having significant costs and being bad for the environment.

But with the rise of the Big Data hype, demand is growing quickly and big companies like IBM are investing massively to build the most performant and scalable data-centers.

On the other hand, nowadays, almost everyone owns a personal computer, and every company provides a computer on their employee's desktop. The computing power and storage of those machines is though never fully used, which is a regrettable waste.

Goal:

We intend to design a P2P protocol that we will implement in the form of a desktop application (possible mobile in the future). Based on the Distributed File System that we will have created, we will define a Map-Reduce API to be able to implement a Map-Reduce application on our network.

One possible outcome of this project would be to convince entreprises to deploy our application on their pool of computers, so that the idle computing power and the storage of those computers can be used to perform our or their computations.

The experience

In this project, even if the team cooperation was really good, the processing was really tedious. We chose to use the JXTA framework to develop our communications between Peers, however, it took us a long time to interpret the documentation and to assimilate it. Although the beginning was more or less slow, we succeeded in catching up our lateness and got a working project for the deadline.

How it works:

You can get the code at https://github.com/MGrin/p2p-mapreduce .

First, wecreated a class named Mishell which implements a Command Line Interpreter, and is designed especially for our DFS, (All the local changes - e.g. index of the present files and the files themselves on the OS - are done by the class Metadata). The commands are the following:

(you can also refer to the manual file, also on the wiki to better know how to use RAIDFS through Mishell)

- **Connect :** The first command the user has to enter, it allows him to connect to the network, ask one of his neighbour (peer already connected) for his “meta.xml” which is the file containing the index.

  This file is automatically updated locally for each connected user, such that when the user asks his neighbour, he’s getting an up-to-date index file.

- **Put :** When a user wants to add a file on the DFS, he enters :
  “put absolute-path-of-the-local-file-to-add absolute-path-wanted-on-the-DFS”

  This command updates the meta.xml (index) file locally,and then sends a “Put Advertisement” on the network which will be discovered by every peer connected to the network.
  This way everyone keeps his local index updated.

- **Rm :** Remove command. When a user wants to delete a file from the DFS, he enters:
  “rm absolute-path-of-the-file-in-the-dfs”

  This will remove the corresponding line in meta.xml (locally) and in the same way that we did for the “put”, it sends an advertisement to warn everyone that this file is deleted. Every peer that receives this advertisement will delete all the chunks of this file.

- **Get :** This command permits a user to get a file present on RAIDFS.

  A user must enter the following command:

  “get absolute-DFS-filename wanted_name”

  The peer is going to send GetChunk messages to his neighbours for each chunk he does not own. It knows which of his neighbours owns which file because of the GlobalChunkfield it is building periodically (see below).

  Then the peer assembles the chunks to get back the original file.

- **Ls :** A user must enter the following command:

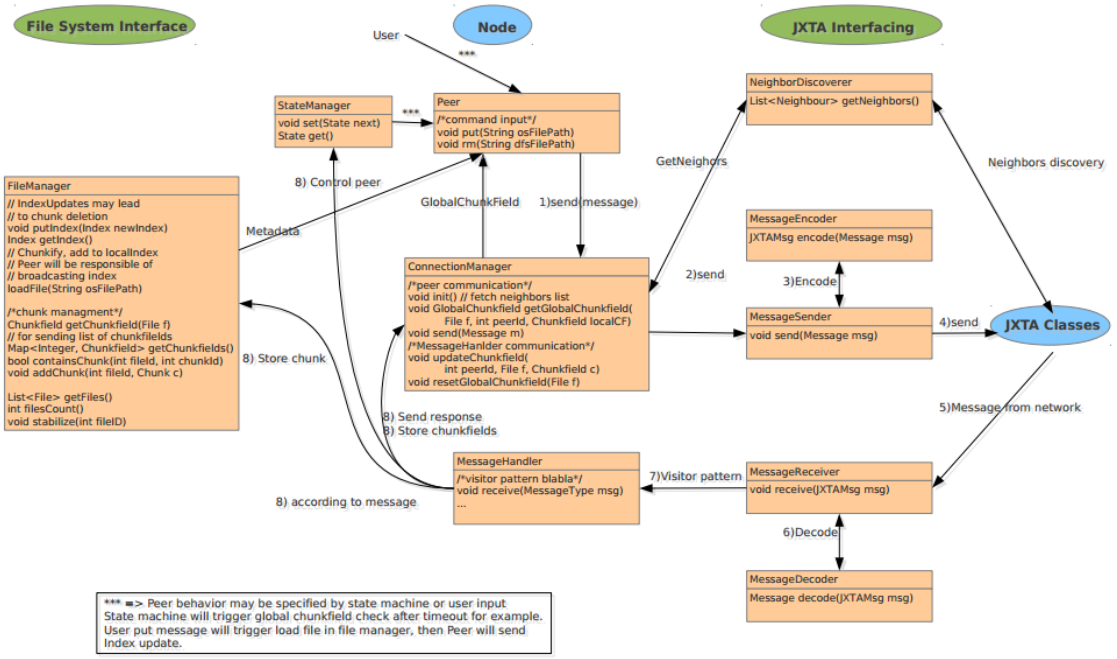
  “ls absolute-path-of-the-folder-OR-nothing”

  By looking in his local meta.xml the user can know what files and what folders are actually available in the DFS.

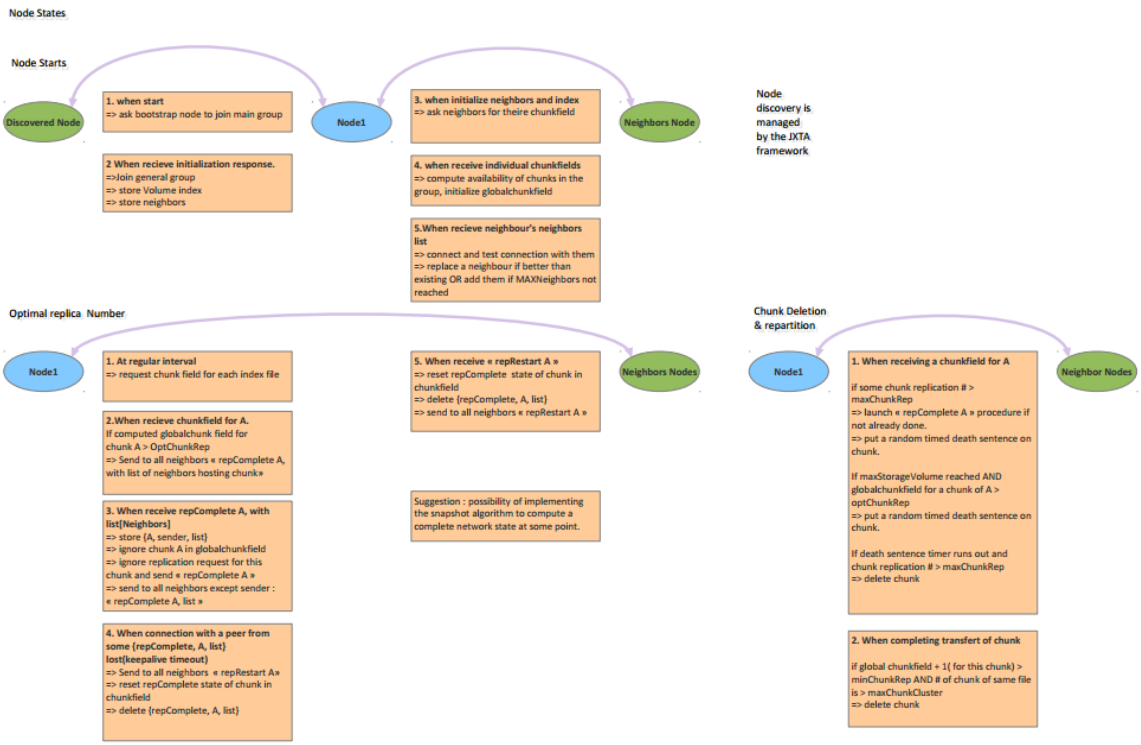
- **Quit :** To disconnect himself from the DFS network and close the program.

Peers, connection and implementation

General structure:



Behaviour of the peer on replication and deletion of chunks :



If the peer successfuly connects to the “tracker”, it opens a IncomingPipe and publishes an associated “PipeAdvertisement” on the network, so that other peers discover it and exchange message with it.

It then fetches a fixed number of neighbours by discovering PipeAdvertisements on the network, and tries to get the current Index by sending GetIndex messages to them.

When a SendIndex message is received, the peer knows the current state of the DFS and can start discovering index updates (via Advertisement discovery), and putting/removing files from the DFS. Currently, the Index Updates are discovered every 20 seconds.

On discovery of a PutAdvertisement, the file gets added to the XML Index file, and a folder corresponding to this file will be created in the *.raidfs_data/* hidden folder, in which we will store the chunks for this file.

It is also added to the internal file abstraction, which allows our peer to query other peers for their chunkfield for this file and build the GlobalChunkfield.

The GlobalChunkfield is a **map** that links, for every file, the number of times each of its chunks are owned by our neighbours. It helps us maintaining an optimal replication state.

We use it to query chunks if their number of duplications is inferior to the optimal replication number of the system (fixed heuristic constant), or to drop chunks if the number of replications is too high (not implemented yet).

In the beginning a file is considered “Unstabilized”, but when the optimal replication number is reached, a “FileStabilizedAdvertisement” is published on the network and peers know they have to stop querying for chunks for this file.

On receival of a RemoveAdvertisement, the file gets removed of the local XML Index file and from the internal file abstraction. Requests from other peers concerning this file will be completely ignored.

API Map-Reduce

Map reduce flow on the p2p network :

1. An Edge (Peer P) receives a Job, consisting of Job1(JobId, ResourcesNeeded, MapFunction, ReduceFunction)
2. Peer P adds a MapFile M1(JobId, ResourcesNeeded, MapFunction, InitiatorPeer) to the index, with all chunks empty. The content of each chunk of the MapFile can be obtained by applying the Map function of Job1 on the Resources. Each chunk corresponds to the mapping of a chunk of the ressources. Then, these chunks will be further splitted into several parts for each key the mapper discovers (KeyChunks)

3. Peer P advertises that the index has been modified with a new MapFile M1.
4. Neighbors receive the update. They check if they can already get chunks of the MapFile from their neighbors. Otherwise, they check which Resource they have, and choose some randomly to create the chunks (Map).
5. Each time a Mapper finishes mapping a chunk, it sends a ChunkMapped(JobId, ChunkId, List(keys)) msg, with all the keys discovered in the chunk, to the Initiator. The initiator keeps track of the keys, of the mappers and of the finished mappings to facilitate the work of the reducers.
6. For each new key the initiator receives, it creates a ReduceFile R1(JobId, Key, ResourcesNeeded, ReduceFunction, List(Mapper Peers), InitiatorPeer) and signals an index update. The resourcesNeeded are all the KeyChunks corresponding to the Key. So a reducer will need to get the content of the right KeyChunks of every Chunks created during the Map phase before starting.
7. Neighbors that ask for a ReduceFile of a particular key become responsible for the reduction of this key. They have to get the ReduceFile chunks by either grabbing the KeyChunks and Reducing them, or copying them from another Source like another file(preferred way).
8. When a reducer has all the KeyChunks of its Key, it applies the Reduce function on them to fill the ReduceFile. When the reducing is completed, the reducer sends a Message ChunkReduced(JobId, Key) to the initiator so that it can keep track of the progress.
9. The operation is known to be finished when all ReduceFiles are stabilized, or when the initiator has received ChunkReduced msg for all keys in its list, from alive peers.

Conclusion

We managed to perform several tests to make sure the DFS stays stable with a couple files and a small number of peers, but we still have to test it on a large scale to test how resilient the system is to peer deconnection and border cases.

For the future, we have to secure the access to the Distributed File System, and emphasize the organisation of neighbours for a file around the “PeerGroup” abstraction of JXTA, which provides narrower Advertisement publishing/discovery, and would reduce overall overhead.

Map/Reduce integration being the ultimate goal of the project, we showed that the DFS is designed to make that integration easy, which will reduce the amount of work necessary for this part.

We also expect some problems with the JXTA Framework: as it is not maintained anymore, it will be hard debugging if we need to.