Software Engineering Lab [CS3410]

Even, 2012

EDGE NODE FILE SYSTEM

- Final Report

Neerad Kumar G [CS09B008] Shailend Chand R [CS09B041]
Vamsheedar Rao S [CS09B044] Vamsi Krishna D [CS09B006]
Venkatesh G [CS09B032] Uday Chowhan M [CS09B034]

1 Abstract

1.1 Problem Statement

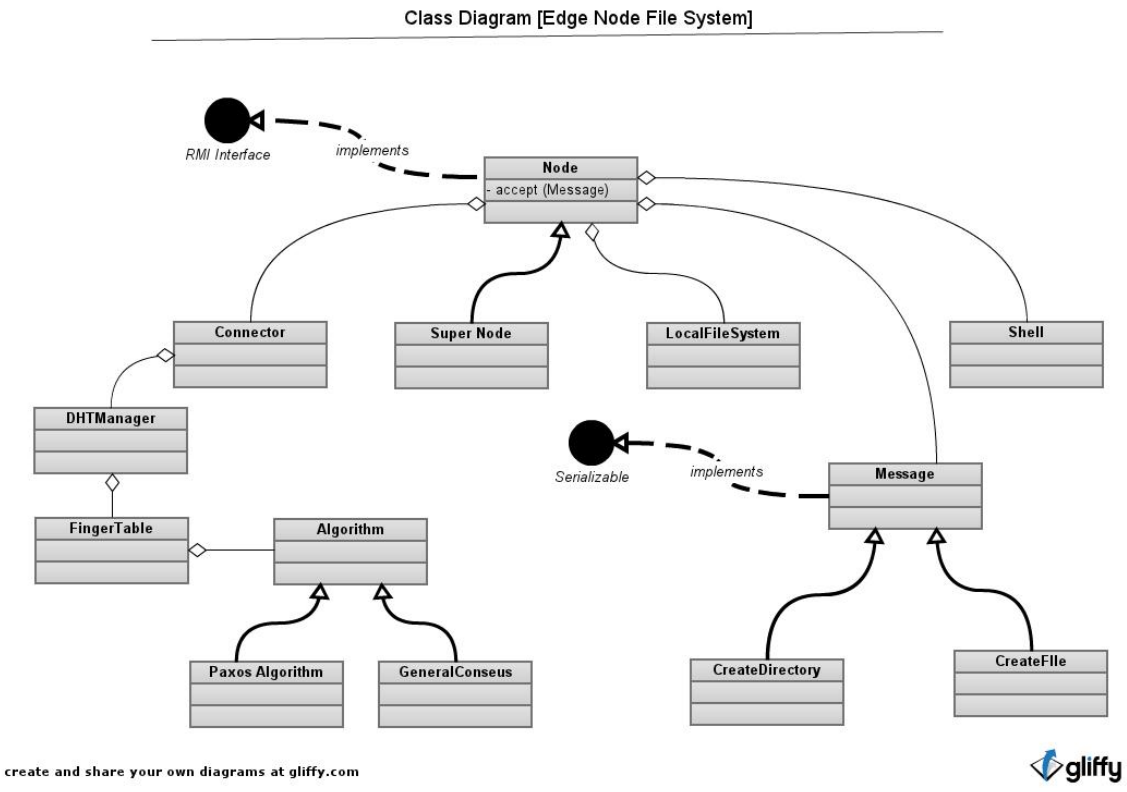
- Refactor the existing code of the implementation into a more readable and manageable set of classes.
- Resolve Load balancing issues
- Implement NAT so that ENFS can be located anywhere on the Internet

1.2 Our work

- We've finished refactoring the whole code. It's now more readable and easily maintainable following several design patterns. The code is now free of hard coded data. The refactoring also resulted in reduction in no. of lines of code. The exact details of the whole implementation is presented in the next section.
- There was a very trivial load balancing in the original codebase and this too was commented out. We now implemented the load balancing as per to the paxos algorithm which was proposed in the ENFS paper and care has been taken that this algorithm is not hard coded. Thus, in future, something else other than Paxos can also be incorporated adhering to an interface.
- Though we've planned to incorporate NAT issues in the original timeline, we couldn't come up with a suitable idea in adding it to ENFS implementation.

2 Implementation

2.1 Class Diagram



2.2 Refactoring

We started refactoring ENFS code even before we had any idea of design patterns, but suprisingly, all the major design choices we chose here are also standard design patterns.

The initial code base consisted of more than 7000 lines of code in a single node class and two other utilities classes none of which were comprehensible at the first read. Several parts of the code is hardcoded to work on the local computers for a single cluster. The node class alone had around 60 functions, many of which were not related to each other, but seemed like a part of the node class as a whole. We refactored the node class into 5 new classes : Connection, DHTManager, FingerTable, Scheduler, LocalFileSystem in addition to the Node class. A *Facade* pattern can be observed here. The interface of the node still remained the same and is refined after the refactoring is done as the initial 60 functions were just private functions used to do various unrelated jobs which are now clubbed into appropriated classes based on their functionality.

We had to throw away several parts of our first design after we came across Java's RMI implementation. In the final design, we came up with this much better solution using RMI rather than writing our own server which would have needed several unit tests. The various design choices which are are substantial in the final design are explained below in detail.

2.2.1 Removal of Request Handler

One of the primary problems in the previous design was how the node listens to various requests and how it responds to it. The original design does this : a server is run to accept the requests which are basically strings requesting a specific operation. These requests ranged from 'a'-'z', 'A'-'G'. This resulted in a 30 case switch statement in the server code. And addition of an new request resulted in modifying the server code which is not desirable. The redundancy here is running a server just to check the type of the request and call a function. So, we have completely removed this part of the code and used the Java's Remote Method Invocation (RMI) API. The RMI allows a node to invoke a function on a remote address space which is exactly what is needed here.

2.2.2 Runtime promotion of Node to Super Node

A normal node, sometimes need to be upgraded to a super node(when there is an excessive load or a super node fails). The original implementation destroyed the current node, and recreated a new node with supernode properties. This is not preferrable, we both waste the resources and also the Quality of Service is compromised. In our approach, we promote Node to a super node at runtime using both inheritance and aggregation to handle the additional objects for the supernode.

2.2.3 Addition of Message class

But the above design choice lead to one more problem, the RMI needs to expose the functions which can be invoked remotely, and this is usually associated with a security risk. So, we have chosen to limit the list of functions to be exposed. This resulted in selecting *Command* pattern and an abstraction of messages. The no. of functions in RMI's interface is reduced to one and it accepts a message object as its parameter (which is transferred over network and taken care by the RMI API). The appropriate function to be called is encapsulated in the message i.e, a polymorphic behavior is seen in this abstraction of messages. But, the messages vary very little in their implementation (only a function to be called on the remote space), so, inheritance might

result in a class explosion if the types of operations that can be performed on node increases. Though we can use a creational pattern like *Prototype* here to tackle this problem, we stick with the inheritance model as of now.

2.2.4 The splitting of Node class

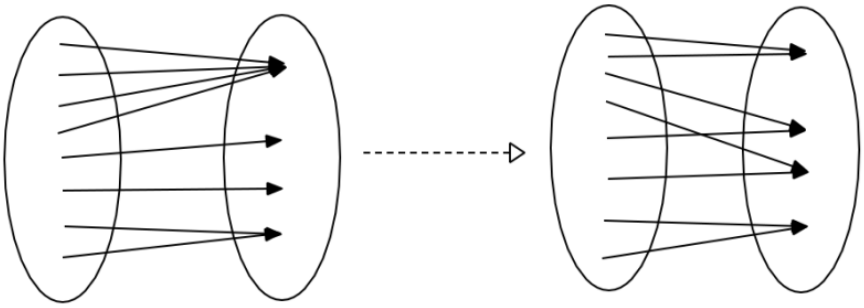
Each node does a plethora of operations like requesting a supernode to storing a block on the file system. Though these make sense under a bigger roof of Node class, this lacks maintainability with such huge number of functions. A thorough analysis of the code gave us the impression that this class can be split without losing the essense of 'node' class. And hence we chose to split the node class into connection, dht, fingertable, filesystem, scheduler (suggesting their obvious use). Connection uses dht manager to find out the details about the peer-to-peer network while dht uses finger tables in its implementation. The fingertable class has one more interesting pattern. The load balancing is supposedly an operation to rearrange the finger table entries but depends on our choice on how to balance them. We simply couldn't find it useful to hardcode `Chord's` implementation into fingertable. Hence, we chose to use the *Strategy* pattern to choose our algorithm to balance the finger tables.

Coming to the file system, this too seemed a good design choice as the metadata and the file system operations like load, store are not related to anything else and can provide a good insight on how we chose to store the local blocks. The scheduler class is another abstraction over different kinds of scheduling opertains like disk scheduling, load balancing, etc.

2.2.5 Addition of an ini styled configuration file

The old design uses a text file with all the configuration parameters. This is one of the reasons for the existence of hard coded ips in node class. Here, we choose to use *ini* configuration files for the job because of their ease of syntax, feature set and availability of libraries to parse them effectively.

2.3 Load balancing



after load balancing

When a query load on a SuperNode increases, the responsibility of Virtual Identifiers of that supernode is transferred to a relatively lightly loaded one.

The ENFS model maintains a large Virtual Identifier space in correspondence to the Physical Identifier Space of the Supernodes. This mapping is stored in supernode_map file, which will be updated accordingly and so as its replicas. Paxos algorithm is used for the implementation of load balancing, in which each SuperNode keeps checking its cpu_load_percentage periodically. If that exceeds a pre-determined threshold (different for each SN), it sends '*prepare*'message to a set of receivers (which have the replica of metadata of client or may be spare SNs) acting as a leader. The receivers which are ready to accept some load send the *promise* message to the leader, which on receiving these promises, if total number promises greater than particular

acceptance_threashold ,it sends '*accept*'message request to those who have promised, else using different prepare number the leader sends the request iteratively. In response the senders of promises gives a '*accepted*'message to the leader. That ends the communication of paxos messages. then in the supernode_map file the new balanced VI space is built and maintained. IpaxosProposal, JpaxosPromise, KpaxosAccept and LpaxosAcceptResponse are the messages corresponding to paxos algorithm. *NodeDetails* file has been given the constants to deal with the required ones. Balancing of supernodes is periodically checked as to make sure no SuperNode is overloaded with queries.

Acknowledgements

- The Edge Node File System: A Distributed file system for high performance computing , Kovendhan Ponnavaikko and Janakiram.
- L. Lamport, The part-time parliament, Trans. Comput. Syst, 16 (1998), pp. 133169.