Submitted by

Vineet Dhanawat

Ishan Parmar

Instructor

Ravi Prakash

Project Documentation

CS 6378: Project II

1. Introduction

* There are n server nodes and m client nodes in the system, numbered from zero to n-1, zero to m-1. Each node executes on a different machine.
* Reliable socket connections (TCP) are established between each pair of server-server nodes and client-server pair of nodes.
* For each object, the HashFunction H(O) returns a server node to perform write or read on.
* If, Client Ci wants to update an object
  + Write is performed at three servers numbered: H(O), H(O)+1 modulo n, and H(O)+2 modulo n
  + Read is performed at any of the three servers numbered: H(O), H(O)+1 modulo n, and H(O)+2 modulo n
* Restrictions:
  + Client should be able to randomly choose any of the three replicas of an object when it wishes to read the value of the object.
  + Client can do update/insert only if two or more servers are available out of the chosen three.
  + Client should abort update/insert in case just one node is available.
  + In case of two or more clients trying to update same object. Updates must be performed in the same order in all servers.
* Few of the nodes in the system can be down or out of reach at times. Such a scenario can be simulated by logical partitioning of nodes into two set of nodes. Nodes in 'up' set of nodes can respond while nodes in the 'down' set of nodes will not respond to requests.

1. Classes

This implementation of the said problem statement uses 2 packages /projects – one for the client and one for the server.

* ClientSrc
  + ClientNode

This class serves as the main class for the ClientSrc package/project and takes 1 argument (its nodeId). It contains information like nodeId, object name, Hashmaps used to store Server sockets, read and write buffers, total number of server nodes and client nodes.

The act of writing of any object is initiated from this class by sending the object to the required servers.

* + DaemonThreadServer

This thread is invoked from the ClientNode class. Its function is to listen for any communication from the server nodes. It is used while reading an object from a server.

* + InputOutputHandler

This class contains 2 methods to read the configuration files containing information of the clients and the servers. It extracts the information of the number of clients and servers from these configuration files. It also stores in Hashmaps, the connection parameters (nodeId, port number, address) of all nodes.

* + SendConnectionThread

This thread is invoked from ClientNode class. Its only functionality is to initiate socket connections from the client to all the servers. For this, it utilizes the Hashmaps created by the InputOutputHandler class.

* ServerSrc
  + ServerNode

This class serves as the main class for the ServerSrc package/project and takes 1 argument (its nodeId). It contains information about the number of clients and servers, Hashmaps to store server/client sockets, read and write buffers and data structures to store the information of writing order of objects. It takes 1 argument (its nodeId). It is responsible for reading the configuration files and initiating all the required threads (mentioned below).

* + DaemonThreadClient

This is the listener thread for all communications to a server from a client. It is started by the ServerNode class. It is used to listen for read/write requests sent by the clients to the server(s).

* + DaemonThreadServer

This is the listener thread for all communications from a server to a server. It is started by the ServerNode and is primarily used by servers to communicate the writing order of objects in a scenario where write request for one object is initiated by multiple clients and these requests arrive at each server in a different order.

* + InputOutputHandler

This class contains 2 methods to read the configuration files containing information of the clients and the servers. It extracts the information of the number of clients and servers from these configuration files. It also stores in Hashmaps, the connection parameters (nodeId, port number, address) of all nodes.

* + SendConnectionThread

This thread is started by the ServerNode class and is responsible for initiating socket connections to other server nodes. This is to implement the logic of connecting all the servers with each other. In order to avoid redundant connections, each server node initiates socket connections only to server(s) with nodeId greater than its own nodeId. For this, it utilizes the Hashmaps created by the InputOutputHandler class.

* + ReceiveConnectionThread

This thread is started by the ServerNode class and is responsible for listening to connections made on the particular socket. It expects connections from all servers with nodeId less than that of its own. Thus, since the number of expected connections is known, it can stop listening after all connections have been made.

1. Experiment

We ran the code with client node trying to write object “vineet” with the value “hello”. The following observations were made using various debugging techniques and console output statements:

* The object “vineet” hashed to node 1. Thus the primary node for writing this object was calculated as server 1 and the secondary servers were server2 and server3.
* The client sent the object to server 1, 2 and 3 in the said sequence. Along with the objects, the servers were also informed if they were writing(only performed by the primary server) or replicating(only performed by the secondary servers)
* The primary server, upon receiving a write request, immediately writes the object and sends the object name and nodeId to corresponding secondary servers for that object. This information is stored in a queue by the secondary servers.
* The secondary server(s) upon receiving a request to replicate, buffer the message until the said request is at the top of the queue for that object type. At this instant, the object is written into stable memory by the secondary server. This mechanism of buffering and delayed writing by the secondary servers ensures total ordering of objects in case multiple clients try to write the same object with different values at the same instant.
* At each server, a file with the filename of the format <Objectname>\_<ServerNodeId>.txt was created and contained the latest update. This file for a particular object was found to be consistent amongst all servers.

1. Flowchart

Client:



Fig 1. Write an object



Fig 2. Read an object

Primary Server



Fig 3. Write request of object at primary server

Secondary Server Fig 4. REPLICATE request of object at secondary server(s)

1. Individual Contribution

We spent almost 20 man hours in designing the structure of the logic for a general read/write operation, maintaining of total ordering in case of concurrent writing of same object and finalizing the data structures required. After the design was finalized, we spent another 20-30 man hours over the next week in coding with Vineet contributing a major part. We used github for sharing the code. After the coding was done with, we started testing and debugging with Ishan helping in setting up the test case scenarios and Vineet testing the code. A few bugs were fixed during this phase with the code undergoing a few more design changes. After complete satisfaction of both team members with the code, this document was designed by Ishan after discussing with Vineet on the points to cover.