CS 6378: Project II

Nimisha Gupta

Suparn Gupta

April 24, 2014

Chapter 1

Introduction

Distributed systems usually keep the copies of data at multiple locations. The use of such copies is to minimize the data loss at the time of failures. The decentralized design is preferred over a centralized architecture because the main controller node becomes a single point of failuer. So, systems which run independently and still assure the data consistency with high availability are desired. Multiple servers provide the data protection guarantees against a given number of failures. However, ensuring consistency is a complex requirement. It depends on the actual requirements of the application. Engines like banking systems, airline reservation systems have strict requirements of consistency across all the replicated nodes. This may cause the overall performance of the system to go down in case of write operations. On the other hand, systems like social networking sites (Twitter) have loose requirements. These applications can tolerate momentary inconsistency among the data while believing that at some point of time in the future, the data will eventually become consistent across the whole system. Such application target the user's experience over the data consistency.

Since the replicated systems have multiple copies, they can also speed up the perfoance of the system. Because the client can chose the nearest replica which has the copy of the required object, the read operations tend to become faster. Using this notion, Facebook came up with a scheme "Write Locally, Read Globally", where the data is written to the nearest data center and is eventually updated across the whole network. However, read operations are performed across the whole network to get the latest version of the data. This is done so because the write operations are more expensive (slow, so as to say) than the read operations. This way, the user can be unblocked while the writes propagate across the system, while he still gets the latest data when reads are performed.

This project requires the design of a protcol which can ensure strict consistency across the system. The writes are allowed only if a mojority of servers are available.This ensures that a majority of servers have the latest data. The majority has a consent if atleast k servers agree to participate in the request. The complete problem can be summarized as follows.

The system consists of n servers and m clients. All the servers are interconnected to each other. Clients try to perform basic CRUD (Create, Read, Update, Delete) operations on the objects stored at the servers. A server must allow the creation, update and deletion only if a majority of servers, i.e atleast k-many servers are participating in the same request. Otherwise the request should be denied. However, the read operations should be allowed even if there is just one server, which has agreed to perform the read. The system can suffer from crash and network failures which can lead to partitioning in the network.

For the purpose of this project, values of n, m and k are chosen to be *7*, *5* and *2*. So, atleast *2* servers must be available to perform any write to the object. In the next sections, we describe how we can achieve this and the actual implementation of the system in Java. The further report is divided into following chapters: Chapter 2 describes the protocol, messages and sequence of events and Chapter 3 describes the implementation of the system in Java

Chapter 2

The Protocol

The goal is to create a protocol which :

1. Replicates the new objects and updates on a majority of servers.
2. Blocks any object updates, including delete operation if a majority of servers is not available.
3. Ensures the consistent concurrent writes if multiple clients try to modify the same object.
4. Allows the clients to read the latest version of the object, which is accessible.

We designed a protocol which satisfies all the above criteria based on Google File System [4]. Before we present the details of the protocol, lets define some terms which we will use in further sections. Any operation which modifies an object on in the file system, which includes, creation, update and deletion are collectively referred to as *mutation operations* or simply *mutations.* All these operations are similar to each other in a sense that, the requirements 1, 2 and 3 must be satisfied in all the operations. All the servers are *peers* in a sense that the client has access to all the servers. This is different from a proxy based architecture where a single node is facing the client hiding the internal topology. In the next section, we present the overall system architecture.

2.1 System Architecture

There are 7 servers in the system which are running identical processes. All the servers are interconnected to each other via direct links. All the communication channels use TCP as the underlying protocol. So the communication is reliable.

Figure 2.1: The System Architecture.

Algorithm 2.1 Computing Server IDs for the request.

- *H(s)* = ASCII sum of object ID, *s*
- Server 1's id = *H(s) % 7*
- Server 2's id = *(H(s) + 1) % 7*
- Server 3's id = *(H(s) + 2) % 7*

In Figure 2.1, the servers are represented in a ring architecture. However, the reader should not be confused as its just a way to arrange the servers in the figure. Internally, there is no specific ring protocol required for this system. Each server has a unique identifier in server namespace and holds a certain number of objects. Each object has a unique identifier. Such an identifier is generated using an approach used by Twitter as discussed in [3].

Then there are client nodes which also have a unique identifier associated with them within the client namespace. In case of no failures, each client can connect to each server via direct TCP connection. Also, the object ids for each object are generated by the client as per the need. Instead of using the network card addresses, we use the client's id as a unique identifier for the client while generating the object id. Each object id thus consists of the following parts:

1. The current local timestamp of the client in milliseconds
2. The client's id
3. A counter value which increments by one after each new id is generated and resets when reaches 10. The maximum value is configurable as per the needs of the application. If the application has a very high amount of writes happening per second, then it makes more sense to increase the counter's maximum value.

A client can send only a single request to *mutate* an object at a time. Only after the operation is successful or it fails, it can issue the subsequent requests. However, multiple clients can issue requests for the same object concurrently. Each object id uniquely maps to exactly 3 servers using a hash function *H(s)* where *s* represents the object id. In this implementation, the hash function returns the ASCII sum of all the characters in the object id. Then server ids are calculated as follows:

2.2 Protocol Model and Messages

An overview of the protocol is presented next. Before servers start serving the clients, a bootstrap process is executed. All servers identify each other via network discovery messages. In this phase, they exchange information about their host names and their node ids. Once all the servers have been identified, the servers become ready to server the clients. Once discovery process completes, the clients can begin requesting services from the servers. All the servers keep probing the other servers to check their liveness by sending the *HEARTBEAT* messages. If the recipient is alive, it responds with *HEARTBEAT*_ACK message.

A client can ask for two kinds of services: *Read* and *Mutation*. *Mutation* includes write, update and delete operations. The client computes the relevant server ids based on the object id. In order to *read* an object, the client simply sends a read request and if the object is available to read, the server returns the object to be read. If the server is unavailable or it doesn't have the object, the client sends a request to the next server. If the client wants to mutate an object, first it verifies if at least two servers are available to perform mutation. If there are at least two servers available, it then sends a request to gather the information about the primary server. The servers decide amongst themselves which server should act as a primary. This is different from the design presented in [4] where there is a separate master server which is used to find the primary server. Once the servers decide upon the primary, the information about the primary server is returned to the client. The client then forwards the mutation request to all the relevant servers. The servers buffer the request in memory.The client then, sends the mutation write request to the primary server to commit the transaction. The primary server assigns this request a unique serial number and relays the request to all the other servers where the operation has to be performed. If the mutation is successful, each replica responds to the primary server with an acknowledgement. Once the primary server obtains at least two acknowledgements from replicas including itself, it sends acknowledgement to the client. The reader may notice that the *mutation* requires the consent of majority of nodes similar to the one described by Gifford in [5] with *weight equal to 1 for each object*.

The relevant servers computed using 2.1 select the primary server for the request based on the node ID. The server, which has the least value of node ID, and is alive is chosen as the primary server. Let us look at the types of messages exchanged among different entities in the system.

2.2.1 Types of messages

2.2.1.1 Network control messages

The set of messages exchanged between servers during the network discovery in order to identify each other are descibed below:

- *SERVER_INTRO*: Each server to every other server to identify itself to other servers.
- *SERVER_INTRO_REPLY*: This message is sent by a server on receiving a SERVER_INTRO message to acknowledge the other server.
- *DISCOVERY_COMPLETE*: This message is sent to all the servers once a server identifies all other servers in the network.

Messages exchanged to check if a server is reachable at any point of time:

- *HEARTBEAT*: Sent from client or server to servers to check if the server is reachable in the network.
- *HEARTBEAT_ECHO*: Reply sent from the server that received the heartbeat request.

2.2.1.2 Read operation

Messages exchanged to read an object:

- *READ_OBJ_REQ*: Sent from client to server to request to read an object.
- *READ_OBJ_SUCCESS*: Sent from server to client upon successful read operation along with the read data.
- *READ_OBJ_FAILED*: Sent from server to client if the read operation fails.

2.2.1.3 Mutation messages

Messages exchanged to mutate an object:

- *WHO_IS_PRIMARY*: Sent from client to servers to identify the primary server.
- *PRIMARY_INFO*: Sent from server to client informing the client which server will act as the primary server for this request.
- *MUTATION_REQ*: Sent from client to server to request a mutation operation on the object.

- *MUTATION_ACK*: Sent from server to client upon receiving the mutation request that it has accepted the request and placed it in buffer.
- *MUTATION_REQ_FAILED*: Sent from the server to the client if the mutation request is rejected.
- *MUTATION_WRITE_REQ*: Sent from client to the primary server to commit the mutation request.
- *MUTATION_WRITE_ACK*: Sent from primary server to the client upon successful mutation request commit.
- *MUTATION_WRITE_FAILED*: Sent from primary server to the client upon mutation request failure.
- *MUTATION_PROCEED*: Sent from primary server to other servers upon receiving *MUTATION_WRITE_REQ* from the client.
- *MUTATION_PROCEED_ACK*: Sent from replica servers to the primary server when they finish mutating the object locally.
- *MUTATION_PROCEED_FAILED*: Sent from replica servers to the primary server in case the mutation of the object failed.

2.3 Communication Model

On boot up, each server starts a discovering the network. It periodically sends out SERVER_INTRO message to all the known addresses. At this point, the server doesn't know which specific node is listening on that address. On receiving this message, the node replies with *SERVER_INTRO_REPLY* message which contains the information about the receiver. Once a server knows about all the other servers, it broadcasts a *DISCOVERY_COMPLETE* message. This message informs the other receivers that the sender node is ready to accept the client connections.

Once the servers are ready to connect to the clients, there are two possible operations as described below:

2.3.1 Read

In order to read an object, the client sends a *READ_OBJ_REQ* to one of the three servers where the object resides. Following cases may arise:

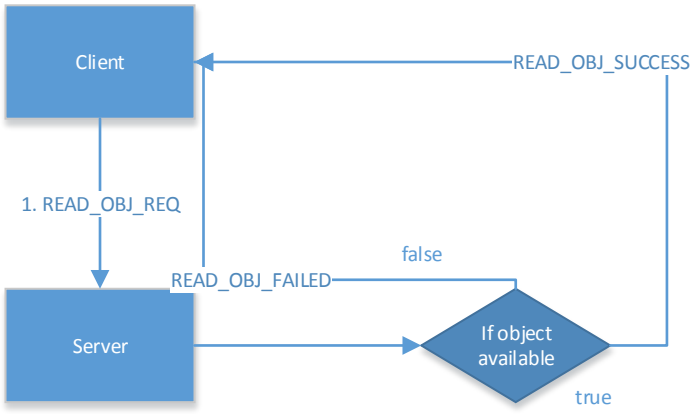

Figure 2.2: Read Operation.

- If the server is available to process the request and the object is available, it returns the object to the client with a *READ_OBJ_SUCCESS* message.
- In case the server is available but the object is being used in another request, the server queues the request and processes it once the object is released from the other request.
- In case the object does not exist on the server, it returns a null value informing the client that the object does not exist on this server with *READ_OBJ_FAILED* message.

The client keeps sending the request to the next relevant server until it receives the object or list of servers exhausts.

2.3.2 MUTATION

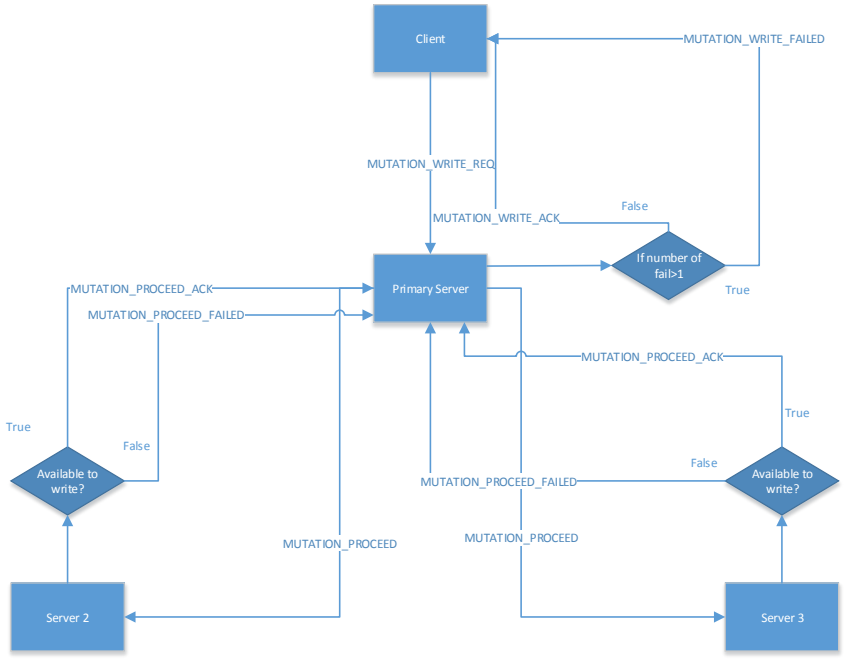

Figure 2.3: Mutation Operation.

To perform any mutation operations (Create, Update, Delete), the following procedure is employed. The client sends a *HEARTBEAT* message to the three servers to identify if they are accessible. If the server is alive, it replies with a *HEART_BEAT_ECHO* message. If the client does not receive at least two *HEARTBEAT_ECHO* messages, it fails the request. Otherwise, It requests for information about which server will act as the primary server for this request by sending a *WHO_IS_PRIMARY* message to all the servers, which replied with *HEARTBEAT_ECHO* messages. The relevant servers decide amongst themselves based on availability and node id, the primary server and respond with a *PRIMARY_INFO* message to the client which holds the information (node ID, host name and port number) about the primary server. The client then sends a *MUTATION_REQ* to all the reachable serves. Upon receiving this message, the server decides if it ready to buffer this ready. If it is able to store the request in the buffer, it replies with a *MUTATION_ACK* message to the client, otherwise replies with a *MUTATION_REQ_FAILED* message. The client on receiving at least two acknowledgements sends the *MUTATION_WRITE_REQ* to the primary server, else it aborts the operation. The primary server on receiving write request assigns a request id to the request and forwards the list of serial numbers and request ids for the same object

with a message type *MUTATION_ PROCEED* to all the reachable servers including itself (to simplify the implementation, the primary server also sends the same request on its client port). The servers on receiving the request decide locally if they are available to perform this operation. Once the decision is made and operation is performed either *MUTATION_ PROCEED_ ACK* or *MUTATION_ PROCEED_ FAILED* is sent. If the primary server receives more than one failed message, it aborts the entire transaction and sends a MUTATION_WRITE_FAILED message to all the servers and the client there by maintaining consistency. Otherwise, it responds the client with *MUTATION_ WRITE_ ACK* message.

2.3.3 CONCURRENT MUTATIONS

If two clients try to mutate the same object at the same time, the primary assigns a serial number to the request id and then the request with a lower serial number is processed first while the subsequent requests are stored in a buffer until the first request is processed. The subsequent request is processed only after the first request has been processed. This ensures the consistency of data across all the replicas.

Chapter 3

Code Implementation

There are two modules in the implementation:

1. Server Module
2. Client Module

Both modules run independently but use a number of common functions, like serialization and de-serialization of the message data. Figures 3.1 and 3.2 show the code structure of both the modules. In both modules Main is the entry point. It requires an argument which acts as the server ID or the client ID based on the module which is being run. Once the server node is started, the connection manager starts the network discovery of the whole system using the ServerToServerChannel and ServerToServerHandler. As soon as the server becomes ready, the ServerToClientChannels are started. All the requests on this channel are handled by ServerToClientHandler.

When the client is started, it starts an InputProcessor to accept various commands from the user. A user can perform CRUD operations using this menu. Both server and client use Logger, Globals and MessageParser for various functions. MessageParser parses the bytestream into meaningful instances of various kinds of messages. Each message itself is an instance of a class which is sent as a byte stream along with a message type (as describe in section 2.2.1) and wrapped in WrapperMessage. Google's JSON parsing library known as Gson [1] has been used in this implementation to parse the messages and data objects while reading from the files. Maven [2] is used as the build tool. A number of test cases have also been designed using JUnit to verify the correctness and measure the performance. The detailed documentation of the code is bundled along with this report.

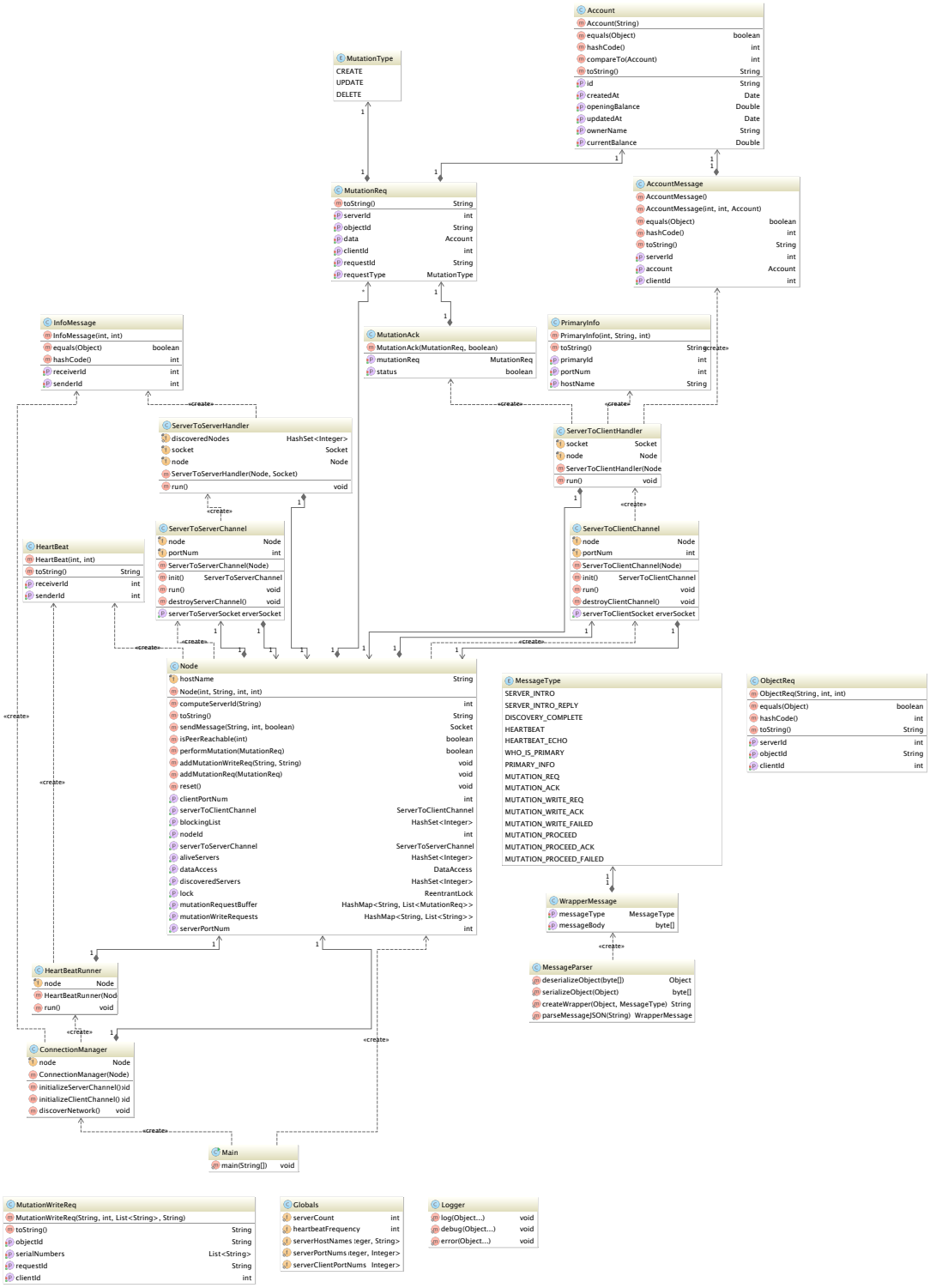

Figure 3.1: UML Class Diagram for Server Node

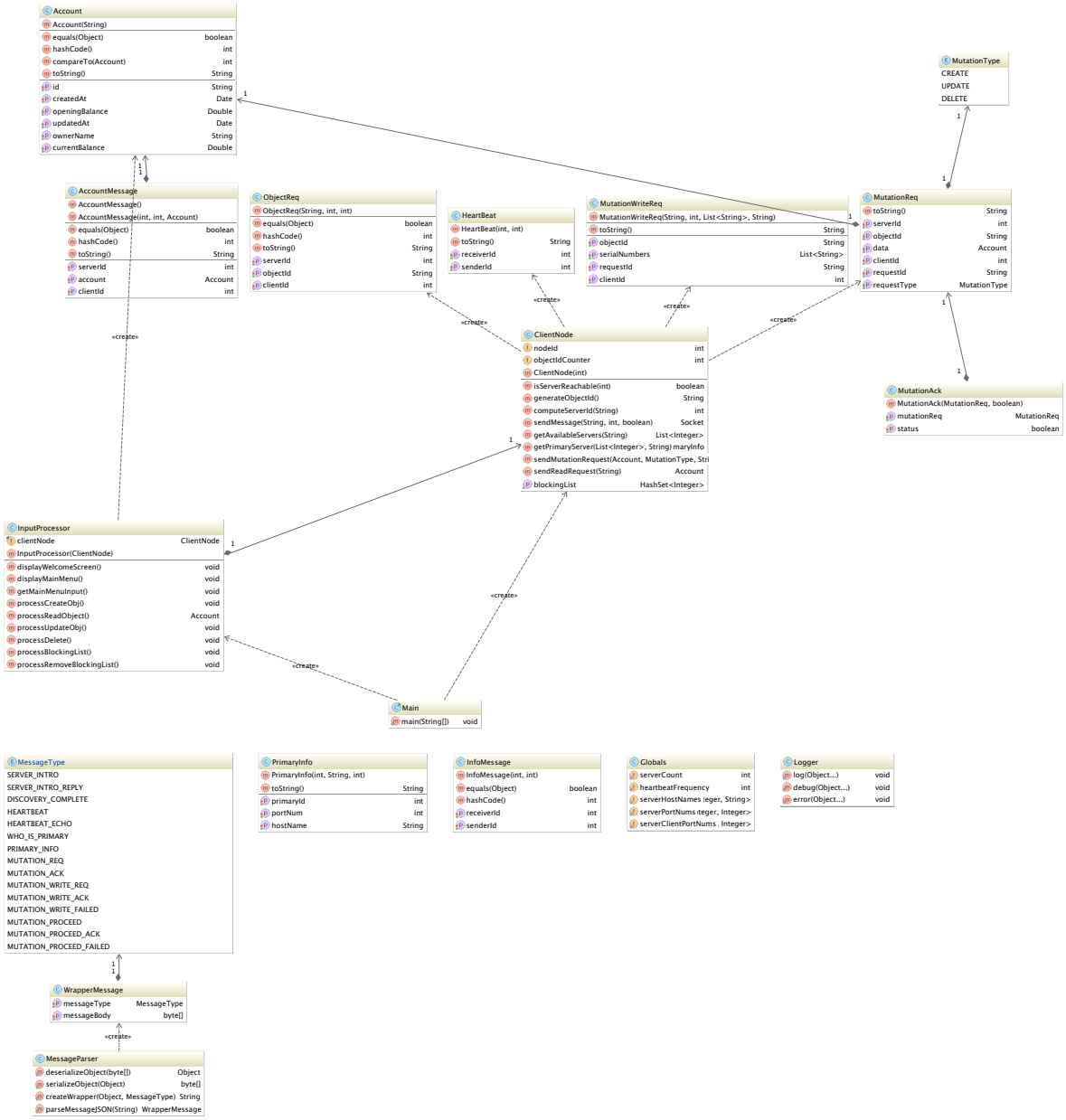

Figure 3.2: UML Class Diagram for Client Node

To configure the system, two files *server.config* and *server-client.config* are required. File *server.config* contains the hostnames and the corresponding port numbers of the servers for server to server communication. The *server-client.config* contains the network information about the client-server communication channels. A number of scripts are also bundled which compile and run the system.

Bibliography

[1] URL `https://code.google.com/p/google-gson/`.

[2] URL `http://maven.apache.org/`.

[3] URL `https://blog.twitter.com/2010/announcing-snowflake`.

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. *ACM*, 2003.

[5] David K. Gifford. Weighted voting for replicated data. *ACM*, 1979.