REPLICATED DATA STORAGE

A totally ordered multicast based solution using intra LAN servers

Angioletti Daniele
Cattaneo Davide

SUMMARY

PROBLEM DESCRIPTION

The goal of the project is implementing a replicated data storage capable of storing data in the format (int id, int value) over multiple servers on the same LAN keeping the information consistent among all the nodes.

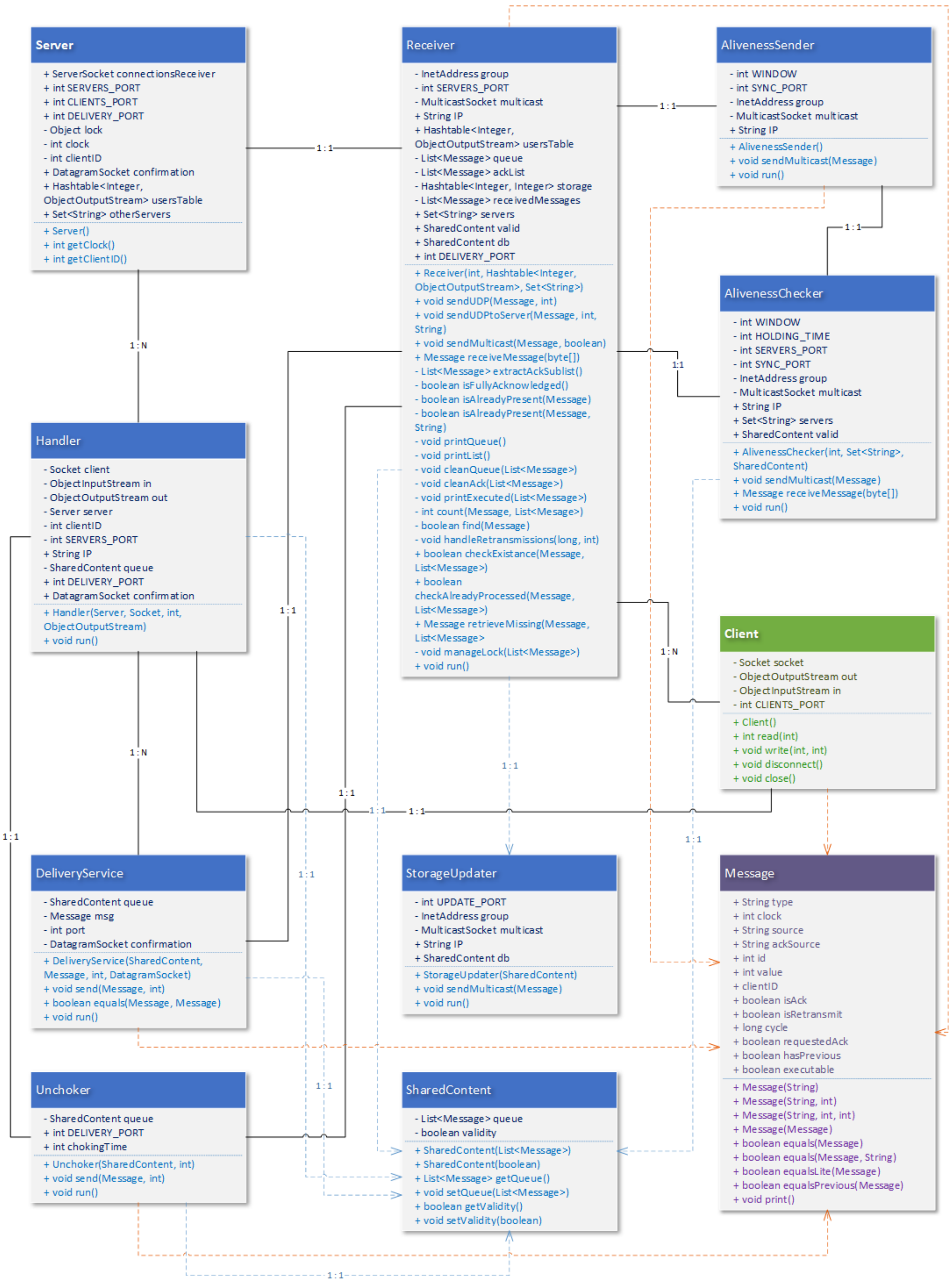
The clients are provided with two primitives:

- *int read(int id)*
- *void write(int id, int value)*

and can connect to any of the available servers. The system provides sequential consistency through the implementation of a totally ordered multicast primitive.

Servers are unknown one to the others and are assumed to be unreliable.

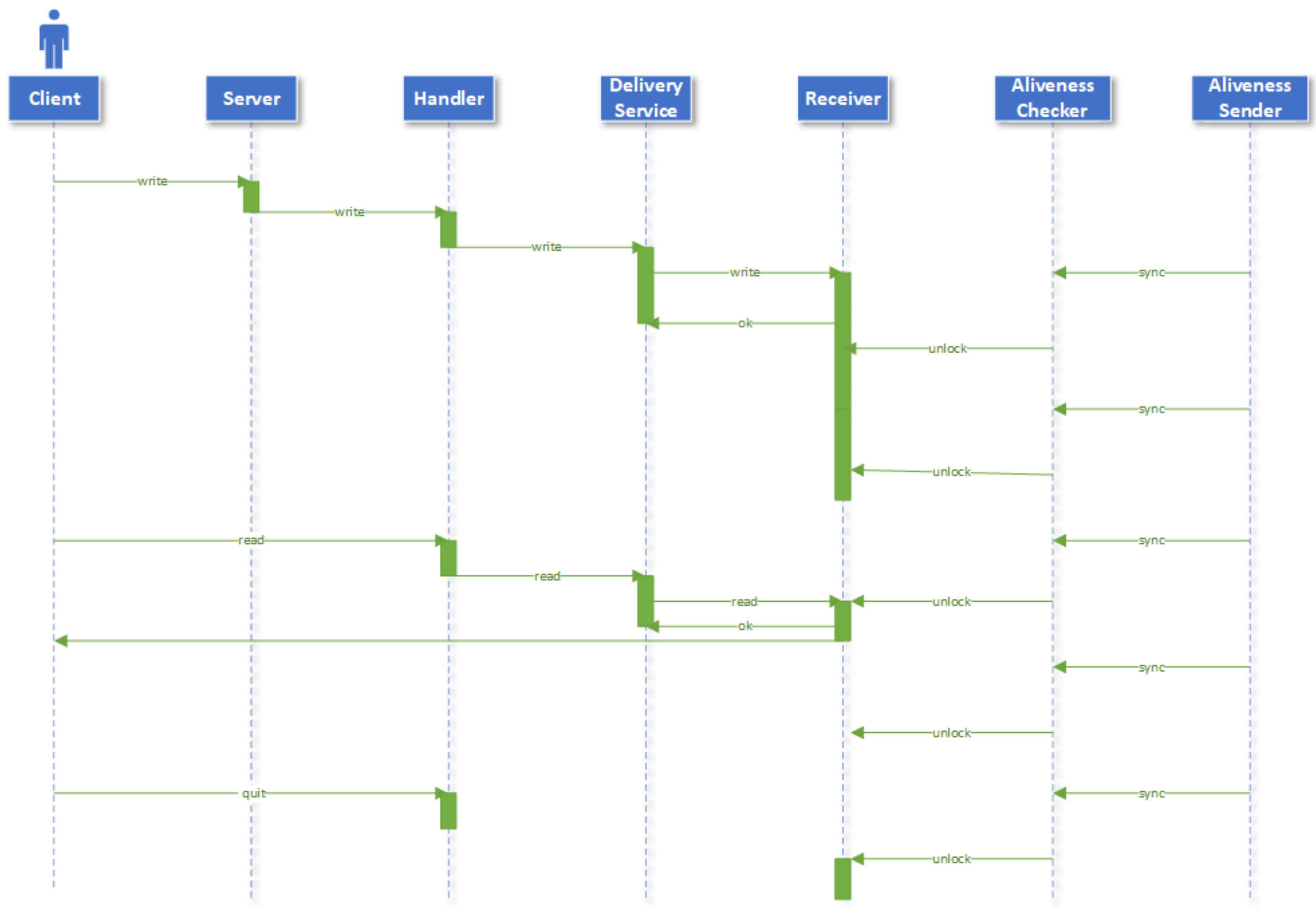
UML CLASS DIAGRAM



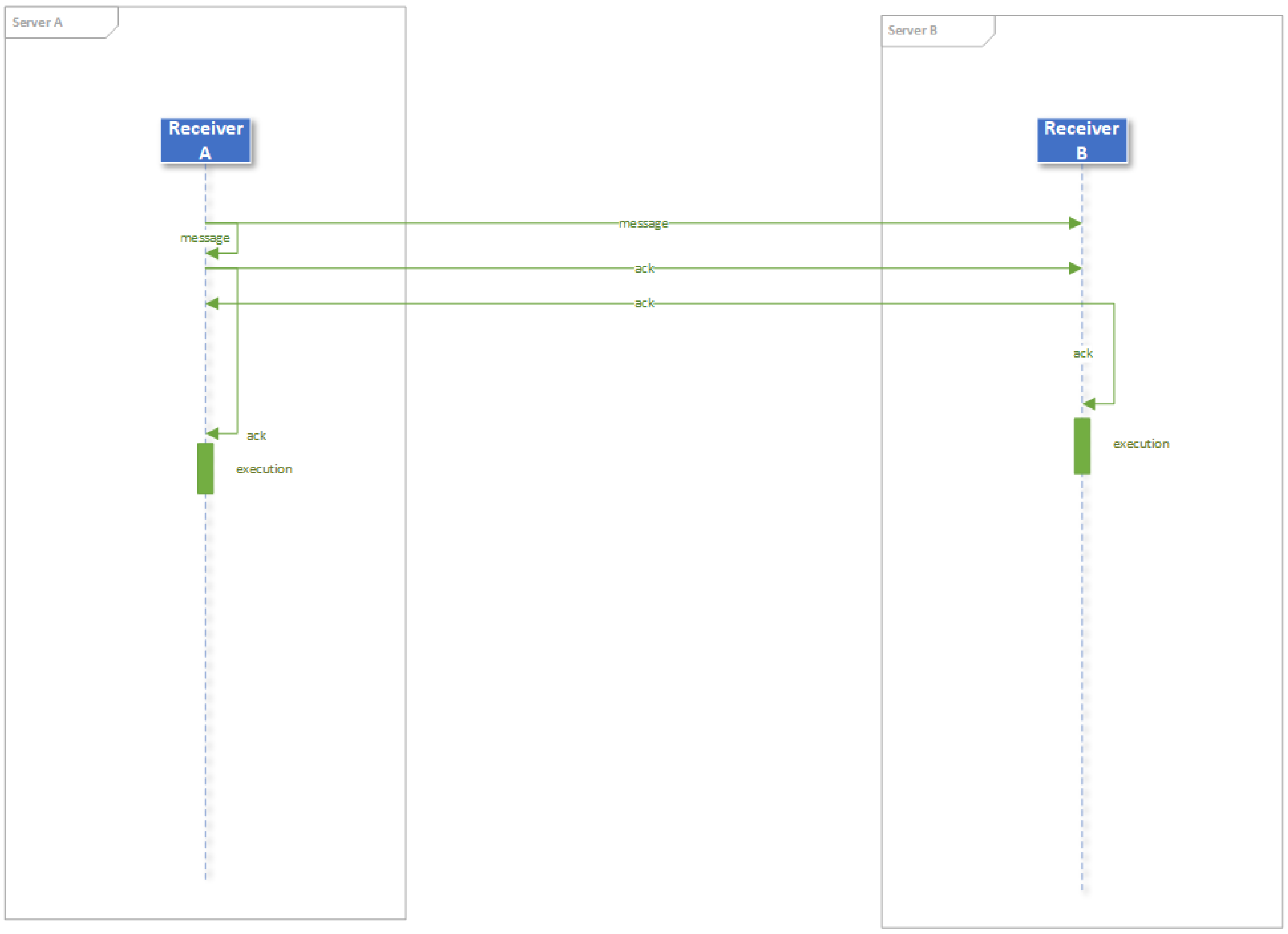
Server
+ ServerSocket connectionsReceiver
+ int SERVERS_PORT
+ int CLIENTS_PORT
+ int DELIVERY_PORT
- Object lock
- int clock
- int clientID
+ DatagramSocket confirmation
+ Hashtable<Integer, ObjectOutputStream> usersTable
+ Set<String> otherServers
+ Server()
+ int getClock()
+ int getClientID()
Receiver
- InetAddress group
- int SERVERS_PORT
- MulticastSocket multicast
+ String IP
+ Hashtable<Integer, ObjectOutputStream> usersTable
- List<Message> queue
- List<Message> ackList
- Hashtable<Integer, Integer> storage
- List<Message> receivedMessages
+ Set<String> servers
+ SharedContent valid
+ SharedContent db
+ int DELIVERY_PORT
+ Receiver(int, Hashtable<Integer, ObjectOutputStream>, Set<String>)
+ void sendUDP(Message, int)
+ void sendUDPtoServer(Message, int, String)
+ void sendMulticast(Message, boolean)
+ Message receiveMessage(byte[])
- List<Message> extractAckSublist()
- boolean isFullyAcknowledged()
- boolean isAlreadyPresent(Message)
- boolean isAlreadyPresent(Message, String)
- void printQueue()
- void printList()
- void cleanQueue(List<Message>)
- void cleanAck(List<Message>)
- void printExecuted(List<Message>)
- int count(Message, List<Mesage>)
- boolean find(Message)
- void handleRetransmissions(long, int)
+ boolean checkExistance(Message, List<Message>)
+ boolean checkAlreadyProcessed(Message, List<Message>)
+ Message retrieveMissing(Message, List<Message>
- void manageLock(List<Message>)
+ void run()
AlivenessSender
- int WINDOW
- int SYNC_PORT
- InetAddress group
- MulticastSocket multicast
+ String IP
+ AlivenessSender()
+ void sendMulticast(Message)
+ void run()
AlivenessChecker
- int WINDOW
- int HOLDING_TIME
- int SERVERS_PORT
- int SYNC_PORT
- InetAddress group
- MulticastSocket multicast
+ String IP
+ Set<String> servers
+ SharedContent valid
+ AlivenessChecker(int, Set<String>, SharedContent)
+ void sendMulticast(Message)
+ Message receiveMessage(byte[])
+ void run()
Handler
- Socket client
- ObjectInputStream in
- ObjectOutputStream out
- Server server
- int clientID
- int SERVERS_PORT
+ String IP
- SharedContent queue
+ int DELIVERY_PORT
+ DatagramSocket confirmation
+ Handler(Server, Socket, int, ObjectOutputStream)
+ void run()
Client
- Socket socket
- ObjectOutputStream out
- ObjectInputStream in
- int CLIENTS_PORT
+ Client()
+ int read(int)
+ void write(int, int)
+ void disconnect()
+ void close()
DeliveryService
- SharedContent queue
- Message msg
- int port
- DatagramSocket confirmation
+ DeliveryService(SharedContent, Message, int, DatagramSocket)
+ void send(Message, int)
+ boolean equals(Message, Message)
+ void run()
StorageUpdater
- int UPDATE_PORT
- InetAddress group
- MulticastSocket multicast
+ String IP
+ SharedContent db
+ StorageUpdater(SharedContent)
+ void sendMulticast(Message)
+ void run()
Message
+ String type
+ int clock
+ String source
+ String ackSource
+ int id
+ int value
+ clientID
+ boolean isAck
+ boolean isRetransmit
+ long cycle
+ boolean requestedAck
+ boolean hasPrevious
+ boolean executable
+ Message(String)
+ Message(String, int)
+ Message(String, int, int)
+ Message(Message)
+ boolean equals(Message)
+ boolean equals(Message, String)
+ boolean equalsLite(Message)
+ boolean equalsPrevious(Message)
+ void print()
Unchoker
- SharedContent queue
+ int DELIVERY_PORT
+ int chokingTime
+ Unchoker(SharedContent, int)
+ void send(Message, int)
+ void run()
SharedContent
- List<Message> queue
- boolean validity
+ SharedContent(List<Message>)
+ SharedContent(boolean)
+ List<Message> getQueue()
+ void setQueue(List<Message>)
+ boolean getValidity()
+ void setValidity(boolean)
1 : 1
1 : 1
1 : 1
1:1
1 : N
1 : N
1 : N
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1
1 : 1

2

UML SEQUENCE DIAGRAMS



Sequence diagram representing the development of the interactions between a client and the main components of the server.



Sequence diagram representing the exchange of messages that brings to the execution of a client’s request.

FUNCTIONAL DESCRIPTION

A client may connect to any of the active Servers and when it does, it gets assigned to a Handler responsible for its management. The Handler receives the messages sent by the Client and for every of them it in instantiates a DeliveryService, namely a thread devoted to the delivery of the message to the Receiver. A correct delivery is granted by means of the Unchoker; it is a thread generated by every Handler that shares a queue of messages (ordered according to their submission) with all the active DeliveryServices. If a message is stalling in the queue due to an unreceived confirmation by the Receiver, the Unchoker will retransmit it after 10 seconds. When a message is confirmed, it’s pop-ed from the queue.

The core component of the software architecture is the Receiver. It’s responsible for managing the queue of the delivered messages and communicates with the other servers to keep a consistent, distributed data set. When it receives a message from a DeliveryService, it immeditely sends back a “confirmation massage” to confirm the reception. Then, after the message is added to the queue, it multicasts that message to all the other available server. Sych multicast is performed just in case of a write request. Reads are performed immediately by the receiving server.

When a Receiver recevices a message from another server, it adds it to its own queue and multicasts an “ack message” to confirm the insertion.

If a server owns the “ack messages” by all the members of the group, it can can finally execute the first message in the queue.

A series of reliability controls is managing every possible retransmission, if needed.

A message, to be executed, must follow the previous one (scalar clock). A list of executed messages helps in this kind of control. This policy is foundamental to grant the correct behaviour of the application due to the nature of the software architecture.

The available servers are detected autonomously at run time and may change during time. Every Receiver generates two threads: the AlivenessSender, that simply sends a “syc message” every X seconds (time window), and the AlivenessChecker. This last component shares a SharedContent with the Receiver, that is a set of IPs. Every “sync message” is signed with the IP of the sender, thus the AlivenessChecker, by collecting those messages, can know the number of active servers and share it with the Receiver.

Given that the reception of the messages is a blocking activity, the AlivenessChecker also generates an “unlock message” at the end of every time window. This allows the Receiver to “unlock” and carry on its tasks even if it’s not receiving any other message from the clients. It’s a way to force a stepped execution.

FUTURE IMPROVEMENTS

At the moment the software architecture relies on the existance of a list of executed messages to keep track of the execution history. This list allows to detect if there’s a missing message that requires to be retransmitted (needed in a multiclient scenario). The actual implementation, however, is never cleaning that list from no-more-needed messages causing an increase in the memory usage on the long run. A future improvement should certainly focus on finding a way to clean that list without deleting the useful information.

Just to give a better idea of the problem, here is a proposed scnario. Imagine to have 2 clients: one is submitting continuous requests and the other one has submitted just one request at the beginning of the session. To clean the execution list, one should count all the clients there present and for every of them keep just the last message, which is the only useful).

A second improvement could provide a better exploitment of the Message fields. At the moment that class uses many fields to transmit the information. Some of those fields could probably be used for multiple purposes and others removed. In the actual scenario, for instance, the field AckSource could used both for marking the source of a reply message and for transmitting the destination of the reply in case of a retransmission request. When the retransmission request is issued by server A, the IP of A could be put as a AckSource and sent to the missing servers among the available ones. The receiving server should extract that field, put its own IP as AckSource and send back the requested message only to the extracted destination.

This optimization will reduce the size of the messages, increasing the number of them that could fit inside the receiving buffer (8KB) before information gets lost and will also maximize the number of messages that can be sent over a channel of fixed capacity.