Replicated data storage

A totally ordered multicast based solution using intra LAN servers

Davide Cattaneo

Tizio Caio

SUMMARY

Problem Description 1

UML class diagram 2

UML sequence diagramS 3

FUNctional DESCRIPTION 5

FUTURE IMPROVEMENTS 7

# 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.

# Immagine che contiene screenshot Descrizione generata con affidabilità elevataUML class diagram

# Immagine che contiene testo, mappa Descrizione generata con affidabilità elevataUML sequence diagramS

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

![Immagine che contiene mappa, screenshot

Descrizione generata con affidabilità molto elevata]()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.

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

If a server owns the “ok messages” by all the other members of the group, it can then multicast an “ack message”. Only when the server has received all the “ack message” can finally execute the first message in the queue.

A series of reliability controls is managing every possible retransmission, if needed. Retransmissions are target specific, i.e. are required just to the missing servers (messages sent to the missing IPs). This kind of optimization avoids flooding the network excessively.

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 8 seconds (time window), and the AlivenessChecker. This last component shares a SharedContent with the Receiver, that is a set of IPs. Every “syc 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 trak 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 improvementshould 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, whish 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 is 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 is put as a AckSource and sent to the missing servers among the available ones. The receiving server extracts that field, puts its own IP as AckSource and sends back the requested message 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.