Middleware Project Report
Simple DBMS

Davide.Perina@studenti.unitn.it (128698)
Stefano.Testi@studenti.unitn.it (128697)

Academic Year 2008-09
March 14th, 2009

1 Introduction

SimpleDBMS is a software implementation of a simple distributed database using 2-phase-commit for the spreading of updates. The atomic element to store is a couple [key,value], and only INSERT, DELETE and QUERY operations can be performed.

Once an INSERT or a DELETE operation is issued from a client, one of the DB server is contacted (based on the key to modify), and this server should take care of contacting all other server and start an agreement process using the 2-phase commit protocol. On the other hand, a QUERY operation can be performed on any of the server, since the entire system should be consistent.

Hence, each one of the server is responsible for handling INSERT and DELETE operation on key belonging to a specific range (if p_i is the server number, it will handle INSERT and DELETE for $p_i \leq key < p_{i+1}$), while QUERY operations can be sent to any server.

To do this, any of the server can act both as *Active Server* (from now on, AS) or *Passive Server* (PS). It is important to highlight that this is simply a logical distinction, since the source code of the server does not create "different" servers, and there are no "active" or "passive" children.

1.1 Server initiation

Every server, once started, reads from a file (*serverlist.txt*) a list of the server belonging to the distributed DB, and the port on which they're listening to incoming connection. Plus, also the local port is taken from the file.

Then, it opens a socket and waits for incoming packets coming from clients or any other servers (see figure 1). Once a packet is received, the server checks if the packet is coming from a client or a server.

- If the packet is coming from a CLIENT, it is passed to the Active Section of the server (see paragraph 1.2).
- If the packet is coming from another SERVER, it is checked to see if the content of the packet is a new 2PC request initiated from another server or if it is an answer to a previous started 2PC. In the first case, the Passive section (see paragraph 1.3) of the server is started, in the latter case, the response from the remote server is read, and the Pending Operation List (POL) is updated.

1.2 The Active Server section

An Active Server (AS, Figure 1) is one which receives a request from a client, and act as master in a two phase commit process. Once a request is received, the Active Server:

- Checks if the request coming from the client should be performed by himself (or, if $p_i \leq key < p_{i+1}$ is satisfied)

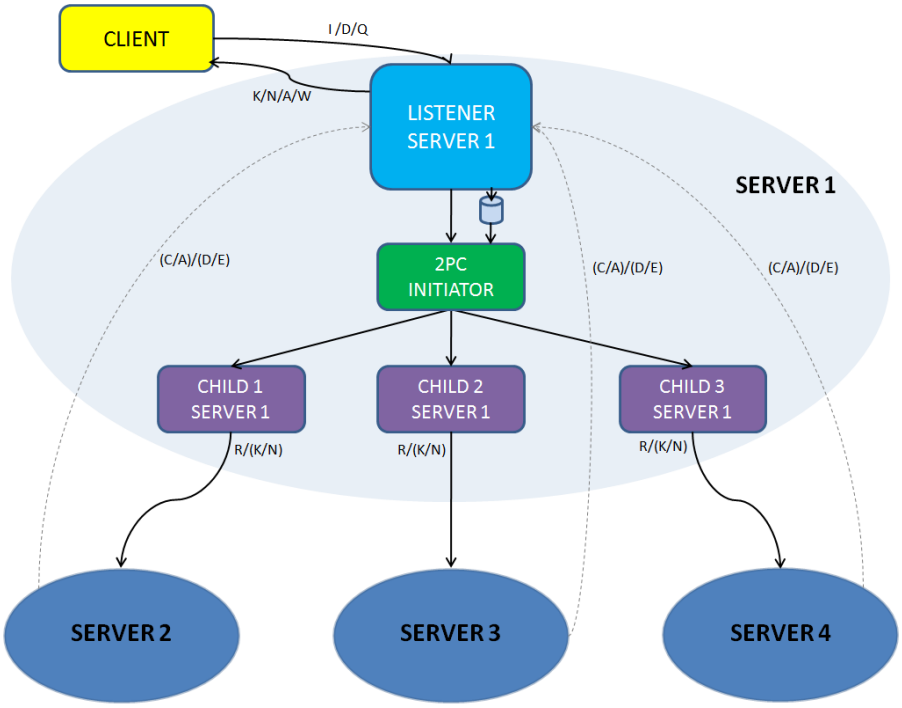

Figure 1: The server when acting as Active

- Checks if another operation on the same key is taking place on the distributed DB, by looking to a local Pending Operation List
- If any (or both) the previous checks are unsuccessful, the client is notified that the requested operation cannot be served because it has been sent to a wrong server (W) or because another operation is taking place on the same key (A). On the other hand, if both previous checks are successful, a unique ID is generated and associated to the operation (which is put into the local Pending Operation List with the flag status set to '1', meaning Active), and the 2PC is started.
- Once a 2PC is started, a new child is spawned, and a pipe is open with the father. This new child is called "2PC initiator" (see Figure 1).
- The 2PC initiator spawns a child for each server, to contact them and send them the Commit Request (R). The Commit Request is sent also to the local server (specifically, to the passive section of the same server), which is seen as another remote server. After sending the Commit Request, all the children will die, and the 2PC initiator will wait for the father to receive all the Commit Response (C/A) from the other servers, and for the global decision to be written into the pipe.
- Once the father has received all the responses, it will decide if the global decision is a Global Commit (K) or a Global Abort (N), and this information is passed through the pipe to the 2PC initiator.
- At this point, the 2PC initiator will contact all the other servers to inform them about the global decision, and then it will kill himself.
- Once the father has received the acknowledgment (D/E, where D is the ACK, and E is a notification of an Error occurred while writing into the Stable Storage) to the global decision (K/N), the operation is removed from the POL, and the client is notified about the commit/abort of the operation.

1.3 The Passive Server section

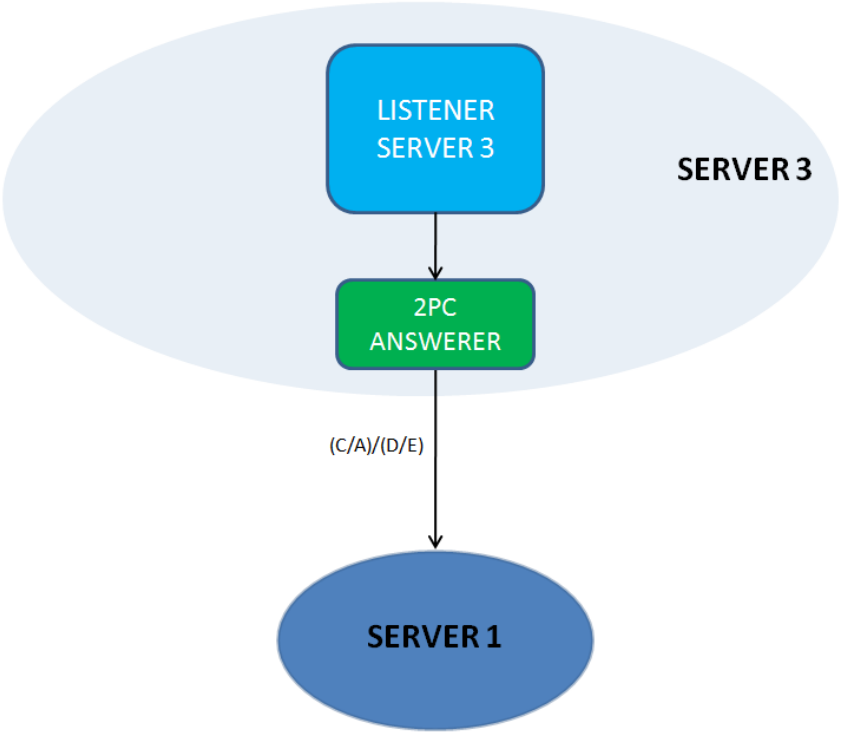

Figure 2: The server when acting as Passive

The Passive Server (PS, Figure 2) is the logical set of actions performed when a Commit Request (belonging to a 2PC remotely initiated) is received. Once a Commit Request is received, the PS:

- Checks if another operation on the same key is taking place on the distributed DB, by looking to a local Pending Operation List
- If another operation is found, the PS responds to the Commit Request with an 'Abort' (A), otherwise the operation is extracted from the received packet, and put into the local Pending Operation List (with the flag status set to '0', meaning Passive). Then, the PS creates a child (2PC answerer) to respond to the Commit request with a 'Commit' (C).
- The 2PC answerer dies, and the PS will wait for the Global Decision (Global Commit 'K' or Global Abort 'A') to be received.
- Once the Global Decision has been received, the operation is either performed on the Stable Storage (if Global Decision was K) or not (A), an ACK (D) sent back to the active server, and the operation removed from the Pending Operation List. If some kind of error occurs while accessing the Stable Storage, the ACK packet contains an 'E' instead of 'D', to enable possible further possibilities on the active side (not handled here, see Crash failure handling paragraph).

1.4 Crash failure handling

It is important to specify that this software is crash-failure resistant. The Pending Operation List, used both by the active and the passive side, has a timeout for each operation.

Once a new operation is inserted into the list, a timestamp is recorded. A timeout is associated to the staying of any operation into the POL: an operation is declared as dead whenever *lifetime_row* seconds have passed without the operation being removed from the POL.

A specific function checks the Pending Operation List every *timeout* seconds for dead operations. If a dead operation is found, it is removed from the list.

As an example, if we consider *lifetime_row* = 20s and *timeout* = 10s, it is granted that any operation can stay in the POL for a maximum time between 20 and 30 seconds.

Any successive operation reaching the distributed DB after a server crash will be aborted, but the system will not be stucked into waiting a response from a crash server. Once a server returns online, the entire system recovers (there is no "recovery procedure", the system simply starts working again), and all subsequent operation will be either accepted or aborted.

Thanks to this, Consistency is achieved, even in presence of crash failures.

There's only problem that could break consistency, and it happens whenever a Global Commit has been received by a passive server, but the server crashes before the operation be actually performed, or whenever the system doesn't crash, but the writing into the stable storage fails.

In this two situations, once (and if) the server returns online, the distributed DB is no longer consistent. This is a specific problem related to the use of 2PC instead of other algorithms like 3PC. It could be solved by changing the agreement procedure, or by using the negative ACK 'E' returned by the passive server when the stable storage fails.