A High Availability Replicated Locking Service

Distributed Systems - 2011/12
Carlo Caprini - Mattia Zeni

The two main components that characterize this systems are the client-side library *locking.c* and the replicable server *server.c*.
locking.c contains the two functions that a client running *application.c* can invoke in order to access and lock a resource controlled by the server (*acquire* function) and later release it (*release* function).
server.c defines both the master server, which is the one responsible for the communication with any client that wants to lock or release a resource, and the replicated servers, which are contacted by the master server in order to be synchronized and kept up to date every time the state of a resource is being modified. The number of replicated servers has been set to two.
In both cases, client/master-server and master-server/replicated-server communication, the system creates a connection through sockest by using a reliable TCP transmission. The exchanged messages can contain two types of data, a structured payload (*msg*) or an int value (usually named *answ*). The latter one is needed when the master server is responding to the acquire or release request of a client and it is sufficient to identify the outcome of the requested operation. The former data type is instead composed by four int values that represent respectively the client ID of the user performing the request (*msg.clientid*), the resource ID of th resource to lock or release (*msg.resid*), the operation that should be performed (*msg.op*) and the identity of the sender of the message (*msg.identity*), a client or a server.
Every client knows that servers can be found in the range of number of ports [5050-5060] and locally stores the current position of the master server in a variable called *master_port_number*.
When *acquire* or *release* functions are called, the client-side library tries to contact the master server on the last port through which a connection between the two has taken place (if it is the first request being issued, the client will iteratively try to contact the master server starting from the defaul port 5050).
As soon as the master server is found and a new connection is established, the client sends a message containing the *struct msg* that describes the request (*client_id*, *res_id*, *op* - 1 for acquire and 2 for release, *identity* set to 1). When the master server receives a message containing a *struct msg*, it first checks the identity of the sender and if this one results to be a client it starts processing the request by analizing the other fields of the *struct*. Depending on the value of *msg.op*, the master server calls the *acquire* or the *release* function. The first task of these functions is to check if the requested operation can be performed and only in this case the replicated servers are going to be contacted for synchronization.
The group of resources is here organized into an *int array* named *resource[10]* initialized at the start up to -1 in order to define them as free and ready to be acquired and locked by a client. When a certain resource (identified by the number of the array element) is locked by a client its value is set to *clientid*; in this way it is straight forward to know not only that a certain resource has been locked but also by which client.
Given these settings it is clear that an *acquire* operation on resource *resid* can be performed by the master server only if *resource[msg.resid-1] is equal to -1,* meaning only if the specified resource is free. Similarly, a *release* operation can be performed only if the specified resource has been previously locked by the client which issued the *release*

function. This means that the resource *resid* can be freed only if *resource[resid-1]* is equal to *msg.clientid*.
If the *acquire* or *release* operation can not be performed, the master server immediately responds to the client notifing the impossibility of processing the request. Else the server modifies its local resource and then contacts in succession all the replicated servers connected forwarding the *struct msg* received from the client and setting *msg.identity* to 2; in this way the replicas know the message is coming from the master.
Also the replicated servers verify their resources to check for the possibility of performing the requested operation. If this is feasible, the resource is immediately updated and a positive answer is sent back, else an error message is sent to the master. In the case in which the master receives from all the replicase a positive answers, then all the replicas have been correctly synchronized and the client is informed that the operation succeded.
Else, if one or more responses from the replicas are negative, a synchronization problem exists between the servers. As the master is assumed to always have a correct knowledge about the state of the resources, the synchronization of the replicated servers is forced by sending to all of them the entire resource array before transmitting a positive answer to the client. This methodology for correcting the errors in the replicated server works only in the case in which the dimension of the array of resources has a limited dimension but in the case in which the size is big a different approach is needed. An option is to directly access the problematic replica and solve the problem while a second one can be the implementation of a function that compares component by component the resources of master and replicas and corrects the mismatchs.
As the master server must be able to face multiple requests by different users, it is forked into two new processes every time a new connection is accepted. In this way, the child process can handle the received request while the parent one is immediately ready for a new incoming connection. Thanks to this strategy an arbitrary number of concurrent connection can be handled.
Because of the possible presence of multiple forked processes in the master server, the resource array is saved on a shared memory which is protected by a set of semaphores, one for each element of the array. This gives the possibility to the system to accept a big number of concurrent incoming connections without voluntarly drop any of them while guaranting mutually exclusive access to the shared elements of the resource. This means that different clients can not try to modify the same element at the same time. As soon as one of the forked master processes needs to handle a request and accesses the shared resource it pulls down the semaphore that protects the requested element and performs all the operations needed before pulling up again the semaphore and answering to the connected client.
In the unlucky case in which the master server crashes, one of the replicated servers is ready to take over as soon as the first client notifies the failure of the old master. The promotion to master of a replicated server works as follows.
When a client can not contact the master server on port *master_server_port* it scans iteratively the range of available ports looking for a replica, it connects to it and it sends the message *msg*. This replica knows right away that the master failed because it is receiving a message from a client. So it exits from the replica loop in *server.c* and enters the master one. First it scans the available ports for finding possible replicated servers and later processes the request from the client.
This is the brief description of our system which represent a high availability replicated locking service. The system can be accessed by an arbitrary number of clients, each one characterized by a client ID, that can connect to a master server in order to lock or realease a specified resource. This master server is backed by two replicated servers that are constantly synchronized in order to be ready to take over in the case of a failure of the master.