Distributed Systems Summative Assignment

000280744: jrvh15 - Z0966651

March 6, 2017

Part 1

Generic Workflow with Passive Replication

In passive replication, all requests made by the client are executed by a primary server, which, upon completion of the request, would then send an update message to its backups. The underlying principle behind passive replication is such that request invocations do not need to be coordinated between backup servers, since they do not execute any invocations themselves, but instead only apply changes produced by execution of invocations at the primary server.

The generic workflow can be described as follows:

1. All incoming requests are received by the front-end, which will then forward these requests to a primary server. Each request will contain a unique identifier, which will be used when coordinating the requests.
2. Once a request has been received, the primary server then decides upon the ordering of the request, relative to other requests. Each request is processed automatically in the order in which it was received. As a mechanism to handle message loss, the server also checks the unique identifier on each request; if the request has already been processed, then the server will retransmit its response to the front-end.
3. The primary server will then execute the request, storing its response.
4. A message is sent out to the replicas, which will then come to a consensus on the effect of the requests. For example, if the request is for an update, then the primary server will send out its updated state, its response and the unique identifier of the request that it received to all its backups, which, in turn, will store the information. The backups then send an acknowledgement to the primary server to confirm receipt of the information; if an acknowledgement is not received, then the information is re-sent. This process is known as **result propagation**.
5. The primary replica will then send its response to the front-end, which may carry out further processing before returning the response to the client.

Although this approach uses little processing power, compared to other replication techniques, passive replication tends to suffer from a high reconfiguration cost when the primary server fails.

Information Availability Guarantees

Passive replication guarantees information availability because, even in the event that the primary server fails, a backup server can take its place. As the primary server waits on acknowledgements from its replicas (and in cases where it does not receive the acknowledgement, it retransmits the data), the current

state of the system is guaranteed to have been propagated to all its replicas.

Other passive replication mechanisms can also be put in place to guarantee information availability. For example, with the **gossip architecture**, replicas share information periodically between themselves by sending each other gossip messages. These messages contain a log of past updates, allowing the network of replicas to achieve consensus on the latest state of the system.

Besides that, to communicate updates, while ensuring that the system is tolerant towards failure of the primary replica before, during and after applying updates, a form of group membership communication could be used; this communication system should include the following features[1] :

- An interface for membership changes
- A failure detector
- A notification system for membership changes
- Address expansion, to ensure that messages sent to the group reach all replicas

Through this, an extension of reliable multicast, i.e. view-synchronous communication, can be provided. A view is a list of the processes currently belonging to the group, and when the group membership changes, a new view is sent to all members. All messages that originate in a given view must be delivered (guaranteed through the use of *ack* messages) before a new one is delivered.

Although this form of communication is costly (and may require several rounds of communication for each multicast), because each replica has the same record of updates, this allows the system to tolerate n crash failures if $n + 1$ replicas are present. The view-synchronous semantics guarantee that either all the backups, or none of them, will deliver any given update before delivering the new view. Through this, the primary and surviving backups will all have to agree on whether an update has been processed, ensuring that all servers maintain the same state of the system and thus guaranteeing information availability.

Part 3

Resolving Failure

As a precautionary measure against system failure when the primary server crashes, the client is not connected directly to the primary server; instead, the client is connected to the front-end, which reroutes all requests that it receives from the client to the primary server. It does this, and provides transparency to the system, through **remote method invocation** (RMI). With the use of RMI, sockets to the servers do not have to be established manually by the client, and processes on the server-side, such as network, replication and failures are hidden from the user. In case the primary server fails, the front-end is responsible for locating a new server to forward requests to. It achieves this by maintaining a list in the background, through a daemon, which contains the **URI** (Uniform Resource Identifier) for each remote object registered to it, so that it can quickly locate the components of the system.

Each time a failure occurs, and the connection between the front-end and the primary server goes down, a backup server may take over as the primary; whenever this occurs, the URI stored by the front-end would have to be updated to ensure that the front-end is able to locate and connect to the new primary server.

[1] Rollins S. (2008) *Replication* [Online] Available at: `http://www.cs.usfca.edu/~srollins/courses/cs682-s08/web/notes/replication.html`

Passive replicas must always hold the latest state of the system, as this ensures that they are ready to take over the role of the primary server if a failure occurs. They do this by periodically requesting the current state of the system from the primary server; but to reduce network traffic, these requests are interleaved with short time intervals. Because the backup servers are constantly pulling data from the primary server, they would be able to automatically detect when the primary server goes down; they would then need to be able to dynamically locate the new primary server that they can reroute their requests to.

Both of the problems listed in the paragraphs above can be solved through the use of a proxy, and a nameserver, which behaves like a directory service. This allows for the primary server to be found using a name instead, hiding its specific URI and port number. By registering the URI of the primary server with a given name in the nameserver, you would only need the server to update the value registered with the nameserver when it takes over as the primary server. This has the added benefit of allowing you to locate free ports, instead of using a hardcoded value, when rebooting a server and reconnecting it to the system, as there may be situations in which a server would be unable to connect to the system using the same port as before, because it is still being held by the previous process that has not been terminated correctly.

When the server recovers and tries to reconnect to the system, it would be able to detect if the system currently has a primary server; if so, it would simply join the distributed system as a passive replica. Trying to reestablish the server as the primary requires too much overhead and, as such, will not be implemented.

Necessary Modifications for a Fault-Tolerant System

Thanks to the transparency provided by the front-end, no changes were necessary on the client program. On the front-end program, to ensure that the URI it holds is up-to-date and points to the currently running primary server, the program looks up the value of the URI in the proxy server before processing each request and sending any commands to the primary server.

On the server program, upon failure of the primary, an exception will be thrown, as the passive replicas are no longer able to pull data from the primary server to update their state. When this exception occurs, each server will look in the nameserver for any changes; if another passive replica has taken over the role of the primary, then the value in the nameserver would have been updated, and the server can now begin pulling data from the new primary. Otherwise, it will take over the role of the primary server itself.

When establishing itself as the new primary, it will remove the previous primary server's URI from the nameserver, and add its own URI, so that the remaining passive replicas would now be able to locate and pull data from it. To ensure that a newly recovered server does not establish itself as the primary, a restriction is placed upon the servers: if they do not carry any data, then they are not allowed to take over as primary. This is because a server that does not carry any data is highly likely to have just recovered from failure, and as such, does not hold the latest state of the system. Thus, the newly recovered server will wait for another passive replica to take over as primary, before pulling the latest state of the system from it.

When a failed server has recovered and attempts to reconnect to the distributed system, it will look in the nameserver for a primary; if one has been found (which would always be the case, unless there has been a total system failure), then it will simply rejoin the system as a passive replica, updating its own state by pulling data from the new primary server.