Tutorial
Distributed Systems (IN2259)

Exercises on Replication

Exercise 1 Primary-backup Replication

Figure 1.1 is a diagram showing the primary-backup model where one of the replicas is assigned the role of the primary replica. Every read and write request by clients is managed by the primary replica. Replication is performed eager the primary waits for acknowledgments from the backup replicas before sending a response back to the client.

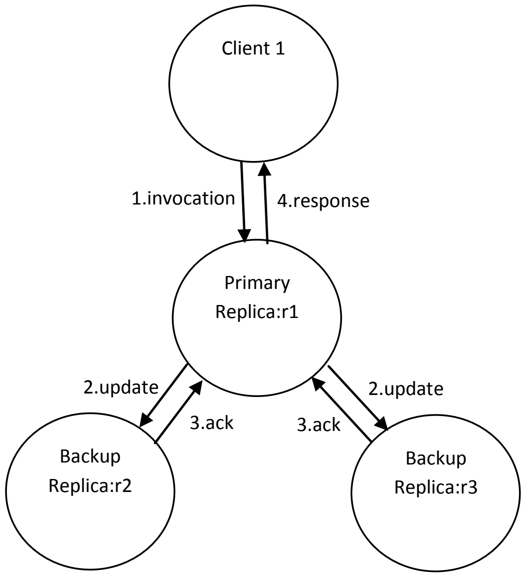

Figure 1.1: Primary-backup replication

Below you see two Java classes (cf. Listings 1.1 and 1.2) implementing the primary-backup protocol. Because the replicated objects can be assigned as primary or as backup, replicas are implemented by the same class. Provide the missing parts for the implementation in Listing 1.2. Put your focus on implementing the protocol logic.

```java
public class Client {

    private int clientId;
    private Replica primary;

    public void update(Object value){
        primary.update(clientId, value);
    }

    public void receiveAck(){
        System.out.println("Update successful");
    }
}
```

Listing 1.1: Class Client.

```java
public class Replica{

    private int replicaId;
    private boolean isPrimary;
    private Object value;
    private Client client;
    private List<Replica> replicas;

    public void update(int clientId, Object value){
        // TODO: Implement
    }

    public void receiveAck(int updateId, int clientId){
        // TODO: Implement
    }
}
```

Listing 1.2: Class Replica.

Exercise 2 Active Replication

Another approach to replication is active replication. Because there is no primary replica, the client has to communicate with all replicas. In order to coordinate the access to the replicas, *Total Order Broadcast* is used. Use the TO-Broadcast algorithm from an earlier assignment as a building block to implement both sides (client and replica) in pseudocode. Use the given interfaces.

```
#Client Interface
sendUpdate(value);
receiveAck(msgId, replicaId);

#Message−Layer Interface
sendTOBroadcast(message, recipients);
sendMessage(message, recipient)

#Replica Interface
receiveUpdate(msgId, clientId, value);
sendAck(clientId);
```

EXERCISE 3 Gossip-based Replication

Figure 3.2 shows a multi-primary lazy replication strategy using gossiping. Clients issue update requests (writes to the replica) and they can formulate queries (reads from the replica). Replicas periodically exchange gossip messages to synchronize their value.

In order to provide consistency, the clients and replicas keep track of clocks, $clock_c$ and $clock_{rep}$, where $clock_c$ is the client clock and $clock_{rep}$ is the replica clock. A clock shows the last system state observed by a particular client or replica. Each clock is a vector and the dimension of the vector is determined by the number of replicas in the system. The vectors are modified only according to the protocol specified below. Clocks can be used to generate timestamps for messages, such as ts_{update} for update messages and ts_{query} for query messages. Each replica maintains a queue of operations called $pending = \emptyset$, initially empty ($\emptyset$).

IDs for update messages are uniquely generated and monotonically increasing in the entire system. Each replica keeps track of a variable $last = \langle id, value \rangle$, which starts initially with value $\langle 0, 0 \rangle$. Queries also have an unique ID, which is tracked separately from the update messages.

In the example of Figure 3.2, there are three replicas, each with initial value 0. Each clock (i.e., clocks at all clients and replicas) is initialized to [0,0,0]. For instance, if the replica r_1 first receives an update message, the vector $clock_{rep}$ for r_1 is now [1,0,0].

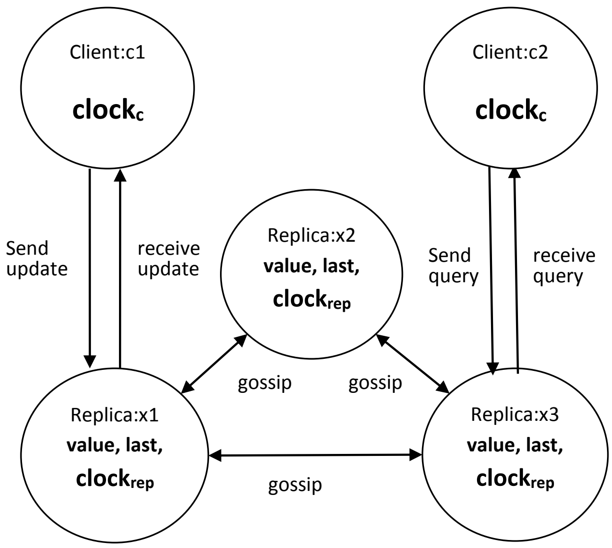

Figure 3.2: Lazy Replication over a gossip protocol.

In the following you can see the different message types and their corresponding protocol:

Query message

- A client sends a query message $\langle id, ts_{query} \rangle$ with $ts_{query} = clock_c$ to a replica r_i
- The replica r_i checks $ts_{query} \leq clock_{rep}$ (stability condition)
  - if $true$, the current value is read and sent back to the client, together with $ts_{val} = clock_{rep}$. The client sets $clock_c = ts_{val}$
  - if $false$, the operation is queued in $pending$ until gossip enables the condition $ts_{query} \leq clock_{rep}$

Update message

- The client sends an update message $\langle id, value, ts_{update} \rangle$ to a replica r_i, where $ts_{update} = clock_c$
- The replica r_i increments the i^{th} element of $clock_{rep}$

- The replica sends a message with ts_{ack}, which consists of ts_{update} merged with the i^{th} element of $clock_{rep}$. The client sets $clock_c$ to ts_{ack}
- The replica checks $ts_{update} \leq clock_{rep}$ (stability condition)
  - if $true$, the update is applied. If $id > last.id$, the replica sets $last = \langle id, value \rangle$ and value $= value$
  - if $false$, the operation is queued in $pending$ until gossip enables $ts_{update} \leq clock_{rep}$

Gossip message

- The replica r_i sends $\langle ts_{gossip}, gossip_{last} \rangle$ to another replica r_j, where $ts_{gossip} = clock_{rep}$, and $gossip_{last} = last$ (at r_i)
- The replica r_j updates its own $clock_{rep}$ with ts_{gossip} by replacing all entries l in $clock_{rep}$, where $clock_{rep}[l] < ts_{gossip}[l]$.
- If $gossip_{last}.ID > last.ID$, then update the value to $gossip_{value}$, and set $last = gossip_{last}$. (Note: any pending update or query operation which is now applicable should be processed in order using their ID)

The following message are exchanged between the processes. Iterate through the steps manually and show the state of each of each process after each step. You can use the table provided in Figure 3.3.

1. c_1 sends update $u_1(6)$ to r_1
2. c_2 sends update $u_2(4)$ to r_3
3. c_1 sends query q_1 to r_2
4. c_2 sends update $u_3(8)$ to r_1
5. r_3 sends gossip g_1 to r_1
6. r_1 sends gossip g_2 to r_2

		Initial	1	2	3	4	5	6
Client c1	$clock_c$	[0,0,0]						
Client c2	$clock_c$	[0,0,0]						
Replica r1	value $clock_{rep}$ last pending	0 [0,0,0] $\langle 0,0 \rangle$ $\emptyset$						
Replica r2	value $clock_{rep}$ last pending	0 [0,0,0] $\langle 0,0 \rangle$ $\emptyset$						
Replica r3	value $clock_{rep}$ last pending	0 [0,0,0] $\langle 0,0 \rangle$ $\emptyset$						

Figure 3.3: Gossip table.