**08/01/2020**

To ensure a more available system it is good practice to adopt software replication in located in different location and make this process the more transparent to the users. When using this technique, we assume there are two operations: one to “call the replica” and one to return the value queried.

Two properties must be respected:

* Atomicity: no two replicas handle different jobs
* Ordering: if two replicas handle two jobs, they handle them in the same order

There are two approaches:

1. Active replication

In this approach we have replicas of a software that are on the same level and have the same role; when a user queries something, every replica respond in order to accomplish the request; there is no leader so the user has not to wait for every replica to respond, it chooses the first that returned the value. If a replica fails, there is no problem since everyone has the same role.

1. Primary backup

In this case we have a *prim* replica is elected as the leader; when a user queries something, the prim receives, works, and update every replica to maintain consistency and waits for replicas to send ACKs (slow). In case of failure there are three case to consider:

1. Prim crashes after sending the returned data to user
2. Prim crashes after sending updates but before receiving ACKs
3. Prim crashes before sending updates

**2.**

**1. URB: validity, no duplication, no creation, uniform agreement**

m1, m2, m3, m4

m1, m2, m3, m4

m1, m2, m3

m1, m2, m3, m4

**2. RB: validity, no duplication, no creation, agreement**

m1, m2, m3, m4

m1, m2, m3, m4

m1, m2, m3, m5

m1, m2, m3, m4

**3. BEB: validity, no duplication, no creation**

m1, m2, m3, m4

m3, m2, m4

m2

m2, m3, m4

**Talking about ordering, the principal ordering properties are:**

**FIFO: (m3 -> m4)** In every run is respected

**CAUSAL (Fifo+ Local) (m3 -> m4 + m1 -> m2, m2 -> m4):** respected in every run

**TOTAL:** First run respect TO(UA), Second TO(NUA), Third no total order (no agreement)

**3. In questo caso il faulty non credo influisca dato che termina l’operazione prima di crashare, ma potrei sbagliarmi**

**1. Regular: termination, validity (no ordering constraints)**

Read1() -> 0,4,2

Read2() -> 4,2

Read4() -> 4,2,7

Read3() -> 2,7

Read5() -> 7

**2. Atomic: termination, validity, atomicity**

Read1() -> 0,4,2

Read2() -> 4,2 (if 1 read 0 or 4), 2 (if 1 read 2)

Read4() -> 4,2,7 (if 2 read 4), 2 or 7 (if 2 read 2)

Read3() -> 2,7 (if 4 read 4 or 2), 7 (if 4 read 7)

Read5() -> 7

**3.**

Read1()->0

Read2()->4

Read4()->2

Read3()->2

Read5()->7

**5. Total order, FIFO pp2p, Line topology, left-right, perfect oracle new\_right() new\_left()**

**Upon event <tob,Init> do**

Wait = False

Correct = PI

Receivedi = [NULL]N

Deliveredi = [NULL]N

For every p in Correct do

Initialize p.left and p.right

**Upon event <tob, Send| m> do**

Wait = true

**Trigger** <pp2p, Send | m> to self.right

**Trigger** <pp2p, Send | m> to self.left

**Upon event <tob, Deliver| m>**

**If** m not in Receivedi **then**

Receivedi = Receivedi u {m}

**Trigger** <pp2p, Send | m> to p.right

**Trigger** <pp2p, Send | m> to p.left

**Upon** m != 0 **and** #correct <= #(Receivedi | m in Receivedi) **and** wait = false **do**

For every i in Correct do

Deliveredi = Deliveredi u {m}

Receivedi = Receivedi \ {m}

Wait = False

**upon event <Oracle, Crash | p> do**

trigger <pp2p, Send | new\_right(p)> p.left

trigger <pp2p, Send | new\_left(p)> p.right

trigger <tob, Send| receivedp> to p.right

trigger <tob, Send| receivedp> to p.left

correct = correct \ {p}