The first of the alternatives to implement our replicated cloud file service is through a passive replication technique. Remember that this requires the use of a **reliable broadcast** primitive to guarantee the atomicity property, without which the service could not provide linearization. Unfortunately, network protocols do not provide such powerful communication primitives. Therefore, we will have to implement it from the available communication primitives, based on protocols such as UDP/IP or TCP/IP.

In this activity we will first explore the properties that must be satisfied by the reliable broadcast primitive, and then implement it.

What is there to be done

1. The teacher will propose a simple protocol of allegedly reliable broadcast (Section 3.4.2 of the Notes of Topic 3), characterized by the primitives:

   broadcast()

   deliver()

   Discuss in the group if this protocol provides the reliable broadcast necessary to support passive replication.

2. To implement a reliable broadcast primitive we must first know the conditions under which it will work and formalize the properties that it has to meet. The teacher will present the *system model* where the primitive will work (Section 3.4.1 of the Notes of Topic 3) and the formal properties (Section 3.4.2.1) that it must comply with. We are now in a position to formally analyze whether the protocol of the previous point provides reliable broadcast. Analyze what properties it meets and which it doesn't.
3. The teacher will present a new protocol (Section 3.4.2.1) based on two new primitives, implemented on top of the previous ones:

   R-broadcast()

   R-deliver()

   Does this protocol meet the properties of reliable broadcast? What is the price we pay for it?

4. On the basic implementation developed in Activity 3.1b, advance in the implementation of the passive replication technique to provide atomicity, introducing reliable broadcast in the communication between the primary and the secondary servers. Specifically, for this activity it is enough that you present the pseudo-code of *R-broadcast()* and *R-deliver()* in a brief report. Work also on the integration of these primitives in your replicated service as part of the Project of the course.

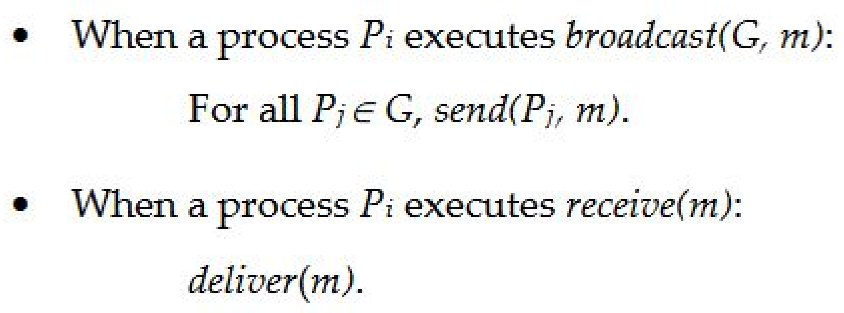
1.- The teacher will propose a simple protocol of allegedly reliable broadcast (Section 3.4.2 of the Notes of Topic 3), characterized by the primitives:

broadcast()

deliver()

Discuss in the group if this protocol provides the reliable broadcast necessary to support passive replication.

1.- In case of the protocol in the following image:



- When a process P_i executes *broadcast(G, m)*:

  For all $P_j \in G$, *send*(P_j, m).

- When a process P_i executes *receive(m)*:

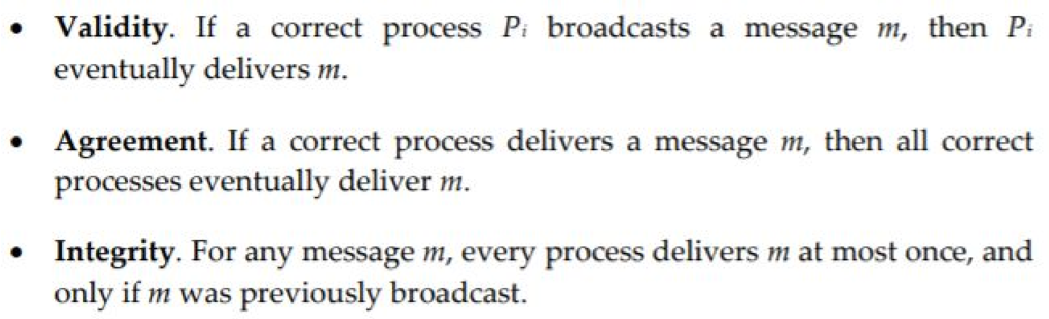
  deliver(m).

In this case, it provides unreliable broadcast. It can be seen that if the process that broadcasts the message fails while executing broadcast, some processes will receive the message and others will not.

2.- To implement a reliable broadcast primitive we must first know the conditions under which it will work and formalize the properties that it has to meet. The teacher will present the *system model* where the primitive will work (Section 3.4.1 of the Notes of Topic 3) and the formal properties (Section 3.4.2.1) that it must comply with. We are now in a position to formally analyze whether the protocol of the previous point provides reliable broadcast. Analyze what properties it meets and which it doesn't.

As said before, that implementation provides unreliable broadcast due to that if the broadcaster fails in the middle of the sending, only part of the group will get the message instead of receiving it by all the group, which means it is not reliable.

So, which are the properties that must be fulfilled to have a reliable broadcast?
These three:



- **Validity**. If a correct process P_i broadcasts a message *m*, then P_i eventually delivers *m*.
- **Agreement**. If a correct process delivers a message *m*, then all correct processes eventually deliver *m*.
- **Integrity**. For any message *m*, every process delivers *m* at most once, and only if *m* was previously broadcast.

The combination of these three properties is known as all-or-none property, which implies that a message broadcast to a group of processes, either reaches all the correct processes of the group or does not reach any correct process.

But, which if these properties are not fulfilled by the unreliable broadcast defined in the previous section? It fulfills:

- Validity, because if it successfully broadcasts the message, it will send it to its own process after, but won't deliver if it doesn't.
- Integrity, because for any message m, once m has been broadcasted, every process is going to deliver m at most once. As we have seen, it could be possible that the broadcast does not send m for all processes, so it is possible to deliver m once or none.

But does not fulfill the agreement property, because as we say it is possible that once a broadcast occurs followed by an error, not sending the message to everybody. This is why it does not fulfill the agreement property, because it says that if a correct process delivers message m, all correct processes will eventually deliver m, and we can't guarantee this with this broadcast algorithm.

3-4. The teacher will present a new protocol (Section 3.4.2.1) based on two new primitives, implemented on top of the previous ones:

R-broadcast()

R-deliver()

Does this protocol meet the properties of reliable broadcast? What is the price we pay for it?

How works the reliable broadcast algorithm?

INITIALIZATION: All process initialize an empty vector call "Received { }"

Pi, MAKES R-BROADCAST:

- Broadcast(g, m)   #Remember, Pi is included on g.

R-DELIVER ON THE REST OF PROCESSES, Pi

WHEN Pj EXECUTES RECEIVE(m)

```
If ( m is NOT inside Received {} ){ /* m not in the received list of Pj*/
        Received {} = Received {} + m;
        If ( Pi != Pj ){
                Broadcast(G, m);
        }
        R-deliver(m);
}
```

This protocol provides reliable broadcast because it meets the properties. When a correct process delivers m, in this algorithm previously has been forwarded to all processes. Due to this, all correct processes will end up receiving m and delivering it, which provides us agreement. So, in this case, we maintain the property of validity of the previous unreliable algorithm. Integrity is also ensured because a process only executes delivery the first time it receives the message (that's why we check the received list of the process), and only broadcast messages are delivered (we ensure with a vector that R-deliver only executes if it is not in the vector). and we add agreement because with this algorithm, as said before, if there is a process which is correct that delivers m, then it has been previously forwarded to all the processes, and, in that case, all the correct processes will end up receiving m, and therefore delivering it.

Cost: The price paid is a more complex algorithm. In this algorithm, unlike in the unreliable one, we add 2 conditional sentences:

"If (m is NOT inside Received {})"

"If (Pi != Pj)"

In the first conditional sentence, there is a list to be analyzed. We could say that the list has 'n' elements in it, elements that reference the messages received by the process Pj. For each of the n elements, we have to check if the element equals m. → **worst case scenario: compare m to the current element n times.**

The second condition is a simple comparison.

If we state that a comparison operation, deliver and broadcast has cost 1, then, the cost in the worst case would be **n**.