Fault Tolerance

Group Based Communication

Pedro F. Souto (`pfs@fe.up.pt`)

May 27, 2021

Groups

Observation The concept of group of processes is recurrent in distributed systems

1. IP multicast groups
2. Garcia-Molina's Invitation Algorithm uses process groups for leader election in asynchronous systems
3. The state machine replication approach uses a group of processes to provide a fault-tolerant service by masking process failures

Types

Static The group membership does not change (SMR with Paxos)

Dynamic The group membership changes (Invitation Algorithm)

- As processes join/leave the group voluntarily
- As processes fail/recover

Observation Use of process groups is more convenient when integrated with group communication

- I.e. multicast

Dynamic Group Communication

Observation SMR with Paxos relies on total order multicast in a static group

- Implemented on top of the Paxos algorithm

Dynamic Group Communication relies on two services

Group membership which provides information on which processes belong to the group

Group communication which provides group-based messaging services

- More precisely reliable multicast communication

Group Membership (1/2)

Basic Service Outputs a **view** of the group

- Each view is composed of a set of processes
- Each view has an identifier, v_i
  - Allows to distinguish among groups with the same composition
  - An alternative approach, is to ensure that a process gets a new identifier every time it joins the group
- If a process has a view:
  - All processes in that view must have agreed to join the view

Type

Primary component Ensures that at "any time" there is at most one view

- More precisely, the views are totally ordered
- This is achieved by requiring a view to comprise a majority of the processes

Partitionable Allows the existence of more than one view at any time

Group Membership (2/2)

Interface

`join/leave` Used by processes to request joining/leaving groups

`new-view` Used to notify a view change, either in response to:

- Voluntary requests (join/leave)
- Unexpected events (failure or recovery of processes)

Failure Detection needs not be reliable

- A process may be expelled from a group by mistake
  - E.g. because of transient communication problems
- If churn is too high, progress may be affected

Reliable Broadcast

Question What does it mean to **reliably broadcast** a message ?

Assuming static groups first

- **Validity** If a correct process broadcasts message *m*, then all correct processes in the group deliver *m* **eventually**
- **Agreement** If a correct process delivers message *m*, then all correct processes in the group deliver *m* **eventually**
- **Integrity** A process delivers message *m* at most once, and only if it was previously broadcasted by another process
  - In the case of **closed groups**, the broadcaster must be a group member too.

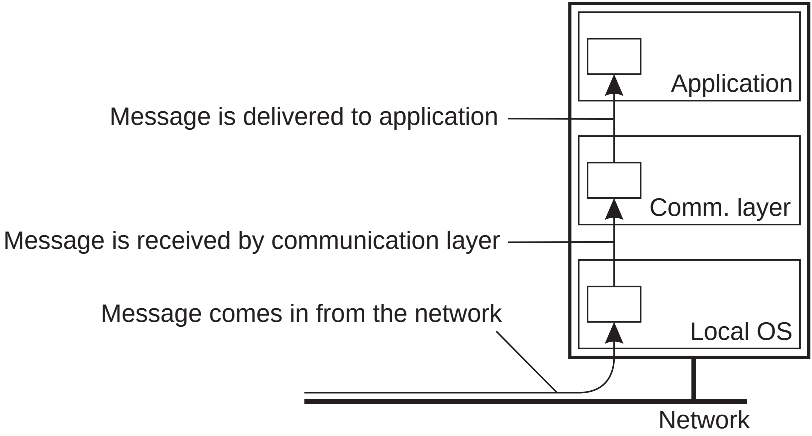
Failure assumptions

- Processes may fail by crash and may recover
- Any network partition **eventually** heals

What if the group is dynamic?

Delivering vs. Receiving

- **Delivering** a message is different from **receiving** it:



Reliable Broadcast in Dynamic Groups (1/2)

- Simplify first ... assume **closed group communication**
- Each multicast message is associated with a group view
- **First attempt**: reliable broadcast properties

  Validity If a correct process broadcasts message m, then all correct processes in the group deliver m **eventually**

  Agreement If a correct process delivers message m, then all correct processes in the group deliver m **eventually**

  Integrity A process delivers message m at most once, and only if it was previously broadcasted by another process

  must hold only for members of that group view:

  Validity If a correct process broadcasts message m in a view, then all correct processes in **that view** deliver m

  Agreement If a correct process **in view** v delivers message m, then all correct processes in that view deliver m **in view** v

  Integrity No need to change

- But . . . we are not there yet.

Reliable Broadcast in Dynamic Groups (2/2)

Challenge **Validity** and **Agreement** require all correct processes to deliver a message. This **conflicts** with:

Groups as a mean to keep track of the state of processes in the system in a coordinated way;

Impossibility of distinguishing between:

- Slow processes;
- Failed processes;
- Unreachable processes.

i.e. accurate and complete process failure detection in an asynchronous system

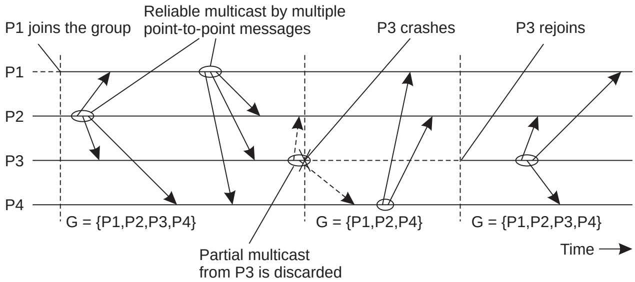
Scenario Assume a clean network partition

- Consider a sender in one of the parts
- Consider the processes in another part

View/Virtual Synchronous Multicast

Virtual Synchrony If processes p and q change from view V to view V', then they deliver the same set of messages in view V

- This is a variation of **agreement**
- It can also be seen as an **atomicity** property
  - Either a message is delivered to all processes or ...



Self Delivery If a correct process p broadcasts message m in view v, then it delivers m in that view

- This is a variation of **validity** and precludes trivial solutions such as a protocol that never delivers messages, even ...

View Synchronous Multicast: Implementation (1/4)

Assumptions/Model

Point-to-point channels Communication is via point-to-point channels

Reliable channels If the processes at the ends of a point-to-point channel are correct, then a message sent in one end will be delivered at the other

- It is well known how to achieve this by acknowledgments/retransmissions
  - As long as communication failures are not "too frequent"

FIFO channels I.e. messages are delivered in order

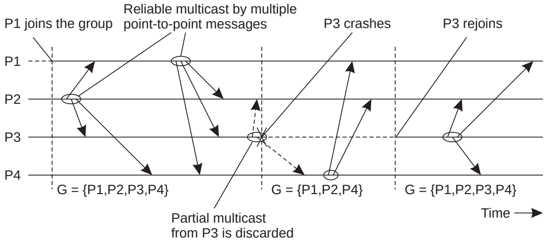
- There are well known techniques

Crash-failures Processes fail by crash only

View Synchronous Multicast: Implementation (2/4)

Problem What if the sender fails in the middle of a multicast

- In this event, some processes may **receive** the message whereas others do not



Solution Two alternatives:

1. Deliver the message only if all correct processes receive it
   - Increases the delivery latency
2. Deliver the message immediately
   - Upon a view change, processes that survive the current view must send each other the messages they have delivered that may have not been received by other view members

View Synchronous Multicast: Implementation (3/4)

Definition A **message *m* is stable** for process *p*, if *p* knows that all other processes in the view have received it

Idea

- Keep a copy of the messages delivered until they become stable
- Upon a view change:
  1. Resend all non-stable messages to the remaining processes
  2. Wait for the reception of non-stable messages from other processes
     - and deliver them if they have not been delivered yet
  3. Change to the new view

  Alternatively a process may be elected as **coordinator**

  1. each process sends its non-stable messages to the coordinator
  2. the coordinator then broadcasts each of them

  Observation Election does not require extra messages

  - given that the group members are known, we can use some *a priori* rule – e.g. the process with the smallest identifier

View Synchronous Multicast: Implementation (4/4)

Problem How does a process know that all non-stable messages have been received?

Solution

- Each process, sends a FLUSH message, after sending all non-stable messages
- Upon receiving a FLUSH message from each process that is in both the current and the next view, a process may change its view

Problem What if the processes crash during this protocol

Solution Processes must start a new view change

Problem How do you ensure progress?

Order in Multicast Communication

Observation Like in unicast communication, order is orthogonal to reliability

- Must be careful in the definitions so that we keep them that way

Unordered no guarantee on the order in which messages are delivered

FIFO if messages m_1 and m_2 are sent by the same process in that order, then if a receiver delivers both of them, it must deliver them in that order

Causal if message m_2 "causally depends" on message m_1, then if a receiver delivers both messages, it must deliver m_1 first

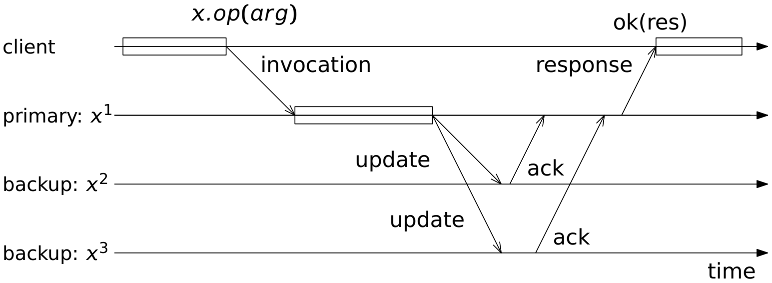
Total if process p delivers message m_2 after message m_1, then if process q also delivers both messages, it must deliver them in that order

Further Reading

- Tanenbaum and van Steen, *Distributed Systems*, 2nd Ed.
  - Section 8.4: Reliable Group Communication
  - Section 7.5.2: Primary-Based Protocols
- K. Birman, A. Schiper and P. Stephenson, *Lightweight Causal and Atomic Multicast*, ACM Transactions on Computer Systems, Vol. 9, No. 2, Aug. 1991, Pages 272-314

Primary Backup Replication: Basic Algorithm

- One server is the **primary** and the remaining are **backups**
- The clients send requests to the primary only
- The primary executes the requests, updates the **state** of the backups and responds to the clients
  - After receiving enough acknowledgements from the backups
- Essentially, the primary orders the different client requests
- If the primary fails, a **failover** occurs and one of the backups becomes the new primary.
  - May use leader election



Source: Guerraoui96

Primary-Backup: Failure Detection and Failover

Failure Detection

How? Usually sending:

either, I'M ALIVE messages periodically
or, acknowledgment messages

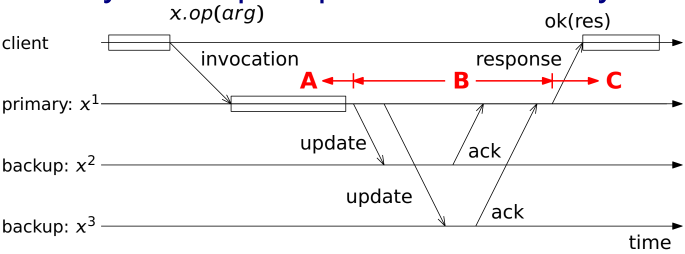
How reliable is it?

- It isn't, unless the system is synchronous ...

Failover

- At least, "select" new primary

Primary Backup Replication: Primary Failure (1/2)



Source: Guerraoui96

What if the primary fails?

Depends when the failure occurs

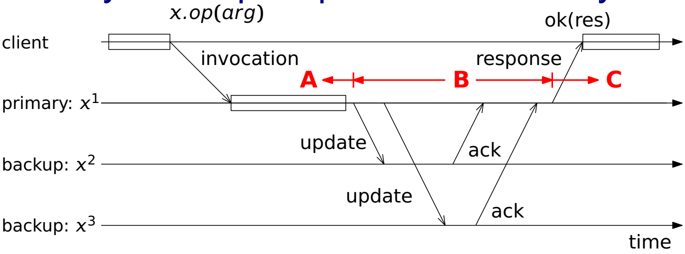
Primary crashes after sending response to client (C)

- Transparent to client
  - Unless response message is lost, and primary crashes before retransmitting it (case B)

Primary crashes before sending update to backups (A)

- No backup receives the update
- If client retransmits request, it will be handled as a new request by the new primary

Primary Backup Replication: Primary Failure (2/2)



Source: Guerraoui96

Primary crashes after sending update (and before sending a response) (B). Need to consider different cases:

No backup receives update as in case A

All backups receive update

- If client retransmits request, new primary will respond
  - Update message must include response, if operation is non-idempotent

Some backups, not all, receive update

- Backups will be in inconsistent state

Must ensure update delivery atomicity

Primary Backup Replication: Recovery

Problem when a replica recovers, its state is stale

- It cannot apply the updates and send ACKS to the new primary

Solution Use a **state transfer** protocol to bring the state of the backup in synch with that of the primary

State transfer protocol Two main alternatives

Resending missing UPDATES

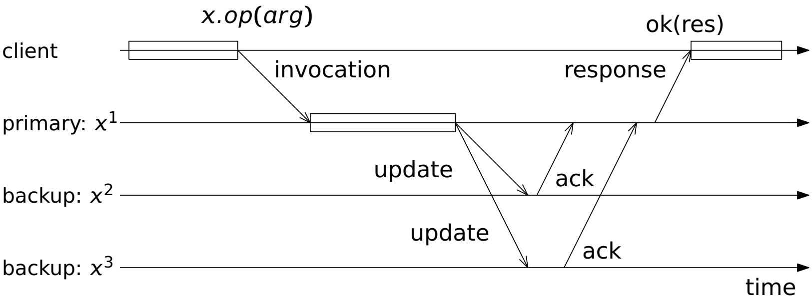
Transferring the state itself

In both cases, the recovering replica can:

- Buffer the UPDATE messages received from the primary
- Process these UPDATES once its state is sufficiently up to date, i.e. reflects all previous UPDATES
  - Update the local replica
  - Send ACK to the primary

Similar issues arise with SMR

Primary-backup fault-tolerance



Question What's the fault-tolerance?

Answer It depends on the failure model

Crash-failure Two faulty replicas

- In general, $n - 1$

Omission In this case, there is a need for a majority to prevent the existence of more than one primary at some time

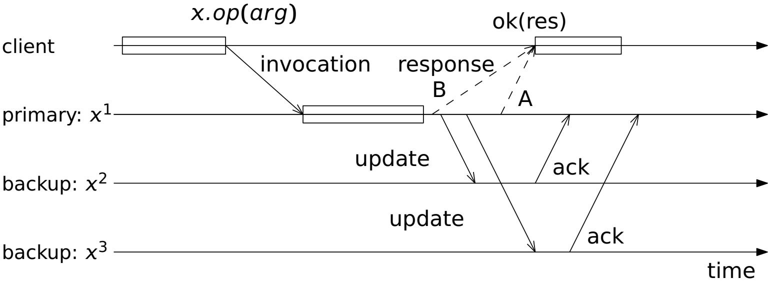
Primary Backup Replication: Non-blocking Algorithm

Observation Waiting for backup acknowledgements increases latency

Solution Primary may send response to client before receiving `ack`'s from backups (A)

Question 1 What is the trade-off?

Question 2 What about sending response before the update to the backups (B)?



PBR Implementation with View Synch. Mcast (1/4)

Idea The primary can be determined from the view membership without further communication

- The primary sends the updates to the backups using view synchronous multicast (ensures message delivery atomicity)
  - What about order?

Primary

1. Upon receiving a request, the primary:
   1.1 executes it
   1.2 generates an **update** representing the change in the state
   1.3 multicasts the UPDATE to the current view, v_i
     - must include **request id** and **client response**, unless ...
2. Upon receiving an ACK from f backups
   - sends the reply back to the client

Backup

1. Upon receiving an UPDATE, the backup:
   1.1 updates its state accordingly
   1.2 sends its acknowledgement to the sender (via multicast?)

PBR Implementation with View Synch. Mcast (2/4)

How does VSynch Mcast help?

Answer

- Upon failure of the primary generates a new view, and "elects" a new primary
  - A new view is also generated upon
    - either a failure
    - or the recovery

    of a backup
- Ensures UPDATE delivery atomicity to replicas that move from one view to the next
  - For every UPDATE, either all replicas that move from that view to the next deliver the UPDATE or none does it.

PBR Implementation with View Synch. Mcast (3/4)

What VSMcast does not address?

Upon a view change new members must synchronize their state

- Still need a state transfer protocol

At most-once semantics i.e. process a request no more than once

- TCP is no solution. Why?
- If uncertain:

1. Cache the response
   - Need to do it to recover from lost messages anyway
   - But backups also need to know the response
2. If the client retransmits the request, resend response

PBR Implementation with View Synch. Mcast (4/4)

The devil is in the details VSC simplifies significantly

- But replica reintegration is not trivial, even with VSC
- Paxos also glosses over the issue of recovery

For a detailed discussion of recovery somewhat application dependent

- B. Liskov, *From Viewstamped Replication to Byzantine Fault Tolerance*, Ch. 7 of LNCS 5959
  - **Viewstamped replication** is an algorithm that
    - Uses the concept of view, like VSC
    - But is more asynchronous, like Paxos

Further Reading

- van Steen and Tanenbaum, *Distributed Systems*, 3rd Ed.
  - Section 7.5.2: Primary-Based Protocols
- R. Guerraoui and A. Schiper, *Software-based replication for fault-tolerance*, in IEEE Computer, (30)4:68-74 (April 1997)(in Moodle)
- R. Guerraoui and A. Schiper, *Fault-Tolerance by Replication in Distributed Systems - A Tutorial* International Conference on Reliable Software Technologies (Invited Paper), Springer Verlag (LNCS1088), 1996