Reliable Client-Server Communication

• A communication channel may lose and/or corrupt messages
• How to handle communication failures?
  – Use a reliable transport protocol (e.g., TCP) or handle at the application layer
• Techniques for reliable communication
  – Use redundant bits to detect bit errors in packets
  – Use sequence numbers to detect packet loss
  – Recover from corrupted/lost packets using acknowledgements and retransmissions

RPC Semantics in the Presence of Failures

Five types of failures can occur in RPC
1. Client cannot locate server
   – Solution: The RPC system informs the client of the failure
2. Server crashes after receiving a request
3. Client request is lost
4. Server response is lost
5. Client crashes after sending a request
Server Crashes after Receiving a Request

- The client cannot tell if the crash occurred before or after the request is carried out
- Three possible semantics
  - At-least-once: keep trying until a reply is received
  - At-most-once: give up immediately and report back failure
  - Exactly once: desirable but not achievable

---

A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

---

Lost Request/Reply Messages

- Client waits for reply message, resents the request upon timeout
  - Problem: Upon timeout, client cannot tell whether the request was lost or the reply was lost
- Client can safely resend the request for **idempotent** operations
  - An **idempotent** operation is an operation that can be safely repeated
  - E.g., reading the first line of a file is idempotent, transferring money is not
- For nonidempotent operations, client can add sequence numbers to requests so that the server can distinguish a retransmitted request from an original request
  - Server need keep track of the most recently received sequence number from each client
  - Server will not carry out a retransmitted request, but will send a reply to the client
Client Crashes after Sending a Request

- What happens to the server computation, referred to as an orphan?
  - **Extermination:** Client explicitly kills off the orphan when it comes back up
    - Client stub makes a log entry on disk before sending an RPC message
  - **Reincarnation:** When a client reboots, it broadcasts a new epoch number; when server receives the broadcast, it kills the computations that were running on behalf of the client
  - **Expiration:** each RPC is associated with an expiration time T
    - The call is aborted when the expiration time is reached
    - If RPC cannot finish within T, the client must asks for another quantum
    - If after a crash the client waits a time T before rebooting, all orphans are sure to be gone

Reliable Multicasting

- Reliable multicasting means that
  - If all processes are nonfaulty, every message should be delivered to each group member
  - In the presence of faulty processes, every message should be delivered to each nonfaulty group member
Reliable Multicasting When Processes are Nonfaulty

- **Assumption:** the underlying communication system offers only unreliable multicasting
- A solution:
  - Sender assigns a sequence number to each message
  - When sender sends message M, it stores M in a history buffer
  - Each receiver acknowledges the receipt of M, or requests retransmission when noticing message loss
  - Sender removes M from history buffer when everyone has returned an acknowledgement (ACK)
- This solution does not scale: The sender may be swamped with ACKs when the number of receivers is large

A simple solution to reliable multicasting when all receivers are nonfaulty. (a) Message transmission. (b) Reporting feedback.
Scalable Reliable Multicasting

- A solution: Receivers send only negative ACKs (i.e., NACKs) when they are missing messages
  - How long should sender keep a message in buffer?
  - The sender may receive a large number of NACKs
- A scalable solution using feedback suppression
  - When a receiver missed a message, it multicasts a NACK to the rest of the group
  - A receiver schedules its NACK with some random delay, and suppresses it when observing another NACK

![Diagram of multicast feedback suppression](image-url)