RPC Semantics in the Presence of Failure

UNIT-2-Part: RPC Semantics in the Presence of Failure

Five different classes of failure that can occur in RPC systems:

- The client is unable to locate the server.
- The request message from the client to the server is lost.
- The reply message from the server to the client is lost.
- The server crashes after receiving a request.
- The client crashes after sending a request.

Client cannot locate the server

- If a client can't locate the server, this might be due to the server being down or a mismatch between client and server versions. For example, if the server interface evolves while the client remains outdated, the client may fail to connect when eventually run.

- **Error Reporting Mechanisms**
- **Simple Return Values:** Procedures can return a specific error code, like -1, to signal failure. However, this is not always reliable, as -1 might also be a legitimate return value (e.g., the sum of 7 and -8).

- **Global Error Variables:** In systems like UNIX, a global variable (e.g., errno) can be set to indicate the error type, such as "Cannot locate server."

- **Limitations**
- The return value approach is limited because it doesn't apply universally across different scenarios.
- Raising exceptions is another method, but not all programming languages support exceptions.

Lost Reply Messages

- **Lost replies are tricky.** The client could retry after a timeout, but it's unclear whether the server is slow or if the request or reply was lost. Retrying may not always be safe, especially with non-idempotent operations like money transfers, where retransmitting could cause duplicate transactions.

- **Ensuring Safe Retries**
- Idempotent Requests: Ideally, structure requests to be idempotent, meaning they can be repeated without adverse effects. For example, retrieving a file segment can be safely retried.

- **Sequence Numbers:** Assign a unique sequence number to each request. The server tracks these numbers to differentiate between new requests and retransmissions, ensuring that duplicate operations are not performed.

- **Distinguishing Initial Requests from Retransmissions:** Use a bit in the message header to mark whether a request is an initial one or a retransmission. Initial requests are always safe to execute, while retransmissions might need careful handling.

Server Crashes

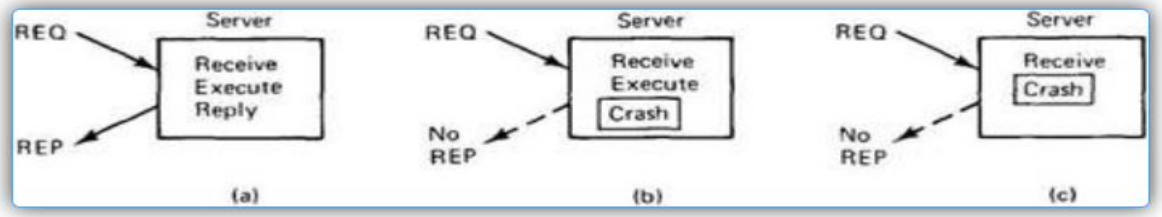

Fig. 2-24. (a) Normal case. (b) Crash after execution. (c) Crash before execution.

When a server crashes during an RPC, handling the situation depends on the timing of the crash:

- **Request Carried Out, Reply Sent (Fig. 2-24a):** The process completes successfully.
- **Request Carried Out, Server Crashes Before Reply (Fig. 2-24b):** The client doesn't know if the operation succeeded, and may mistakenly retry, leading to duplicate operations.
- **Server Crashes Before Carrying Out Request (Fig. 2-24c):** The request can be safely retried.

- The challenge is that the client's kernel cannot differentiate between scenarios (b) and (c), only knowing that the server is unresponsive.

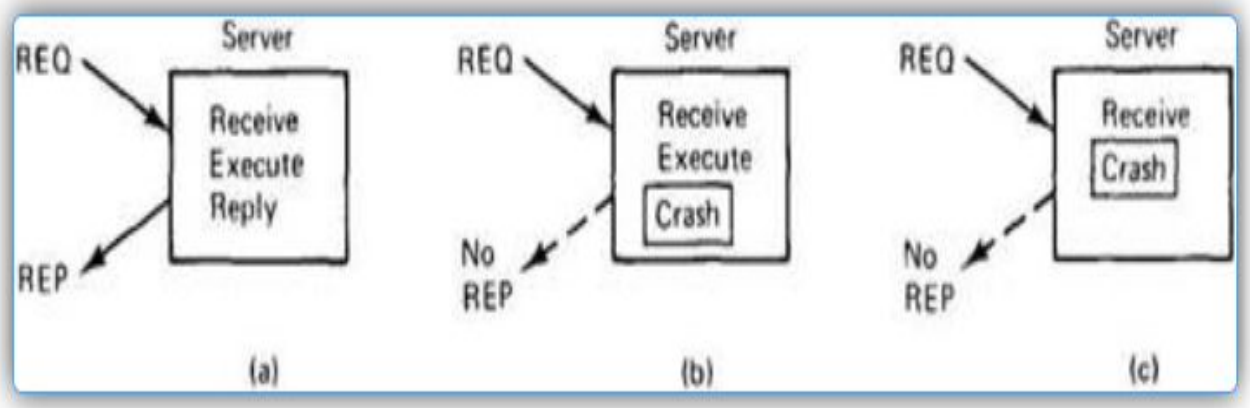

Fig. 2-24. (a) Normal case. (b) Crash after execution. (c) Crash before execution.

- **Three Approaches to Handling Server Crashes**
- **At Least Once Semantics:** Retry the operation until a reply is received, ensuring the operation is carried out at least once but potentially more.
- **At Most Once Semantics:** Immediately report failure, ensuring the operation is carried out at most once, possibly not at all.
- **No Guarantees:** Make no promises; the operation may have been executed any number of times, but this approach is easy to implement.

Client Crashes

- When a client crashes after sending an RPC, but before receiving a reply, the server may continue processing, creating an "orphan" computation that wastes resources and can cause confusion.

- **Solutions for Managing Orphans**
- **Extermination:** The client logs each RPC before sending it. Upon reboot, the log is checked, and orphans are explicitly killed. However, this method is expensive and may fail if orphans create further descendants (e.g., "grandorphans") or if network partitioning occurs.

- **Reincarnation:** Time is divided into epochs. Upon reboot, the client broadcasts a new epoch, causing all remote computations to be killed. Orphans that survive due to network partitioning are easily detected by their outdated epoch number.

- **Gentle Reincarnation:** Similar to reincarnation, but less aggressive. Machines try to locate the owner of remote computations before killing them. If the owner can't be found, the computation is terminated.

- **Expiration:** Each RPC is given a fixed time (T) to complete. If it cannot finish, it must request more time. After a crash, the server waits T before rebooting, ensuring orphans are gone. The challenge is setting an appropriate value for T.

- **Drawbacks of Orphan Management**
- Unforeseen Consequences: Killing orphans can leave issues like lingering file locks or initiated processes that remain even after the orphan is terminated.

Lost Request Messages

- The second item on the list is dealing with lost request messages.
- This is the easiest one to deal with: just have the kernel start a timer when sending the request. If the timer expires before a reply or acknowledgement comes back, the kernel sends the message again.
- If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine.
- Of course, so many request messages are lost that the kernel gives up and falsely concludes that the server is down, in which case we are back to "Cannot locate server."


Q&A

Thank You!