Design Document: At-Most-Once Client-Server Protocol

Preface

- **Goals:** Building a reliable request-response protocol between a single client and a single server over a network. The system must guarantee **at-most-once (AMO)** semantics, which ensures that a client's command is executed by the server exactly once, even with some network failures.
- **Desired Fault Model:** The protocol is designed to tolerate an unreliable network where messages can be arbitrarily **dropped, delayed, duplicated, or reordered**. But it is assumed that nodes (the client and server) themselves do not crash.
- **Challenges:** The core challenge is maintaining the at-most-once guarantee despite network faults. There are few examples of malfunctioning without AMO semantics:
    - If a Request is dropped, the client's command is never executed.
    - If a Reply is dropped, the client doesn't know the command succeeded and may re-send it, causing the server to execute it twice.
    - If a Request is duplicated, the server might execute the same command multiple times.
- **Assumptions:**
    - Nodes do not crash or fail; all faults occur within the network.
    - The client sends a single command and waits for a corresponding result before sending a new command.
    - The underlying application logic (e.g., the Key-Value Store) is deterministic.

Protocol

- **Kinds of Node:**
    - **Client:** A node that sends commands to the server and waits for results.
    - **Server:** A node that executes commands from the client and sends back results.
        - **Roles:** The nodes have fixed kinds. There are no temporary or changing roles.

- **State at Each Kind of Node:**
    - **Client State:**
        - serverAddress: (Configuration) The network address of the server.
        - sequenceNum: (Integer) The sequence number for the *next* command to be sent. Initialized to 0 and is monotonically incremented for each new command.
        - pendingRequest: (Optional Request Message) The last request sent to the server for which a reply has not yet been received. It is absent initially.
        - result: (Optional Result Message) The result received from the server corresponding to the pendingRequest. It is absent until a valid reply arrives.
    - **Server State:**
        - application: The underlying application which commands are executed.
        - lastSequenceNum: (Map: Client Address → Integer) Stores the sequence number of the last successfully executed command for each client. Initialized to empty.
        - lastResult: (Map: Client Address → Result) Stores the result of the last successfully executed command for each client, used for retransmitting replies. Initialized to empty.
- **Messages:**
    - **Request**
        - **Source:** Client
        - **Destination:** Server
        - **Contents:**
            - command: The application-level command to execute.
            - clientAddress: The address of the sending client.
            - sequenceNum: The unique sequence number for this command.

- **When is it sent?**
    - sendCommand method is invoked.
    - ClientTimer fires before a reply is received.
- **What happens at the destination (Server)?**
    - Receive a Request, server compares to sequenceNum to lastSequenceNum value with clientAddress key.
    - (sequenceNum < lastSequenceNum) already been superseded. Server Ignores it and sends no reply
    - (sequenceNum == lastSequenceNum) retransmission of the recent command. Server retrieves the cached lastResult and sends it back
    - (sequenceNum > lastSequenceNum) the Request is new. The server executes the command and updates lastSequenceNum and lastresult and sends the result as the reply.

- **Reply**
    - **Source:** Server
    - **Destination:** Client
    - **Contents:**
        - result: The result of the executed command.
        - sequenceNum: The sequence number of the command this reply corresponds to.
    - **When is it sent?** In response to a Request that is not stale.
    - **What happens at the destination (Client)?**
        - Receive Reply, checks sequenceNum matches sequenceNum of its pendingRequest.
        - (Match) Client sotres the result, clear pendingRequest, and cancel retransmission timer
        - (No match) reply is old or unknown. Ignores it.

- **Timers:**
    - **ClientTimer**
        - **Node:** Client
        - **Contents:** sequenceNum (the sequence number of the request this timer is for).
        - **When is it set?** When the client first sends a Request.
        - **What happens when it fires?** This timer follows the **discard pattern**.

            The client checks if it is still waiting for a reply for the sequenceNum stored in the timer.

            If a reply has already been received: The timer is for a completed request. It is ignored and not reset.

            If a reply has not been received: The client re-sends the pendingRequest to the server and resets the ClientTimer.

Correctness/Liveness Analysis

Failure Case	Request Message Handling	Reply Message Handling
Drop	The client's ClientTimer will fire, triggering retransmission. The server will eventually receive a request. Liveness maintained	The client's ClientTimer will fire, triggering a retransmission of the Request. The server will then send a new Reply (from its cache).
Duplicate	The server receives a duplicate Request. It checks the sequence number. If it's the most recent, it resends the cached Reply. If it's stale, it's ignored. Preserves correctness.	The client receives a duplicate Reply. Since it has already received and processed the first reply for that sequence number, it will ignore all subsequent duplicates.
Delay	A long delay is equivalent to a drop followed by a duplicate. The client retransmits on timeout. When the delayed Request arrives, the server's sequence number logic correctly identifies	A long delay is equivalent to a drop. The client retransmits the Request. It will accept the first valid Reply it receives and ignore the delayed one, as it will no longer be waiting for that sequence number.

	it as either a retransmission or stale. Prevents re-execution.	
Reorder	If Request A was sent and then B but server receives B, A order, server will discard B and execute A. As the timer for A expires on client side, it will eventually send back the request again. The first arrived B will never be executed if A hasn't arrive first using sequence number	With same logic, client side will handle the reply with sequence number, ignoring the first arrived later request's reply, eventually guaranteeing the correct order after few retransmission

- **Node Disappearance/Partition:**
    - If the **server** disappears, the client will repeatedly time out and retransmit its request. It will not make progress (a liveness failure), but it will not get an incorrect result. Correctness is preserved.
    - If the **client** disappears, the server simply stops receiving requests and does nothing.
    - When the partition is resolved, the client's ongoing retransmissions will eventually reach the server, and the protocol will resume normal operation.

Conclusion

- **Goals Achieved:** The protocol successfully achieves at-most-once semantics.
    - **Client-side retries** ensure liveness in the face of dropped messages.
    - **Server-side sequence number checking** ensures correctness by filtering duplicates and stale requests, preventing re-execution.
    - **Server-side result caching** ensures that replies to retransmissions can be sent efficiently without re-computation and the server state is not disturbed (idempotent).
    - **Client-side reply filtering** ensures that only relevant and timely results are accepted.
- **Limitations:**
    - The protocol does not handle node crashes.

- Liveness is not guaranteed if the network is partitioned permanently.
- The design is specific to a single-client, single-server scenario.