SOLUTION NOTES

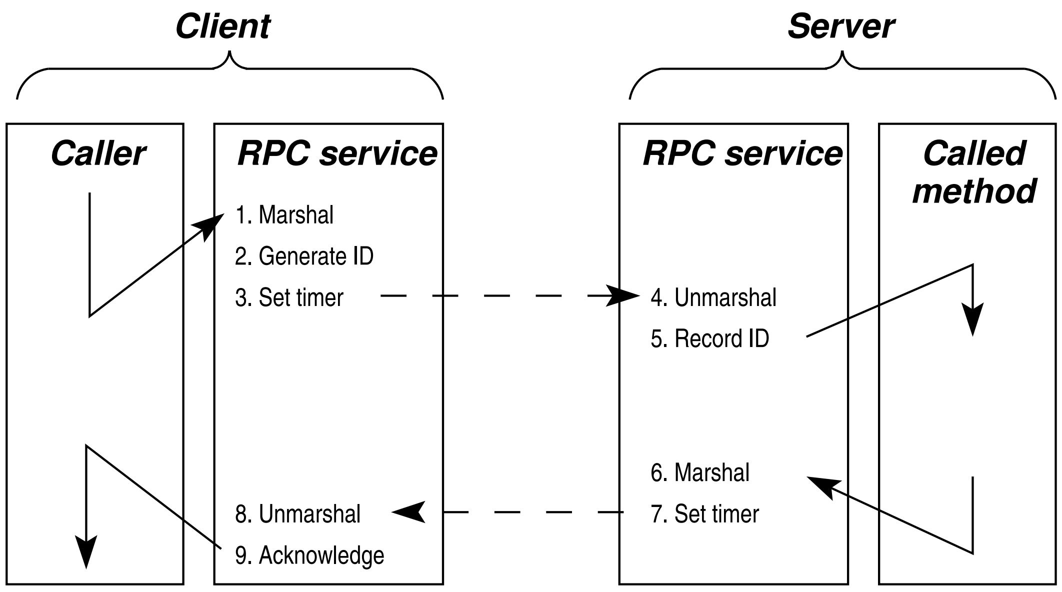
Concurrent Systems and Applications 2003 Paper 6 Question 4 (TLH)

(*a*) *Lecture 5, reflection and serialization* and *lectures 16-18, communication between address spaces.* The basic scheme is summarised at the end of Lecture 18, showing the use of identifiers to distinguish individual RPC requests and the use of retransmissions and the retention of state at the server to enable exactly-one semantics.

(*i*) Marshalling can be performed most easily by using serialization to convert the `Client` and `Appointment` objects to a form that can be embedded in packets sent over the network. For instance by instantiating an `ObjectOutputStream` over a `ByteArrayOutputStream`. The client and server will need to agree on the order in which the parameters are marshalled, on the definitions of the classes involved and on a further way of distinguishing between the different RPC operations that the server exposes.

(*ii*) The client generates a fresh ID for each RPC request that is made. It sets a time when it makes the request and will re-send it, using the same ID, if the timer expires without a response having been received. When the server receives a request with an ID it has not seen before then it records the ID, calls the appropriate function, records the result against the ID, marshalls that result and sends it back to the client. At that point the server sets a timer and will re-send its reply if it has not been acknowledged. If the server receives a duplicate ID then it either (i) ignores it if the request is still in progress or (ii) re-sends a response that it recorded earlier. Once the client has acknowledged a response (for instance, by making a subsequent request in a simple synchronous system) then the server can remove its record for that request.

(*iii*) In addition to the "main" threads used by the client application, it may use a further thread to manage retransmissions (in particular of lost acknowledgements). A simple server may use a single thread to deal with incoming requests and a further thread to deal with retransmissions (in particular of lost replies). While this simplifies implementation, it limits concurrency and may lead to deadlock if a server `S` makes RPC requests on a server `T` and `T`, in turn, makes invocations on `S`. Another option is to create separate threads at stage 5 (overleaf) for each distinct RPC request.



(*b*) *Lectures 20-21, isolation, serializability, TSO.* In *strict isolation*, transactions see only the effects of committed transactions. [In contrast, non-strict isolation relaxes this during a transaction's execution, but requires it to have only seen committed updates if it itself is to be allowed to commit]

Timestamp ordering associates a timestamp with each shared object; in this case associating a timestamp with each user's diary would be reasonable. Timestamps must be unique and totally ordered. In practice a starting time could be used with a suitable tie-break. The server associates a timestamp with a client's transaction when it invokes `startTransaction`.

When a client requests an operation on a shared object then its transaction's timestamp is compared with the one on the object. If the transaction's timestamp is earlier then the transaction is "too late" and the transaction is doomed to failure (either signalled immediately, or exposed at the subsequent `commitTransaction`). If the transactions's timestamp is later then the operation can proceed (updating the object's timestamp to match) as soon as the previous transaction on that object has committed or aborted.

The system should be made robust against clients failing mid-transaction. For instance, transactions could be aborted if no contact has been had with the client for a specified amount of time. Similarly, sufficient information must be held to roll-back the effects of aborted transactions.

This enforces strict isolation because transactions are never permitted to see the effects of non-committed transactions. Note that the ordering between the timestamps of committed transactions gives a serialization order.