RPC: Remote Procedure Call

March 2, 2020

Roadmap

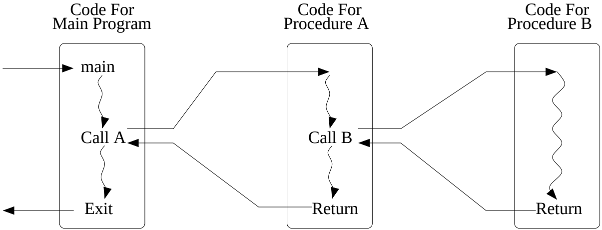
Roadmap

Remote Procedure Call (RPC)

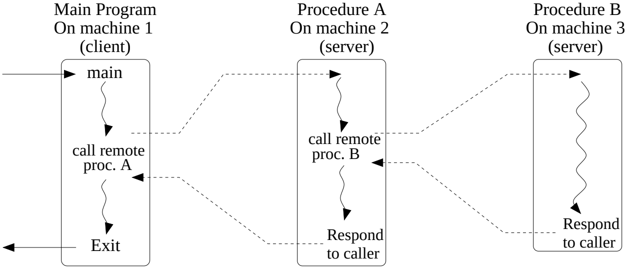
- Message-based programming with `send()/receive()` primitives is not convenient
  - depends on the communication protocol used (TCP vs. UDP)
  - requires the specification of an application protocol
  - akin to I/O
- Function/procedure call in a remote computer
  - is a familiar paradigm
  - eases transparency
  - is particularly suited for client-server applications

RPC: the Idea

Local procedure call:

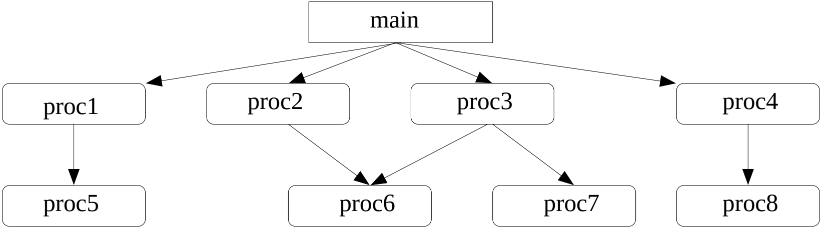


Remote procedure call:

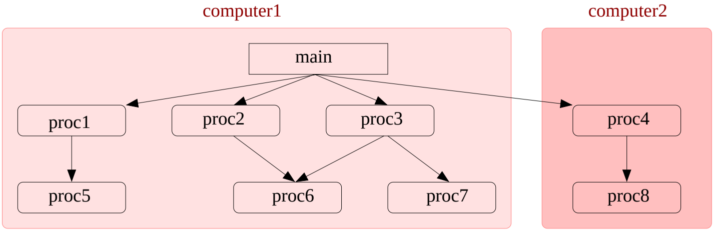


Program Development with RPCs: the Vision

- Design/develop an application ignoring distribution



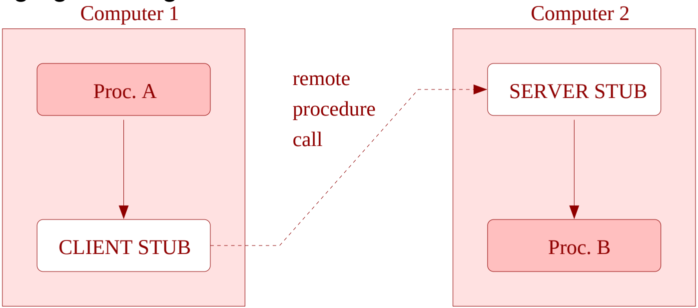
- Distribute *a posteriori*



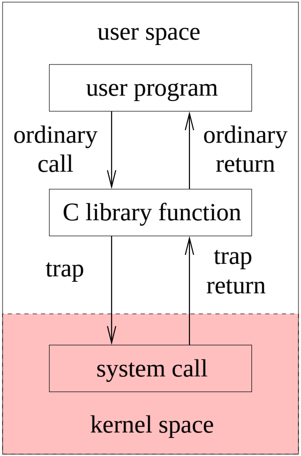
Roadmap

RPC Stub Routines

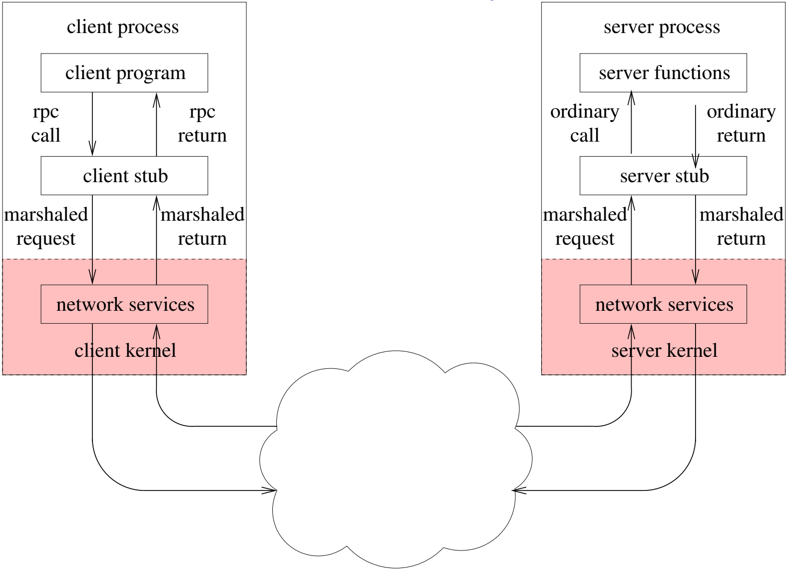
- Ensure RPC transparency
  Client invokes the **client stub** – a local function
  Remote function is invoked by the **server stub** – a local function
- The stub routines communicate with one another by exchanging messages



Well Known Trick: also Used for System Calls



Typical Architecture of an RPC System



Obs. RPC is typically implemented on top of the transport layer (TCP/IP)

Client Stub

Request

1. Assembles message: **parameter marshalling**
2. Sends message, via `write()`/`sendto()` to server
3. Blocks waiting for response, via `read()`/`recvfrom()`
   - Not in the case of **asynchronous RPC**

Response

1. Receives responses
2. Extracts the results (**unmarshalling**
3. Returns to client
   - Assuming **synchronous RPC**

Server Stub

Request

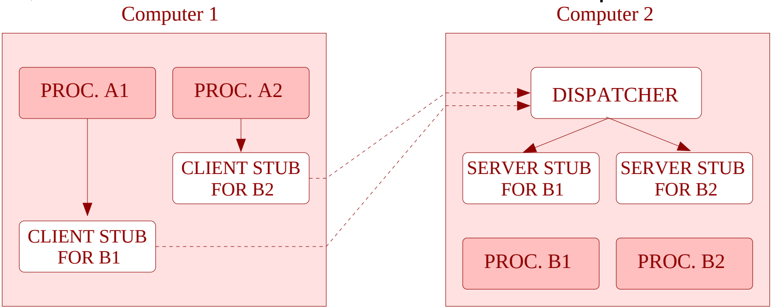
1. Receives message with request, via `read()`/`recvfrom()`
2. Parses message to determine arguments (**unmarshalling**
3. Calls function

Response

1. Assembles message with the return value of the function
2. Sends message, via `write()`/`sendto()`
3. Blocks waiting for a new request

RPC: Dispatching

- Often, **RPC services** offer more than one remote procedure:



- The identification of the procedure is performed by the **dispatcher**
  - This leads to a hierarchical name space **(service, procedure)**

Roadmap

Transparency: Platform Heterogeneity

Problems at least two:

1. Different architectures use different formats
   - 1's-complement vs. 2's complement
   - big-endian vs. little-endian
   - ASCII vs. UTF-??
2. Languages or compilers may use different representations for composite data-structures

Solution mainly two:

standardize format in the wires

+ needs only two conversions in each platform
– may not be efficient

receiver-makes-right

Transparency: Addresses as Arguments

Issue The meaning of an address (C pointer) is specific to a process

Solution Use **call-by-copy/restore** for parameter passing

+ Works in most cases
- Complex
  - The same address may be passed in different arguments
- Inefficient
  - For complex data structures, e.g. trees

Transparency in the Presence of Faults

Problem What if something breaks?

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes
  - Need to prevent **orphan** computations, i.e. on behalf of a dead process.

Transparency in the Presence of Faults

Problem What if something breaks?

- The client cannot locate the server
  - RPC can return an error (like in the case of a system call)
- The request-message is lost
  - Retransmit it, after a timeout
- The response-message is lost
  - Must use request identifiers (sequence nos.)
  - Must save most recent responses for replay, if the request is not **idempotent**
- Server crashes
  - Was the request processed before the crash?
- Client crashes
  - Need to prevent **orphan** computations, i.e. on behalf of a dead process.

Issue A client cannot distinguish between loss of a request, loss of a response or a server crash

- The absence of a response may be caused by a slow network/server

Roadmap

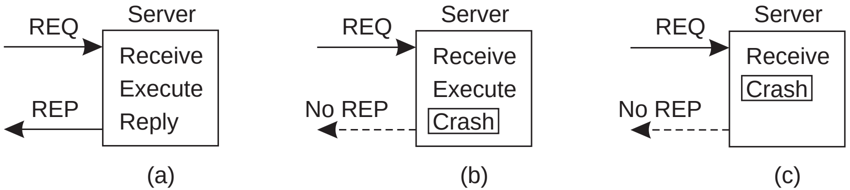
RPC Semantics in the Presence of Faults (Spector82)

Question What can a client expect when there is a fault?

Answer Depends on the semantics in the presence of faults provided by the RPC system

- **At-least-once** Client stub must keep retransmitting until it obtains a response
  - Be careful with non-idempotent operations
  - Spector allows for zero executions in case of server failure
- **At-most-once** Not trivial if you use a non-reliable transport, e.g. UDP.
  - If the RPC uses TCP, it may report an error when the TCP connection breaks
- **Exactly-once** Not always possible to ensure this semantics, especially if there are external actions that cannot be undone

Faults and Exactly-once Semantics



Problem In the case of external actions, e.g. file printing, it is virtually impossible to ensure Exactly-once Semantics

Server policy One of two:

1. Send an `ACK` after printing
2. Send an `ACK` before printing

Client policy One of four:

1. Never resend the request
2. Always resend the request
3. Resend the request when it receives an `ACK`
4. Resend the request when it does not receive an `ACK`

Server Faults and Exactly-once Semantics

Scenario Server crashes and quickly recovers so that it is able to handle client retransmission, but **it has lost all state**

Let

`A`: ACK   `P`: print   `C`: crash

Fault scenarios (`ACK->P`)

1. `A->P->C`
2. `A->C(->P)`
3. `C(->A->P)`

Fault scenarios (`P->ACK`)

1. `P->A->C`
2. `P->C(->A)`
3. `C(->P->A)`

Client	Server					
	Strategy A→P			Strategy P→A		
Reissue Strategy	**APC**	**AC(P)**	**C(AP)**	**PAC**	**PC(A)**	**C(PA)**
Always	Dup	OK	OK	Dup	Dup	OK
Never	OK	Zero	Zero	OK	OK	Zero
When Ack	Dup	OK	Zero	Dup	OK	Zero
When not Ack	OK	Zero	OK	OK	Dup	OK

OK = Text printed once   Dup = Text printed twice   Zero = Text not printed at all

Conclusion No combined strategy works on every fault scenario

- What if server saved state on disk?

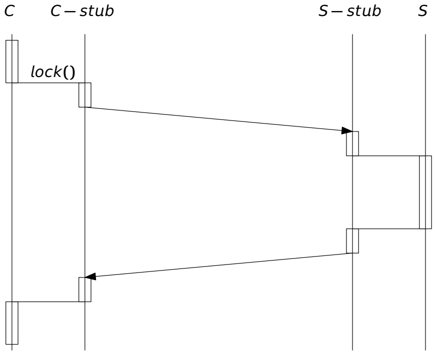
At-least-once vs. At-most-once

- Consider a locking service using two RPCs:

  `lock()`

  `unlock()`

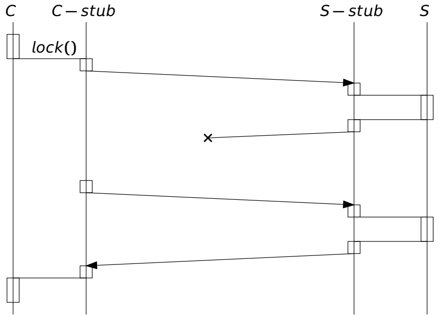
No failures and no message loss



- It does not matter the semantics supported by the RPC library

At-least-once vs. At-most-once: Lost Response

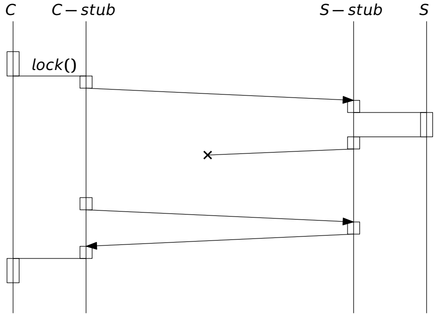
At-least-once



- Remote procedure may be invoked more than once
  - If procedure is not **idempotent**:
    - RPC must include an id as argument
    - Server must keep table with responses previously sent
  - Is `lock()` an idempotent procedure?

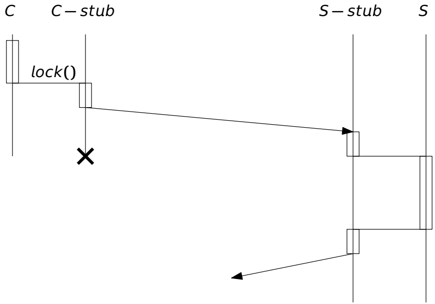
At-least-once vs. At-most-once: Lost Response

At-most-once (UDP?):



- There is no guarantee that the procedure will be executed
  - But in that case, the caller should receive an exception
- The RPC middleware ensures that the procedure is not executed more than once
  - RPC requests include an id
  - RPC system keeps table with responses
- What would be different if using TCP?

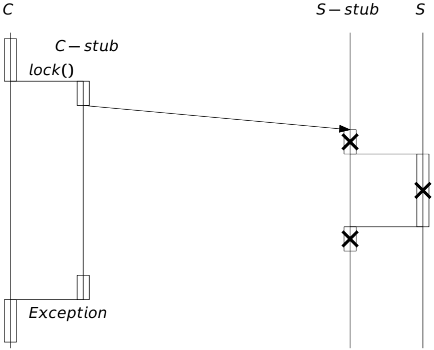
At-least-once vs. At-most-once: Client crash



- Again, the RPC semantics is irrelevant

At-least-once vs. At-most-once: Server crash

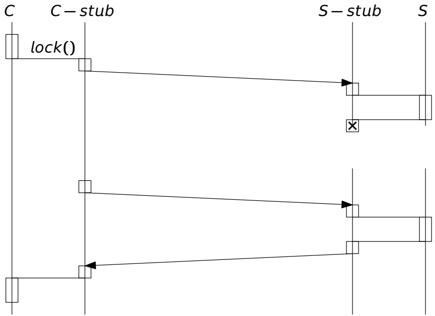
At-most-once



- Client does not know if server granted it the lock
  - Depends on when the server crashed
- Client, **not RPC**, may ask the server (or just retry)
  - Server needs to remember state across reboots
    - E.g. store locks state on disk
- Is this different from an exception upon message loss?

At-least-once vs. At-most-once: Server crash

At-least-once



- Server may run the procedure several times
  - Client stub may send several requests before giving up
- Server needs to remember previous requests across reboots (if requests are not **idempotent**). E.g.:
  - Store table request ids on disk
  - Check the request table on each request

At-least-once vs. At-most-once: Conclusions

Message loss

At-least-once

- Suits if requests are idempotent

At-most-once

- Appropriate when requests are not idempotent

Server crashes

- No clear advantage: the service itself may have to take special measures

Upon an exception can the caller tell whether the cause is message loss or server crash?

Roadmap

Idea

Implementation

Transparency

RPC Semantics in the Presence of Faults

Further Reading

Further Reading

- Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Section 4.2 *Remote Procedure Call*, except subsection 4.2.4
  - Subsection 8.3.2 *RPC Semantics in the Presence of Failures*
- Birrel and Nelson, *"Implementing Remote Procedure Calls"*, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, Pages 39-59
- A. Spector, *"Performing Remote Operations Efficiently on a Local Computer Network"*, Communications of the ACM, Vol. 25, No. 4, April 1982, Pages 246-260