Communication - 1

Distributed Systems [4-1]

殷亚凤

Email: yafeng@nju.edu.cn

Homepage: http://cs.nju.edu.cn/yafeng/

Room 301, Building of Computer Science and Technology

Review

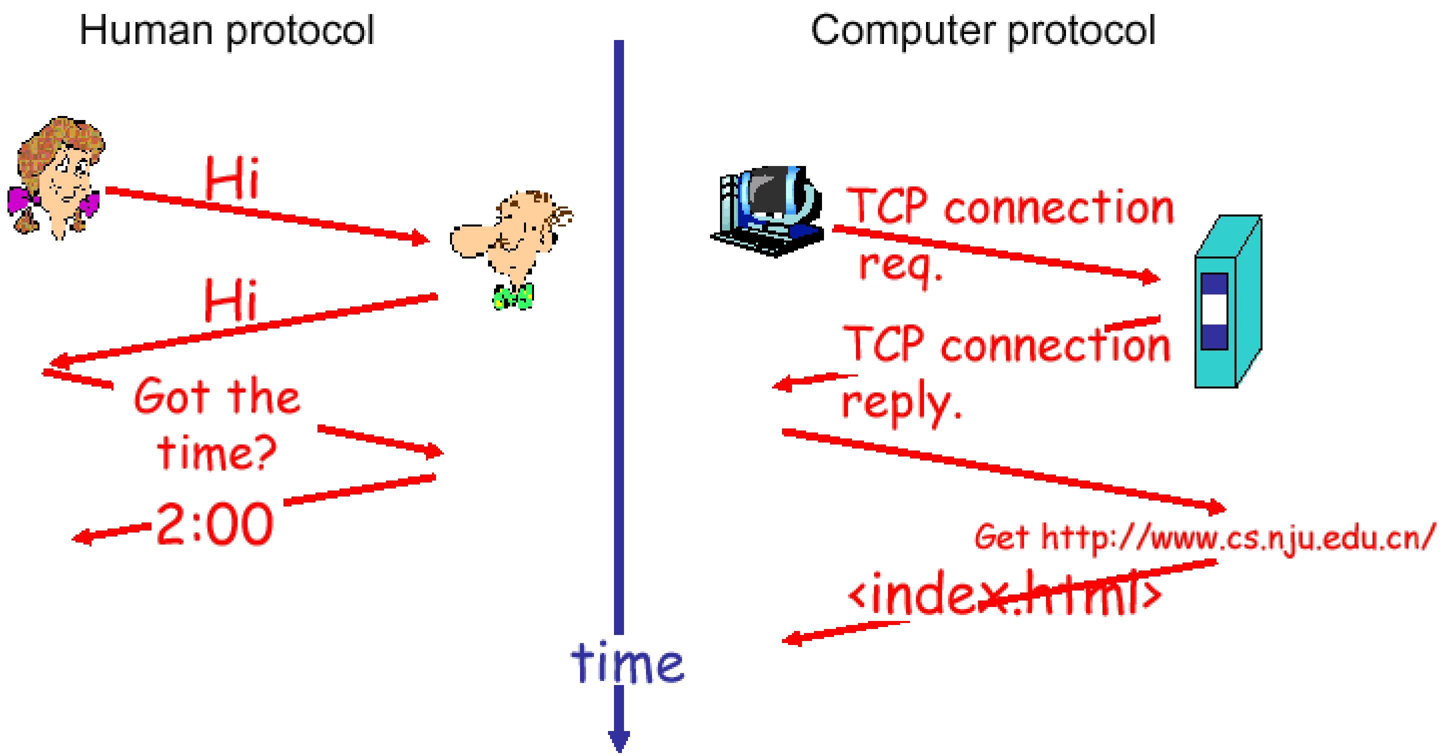
- **Process**: Execution stream, Process state
- **Thread**: A minimal software processor in whose context a series of instructions can be executed
- **Client – Server**: Multithreaded Clients, Multithreaded Servers
- **Code Migration**: Code segment, resource segment, execution segment, weak and strong mobility

This lesson

- **Protocols**: Low-level layers, Transport layer, Application layer, Middleware layer

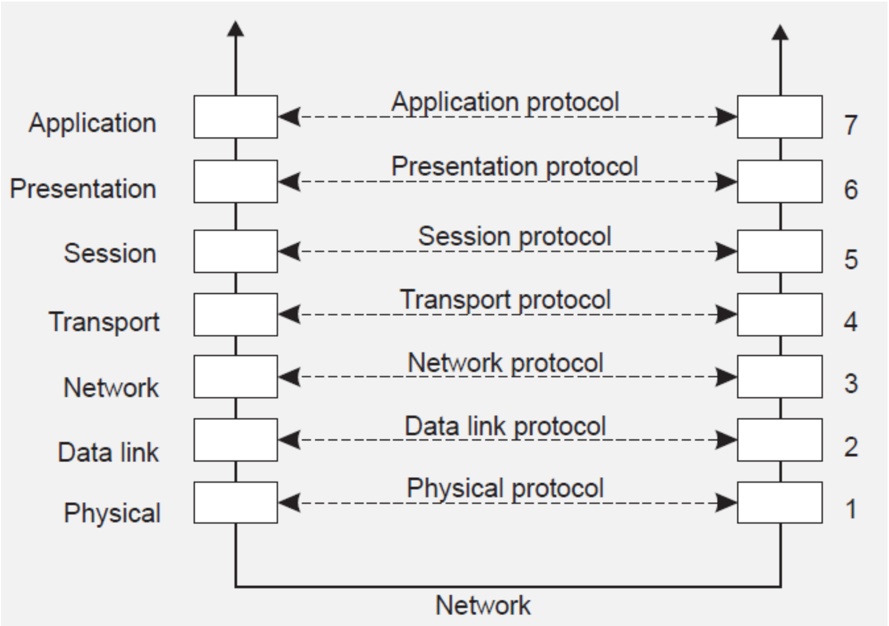
- **Remote Procedure Call**
  - Basic RPC operation
  - Parameter passing
  - Asynchronous RPC
  - RPC in Presence of Failures

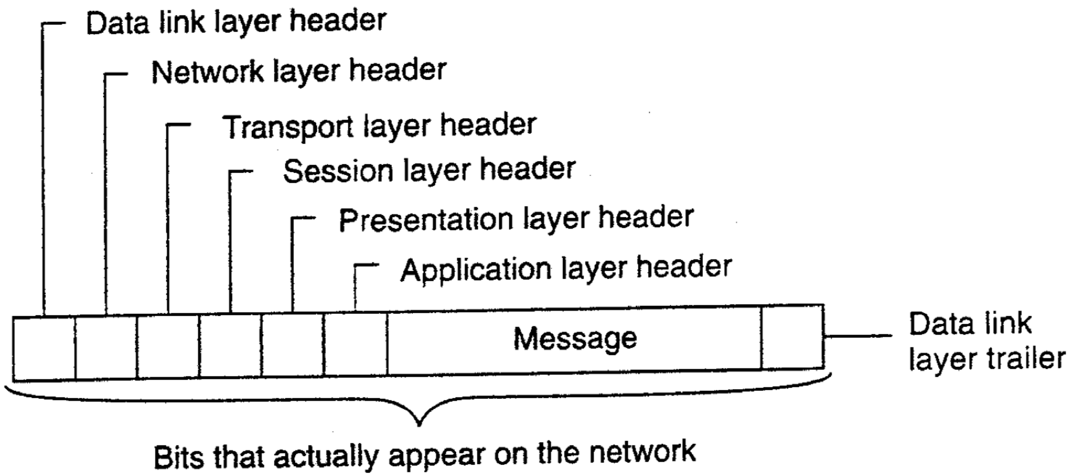
What is a protocol?



Client-Server TCP

Basic networking model





Layered Protocols

- Low-level layers
- Transport layer
- Middleware layer
- Application layer

Low-level layers

- Recap:
  - Physical layer: contains the specification and implementation of bits, and their transmission between sender and receiver
  - Data link layer: prescribes the transmission of a series of bits into a frame to allow for error and flow control
  - Network layer: describes how packets in a network of computers are to be routed.

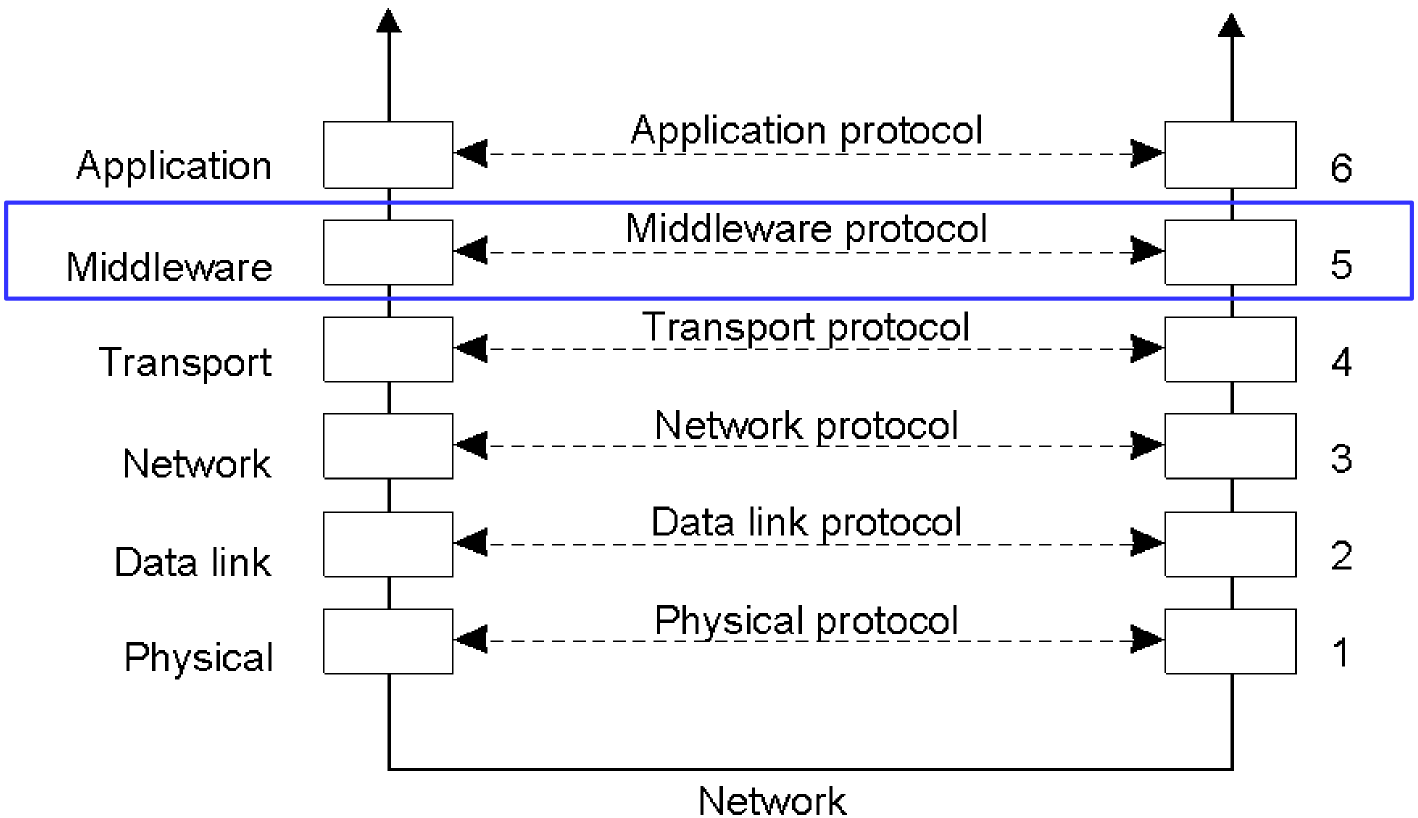
Transport Layer

- The transport layer provides the actual communication facilities for most distributed systems.
- Standard Internet protocols:
  - TCP: connection-oriented, reliable, stream-oriented communication
  - UDP: unreliable (best-effort) datagram communication
- IP multicasting is often considered a standard available service (which may be dangerous to assume).

Middleware Layer

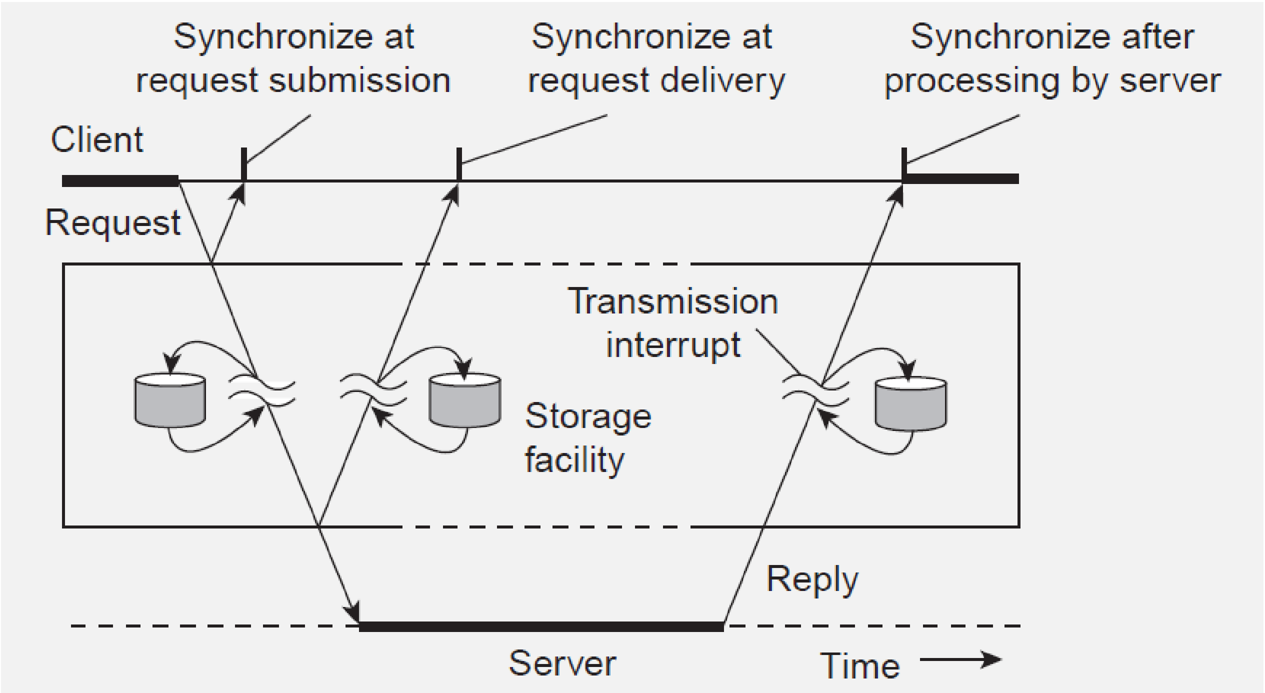
- Middleware is invented to provide common services and protocols that can be used by many different applications
  - A rich set of communication protocols
  - (Un)marshaling of data, necessary for integrated systems
  - Naming protocols, to allow easy sharing of resources
  - Security protocols for secure communication
  - Scaling mechanisms, such as for replication and caching
- What remains are truly application-specific protocols…

Middleware Protocols



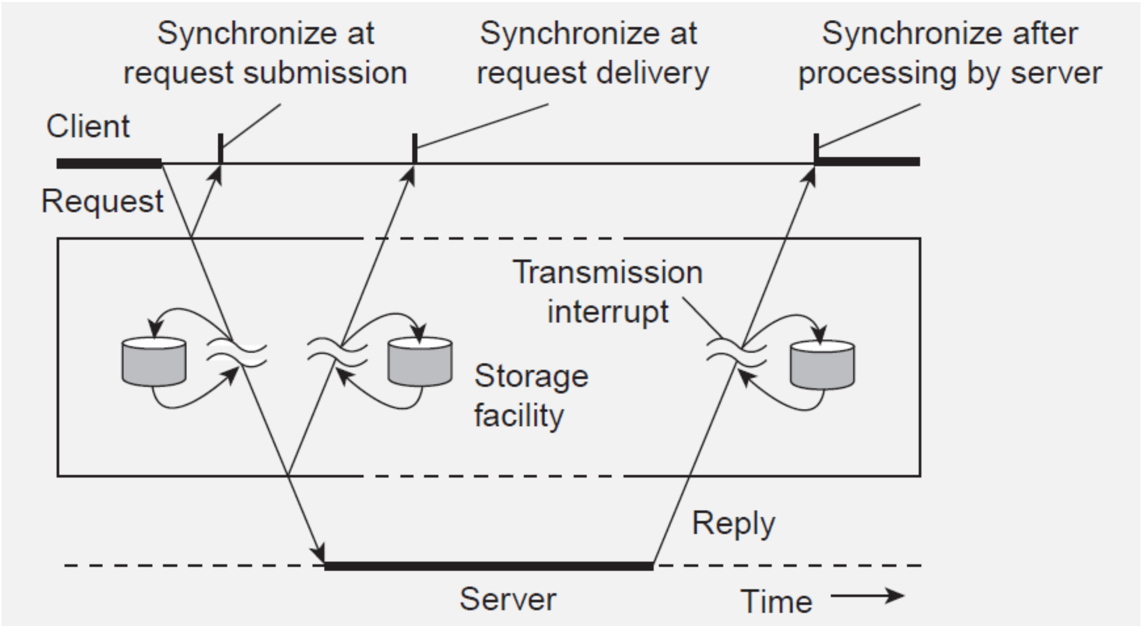
- An adapted reference model for networked communication.

Types of communication



- Persistent communication: A message is stored at a communication server as long as it takes to deliver it.
- Transient communication: Comm. server discards message when it cannot be delivered at the next server, or at the receiver.

Types of communication



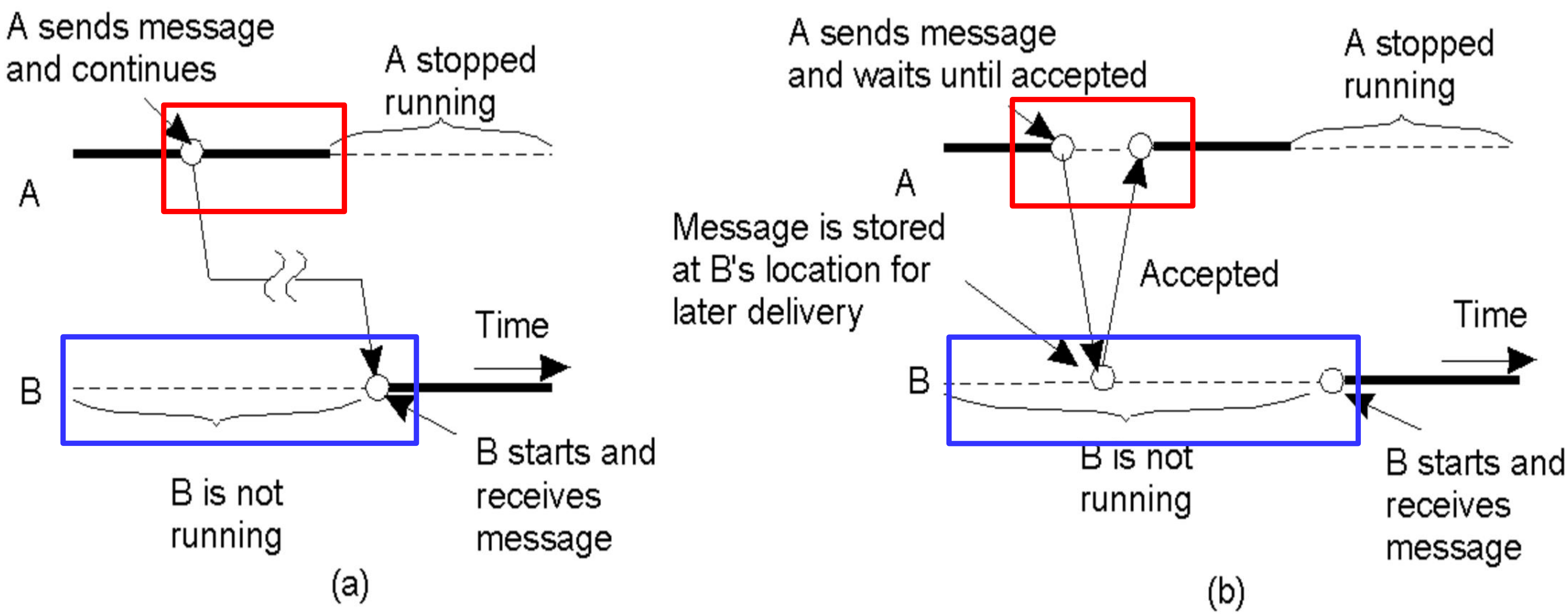
- Asynchronous versus synchronous communication
- Synchronous communication
  - At request submission
  - At request delivery
  - After request processing

Combination

- Persistent + Asynchronous communication
- Persistent + Synchronous communication

- Transient + Asynchronous communication
- Transient + Synchronous communication

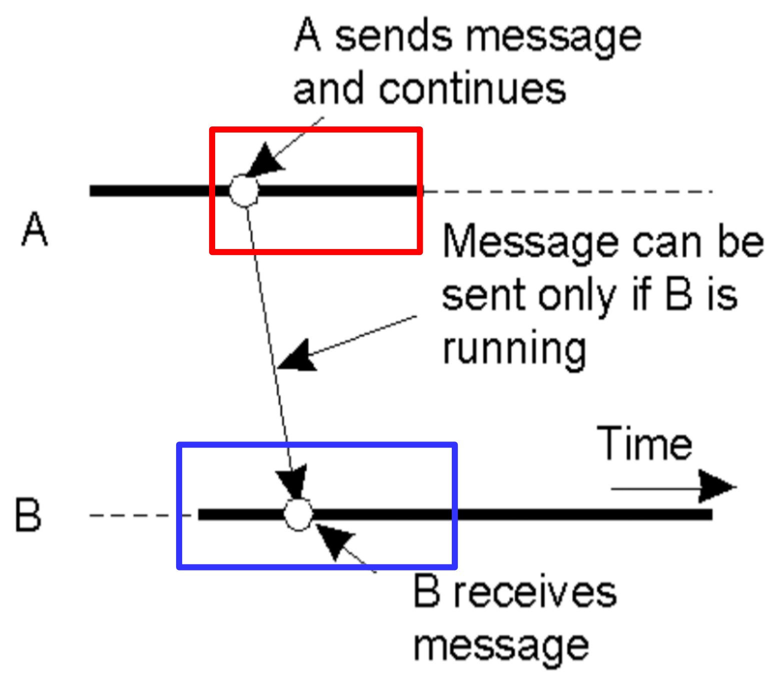
Persistent communication



Persistent + Asynchronous communication

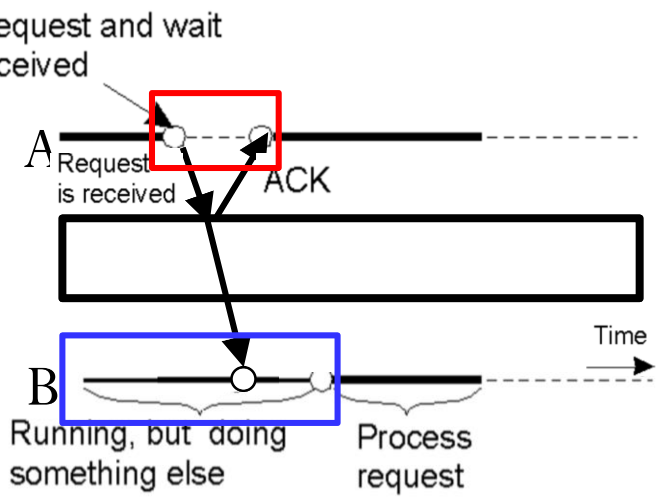
Persistent + Synchronous communication

Transient + Asynchronous communication

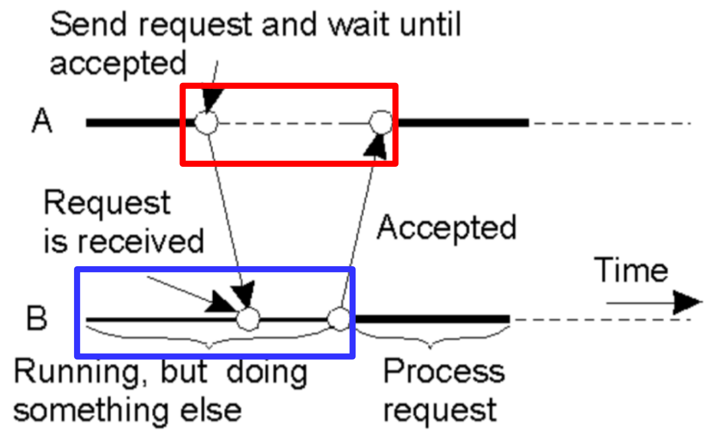


Transient + Asynchronous communication

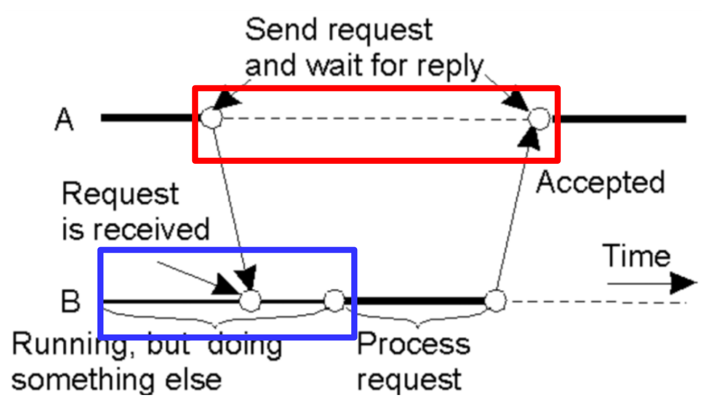
Transient + Synchronous communication



At request submission



At request delivery



After request processing

Client/Server

- Client/Server computing is generally based on a model of transient synchronous communication:
  - Client and server have to be active at time of communication
  - Client issues request and blocks until it receives reply
  - Server essentially waits only for incoming requests, and subsequently processes them
- Drawbacks of synchronous communication
  - Client cannot do any other work while waiting for reply
  - Failures have to be handled immediately: the client is waiting
  - The model may simply not be appropriate (mail, news)

Messaging

- Aims at high-level persistent asynchronous communication:
  - Processes send each other messages, which are queued
  - Sender need not wait for immediate reply, but can do other things
  - Middleware often ensures fault tolerance

Remote Procedure Call (RPC)

- Basic RPC operation
- Parameter passing
- Variations

Basic RPC operation

- Observations
  - Application developers are familiar with simple procedure model
  - Well-engineered procedures operate in isolation (black box)
  - There is no fundamental reason not to execute procedures on separate machine

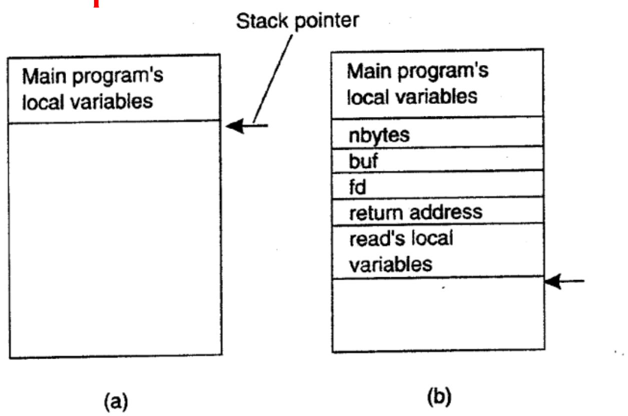


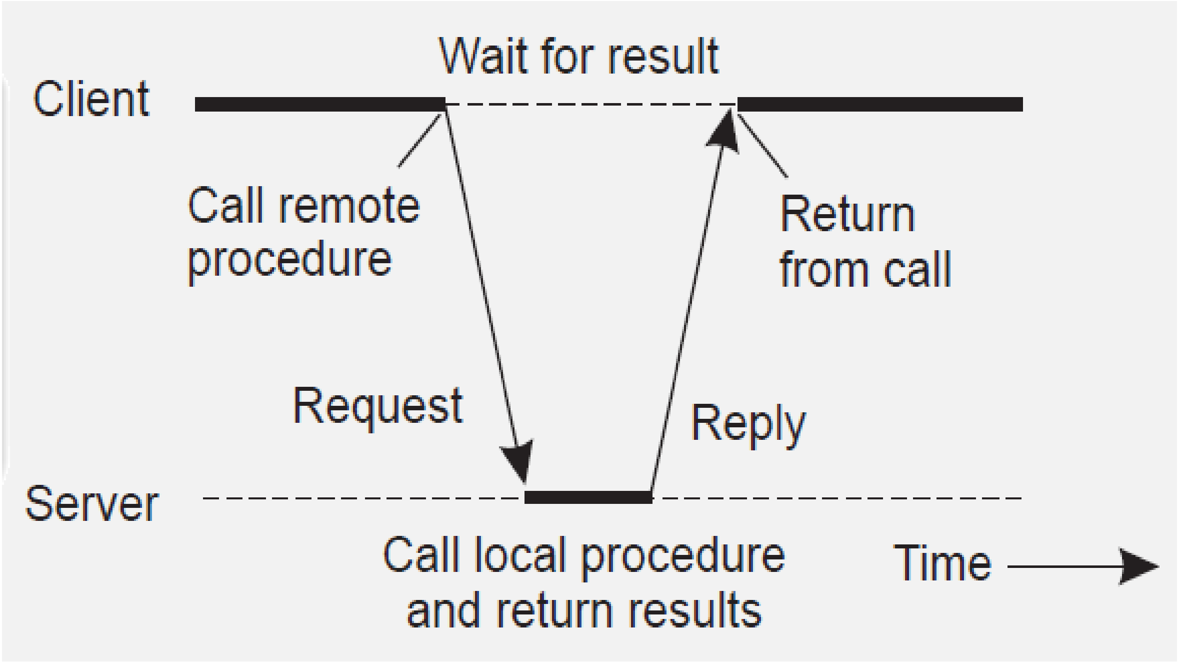
Figure 4-5. (a) Parameter passing in a local procedure call: the stack before the call to read. (b) The stack while the called procedure is active.

Conventional Procedure Call

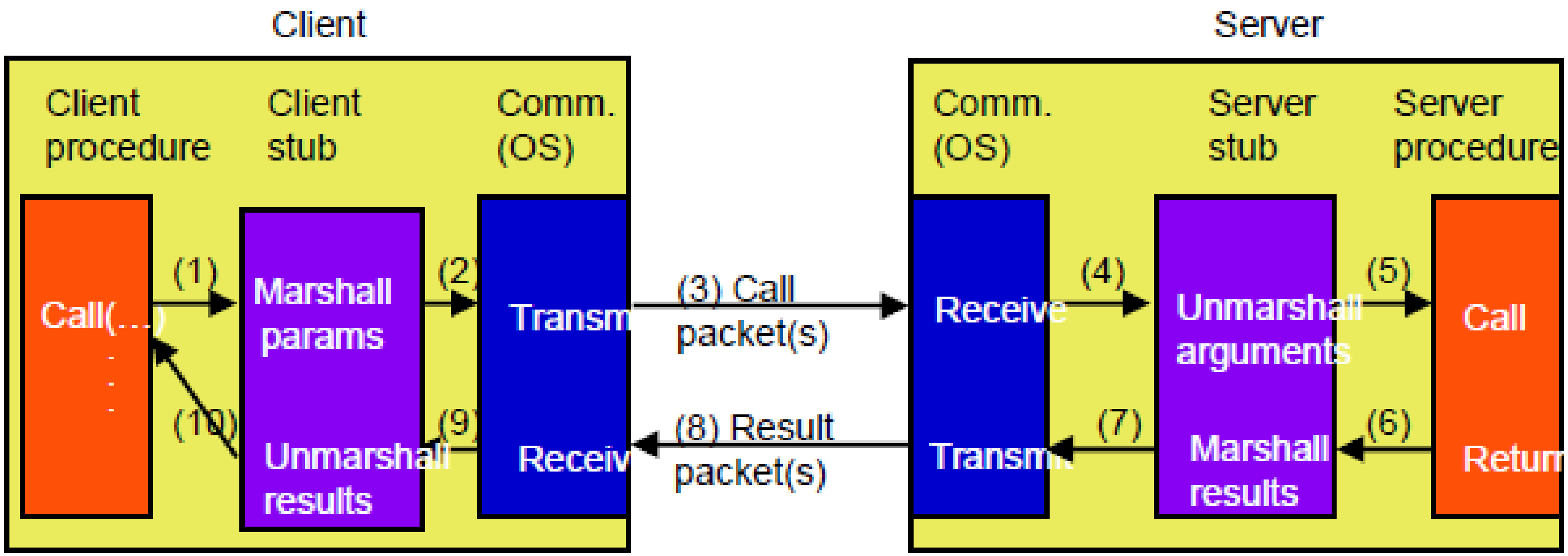
- Communication between caller & callee can be hidden by using procedure-call mechanism.

Basic RPC operation

- Communication between caller & callee can be hidden by using procedure-call mechanism.



Basic RPC operation



1. Client procedure calls client stub.
2. Stub builds message; calls local OS.
3. OS sends message to remote OS.
4. Remote OS gives message to server stub.
5. Server stub unpacks parameters and calls server.
6. Server makes local call and returns result to server stub.
7. Stub builds message; calls server OS.
8. OS sends message to client's OS.
9. Client's OS gives message to client stub
10. Client stub unpacks result and returns to the client.

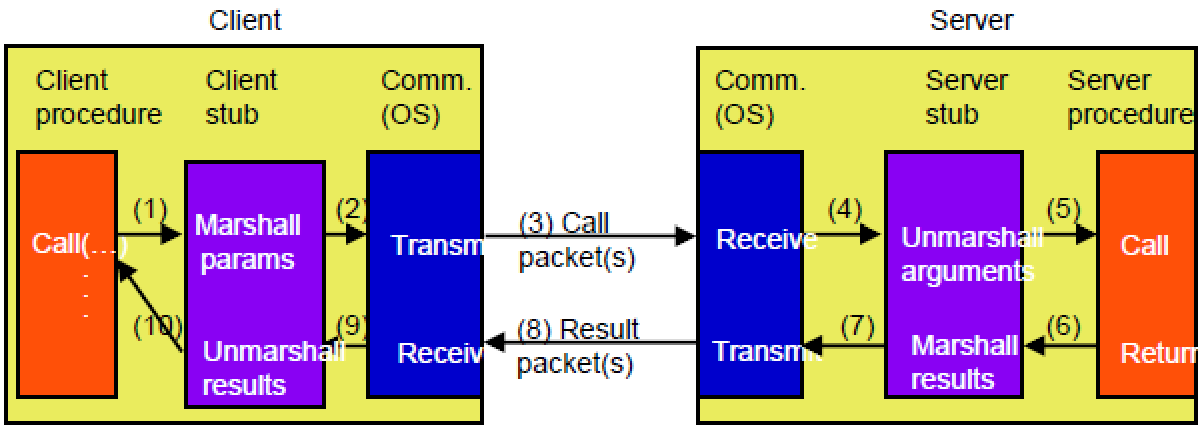
RPC: Parameter passing

- There's more than just wrapping parameters into a message:
  - Client and server machines may have different data representations (think of byte ordering)
  - Wrapping a parameter means transforming a value into a sequence of bytes
  - Client and server have to agree on the same encoding:
    - How are basic data values represented (integers, floats, characters)
    - How are complex data values represented (arrays, unions)
  - Client and server need to properly interpret messages, transforming them into machine-independent representations.

RPC: Parameter passing

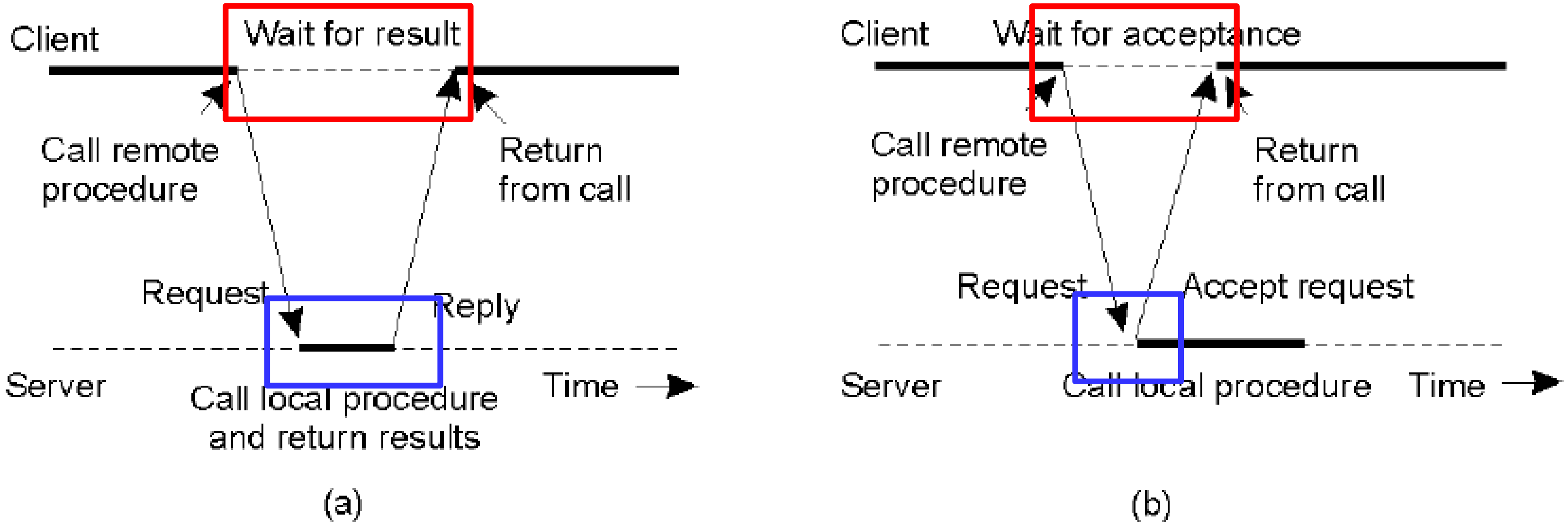
- Some assumptions:
  - Copy in/copy out semantics: while procedure is executed, nothing can be assumed about parameter values.
  - All data that is to be operated on is passed by parameters. Excludes passing references to (global) data.
- Full access transparency cannot be realized.
- A remote reference mechanism enhances access transparency:
  - Remote reference offers unified access to remote data
  - Remote references can be passed as parameter in RPCs

RPC Recap



- Logical extension of the procedure call interface
  - Easy for the programmer to understand and use
- Requires an Interface Definition Language (IDL) to specify data types and procedure interfaces
  - Provides language independence

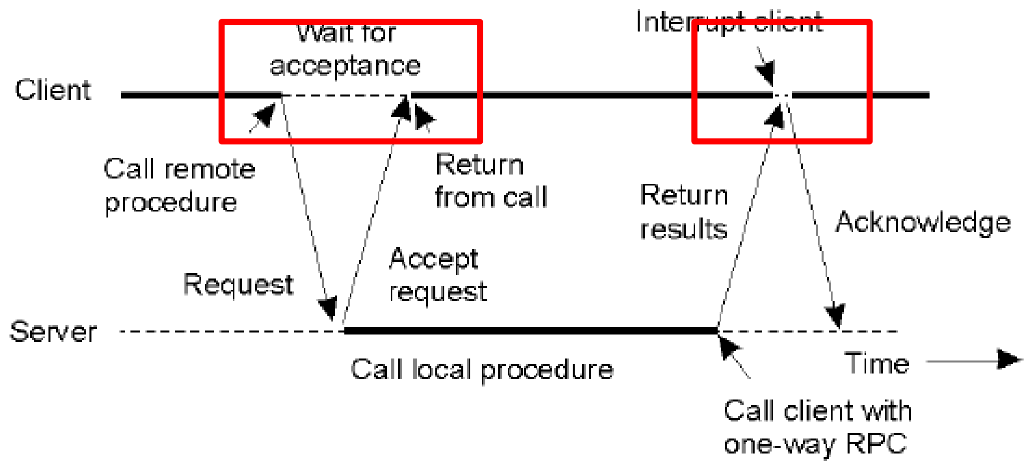
Asynchronous RPC (1)



Traditional (synchronous) RPC interaction

Asynchronous RPC interaction

Asynchronous RPC (2)

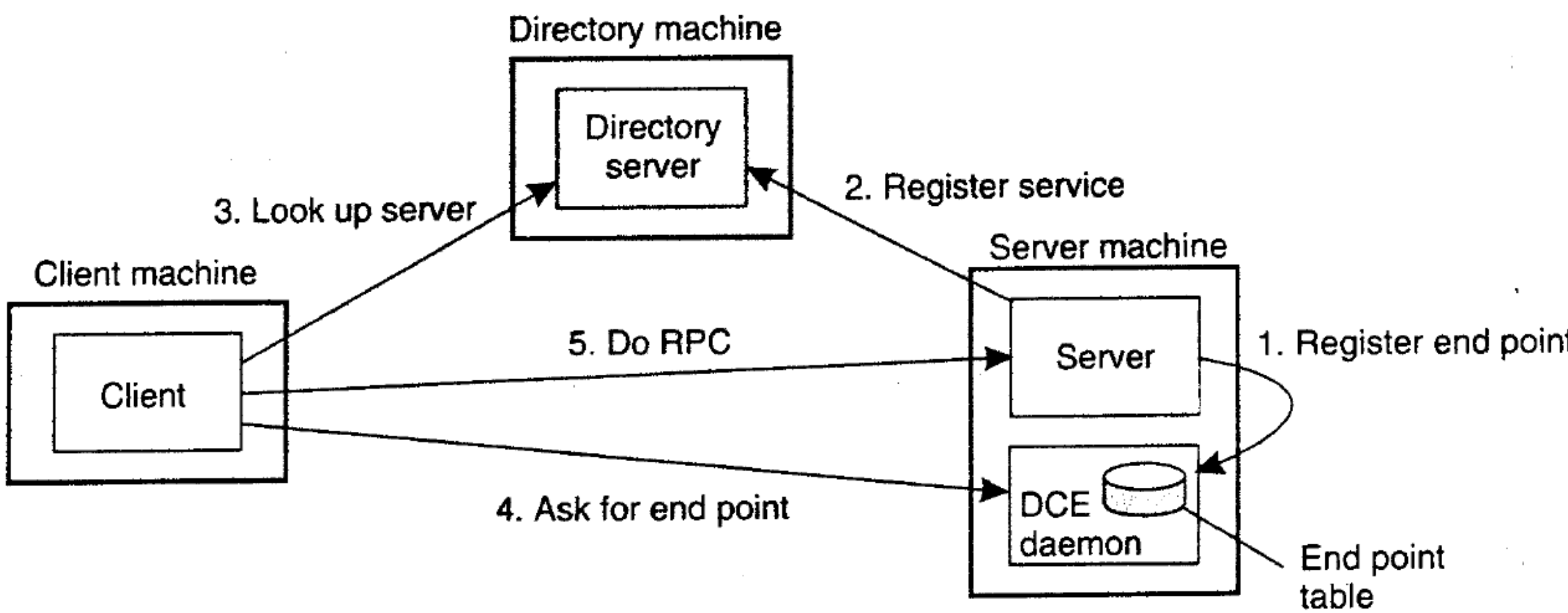


- A client and server interacting through two asynchronous RPCs
  - Known as deferred synchronous RPC
- Some RPCs do not require that the client waits for the acceptance
  - Known as one-way RPC

How does the client locate the server?

- Hardwire the server address into the client
  - Fast but inflexible!
- A better method is dynamic binding:
  - When the server begins executing, the call to initialize outside the main loop exports the server interface
  - This means that the server sends a message to a program called a binder, to make its existence known. This process is referred to as registering

How does the client locate the server?



Advantages of Dynamic Binding

- Flexibility
- Can support multiple servers that support the same interface, e.g.:
  - Binder can spread the clients randomly over the servers to even load
  - Binder can poll the servers periodically, automatically deregistering the servers that fail to respond, to achieve a degree of fault tolerance
  - Binder can assist in authentication: For example, a server specifies a list of users that can use it; the binder will refuse to tell users not on the list about the server
- The binder can verify that both client and server are using the same version of the interface

Disadvantages of Dynamic Binding

- The extra overhead of exporting/importing interfaces costs time
- The binder may become a bottleneck in a large distributed system

RPC in Presence of Failures

- Five different classes of failures 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

- Examples:
  - Server might be down
  - Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub
- Possible solutions:
  - Use a special code, such as "-1", as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable errno.
    - "-1" can be a legal value to be returned, e.g., sum(7, -8)
  - Have the error raise an exception (like in ADA) or a signal (like in C).
    - Not every language has exceptions/signals (e.g., Pascal). Writing an exception/signal handler destroys the transparency

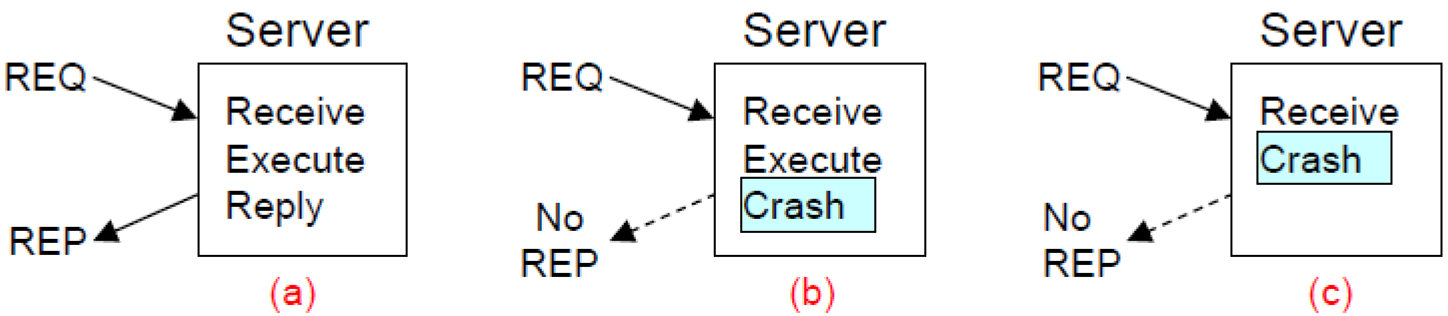
Lost Request Message

- Kernel starts a **timer** when sending the message:
  - If timer expires before reply or ACK comes back: Kernel retransmits
  - If message truly lost: Server will not differentiate between original and retransmission ⇒ everything will work fine
  - If many requests are lost: Kernel gives up and falsely concludes that the server is down ⇒ we are back to "Cannot locate server"

Lost Reply Message

- Kernel starts a **timer** when sending the message:
  - If timer expires before reply comes back: Retransmits the request
    - Problem: **Not sure why no reply** (reply/request lost or server slow) ?
  - If server is just slow: The procedure will be executed several times
    - Problem: What if the request is not idempotent, e.g. money transfer
  - Way out: Client's kernel assigns sequence numbers to requests to allow server's kernel to differentiate retransmissions from original

Server crashes



- Problem: Clients' kernel cannot differentiate between (b) and (c)
- Note: Crash can occur before Receive, but this is the same as (c).

Server crashes

- 3 schools of thought exist on what to do here:
  - Wait until the server reboots and try the operation again. Guarantees that RPC has been executed at least one time (at least once semantic)
  - Give up immediately and report back failure. Guarantees that RPC has been carried out at most one time (at most once semantics)
  - Client gets no help. Guarantees nothing (RPC may have been carried out anywhere from 0 to a large number). Easy to implement.

Client Crashes

- Client sends a request and crashes before the server replies: A computation is active and no parent is waiting for result (orphan)
  - Orphans waste CPU cycles and can lock files or tie up valuable resources
  - Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)

Client Crashes

- Possible solutions
  - Extermination: Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - **Expense of writing** a disk record for every RPC; orphans may do RPCs, thus creating **grandorphans impossible to locate**; **impossibility to kill orphans** if the network is partitioned due to failure.

Client Crashes (2)

- Possible solutions (cont'd)
  - Reincarnation: Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - Gentle reincarnation: A variant of previous one, but less Draconian
    - When an epoch broadcast comes in, each machine that has remote computations tries to locate their owner. A computation is killed only if the owner cannot be found

Client Crashes (3)

- Possible solutions (cont'd)
  - Expiration: Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - Choosing a reasonable value of T in the face of RPCs with wildly differing requirements is difficult

RPC Recap

- New failure models
  - Cannot locate server (throw exception)
  - Lost request message (resend request)
  - Lost reply message
    - Ack reply and resend reply if ack not received

      --- What if network failure is sufficiently long that your TCP connection times out?
    - Retransmit request without incrementing the request sequence number
    - Requires that the server retain old replies for some set time period
  - Server crashes before sending reply
    - RPC resends request (at least once semantics)
    - Give up and report failure (at most once semantics)

      --- Or server saves all replies on persistent storage and re-send previous replies
    - Client gets no help, semantics determined by the client
  - Client crashes