

# Last Class: Communication in Distributed Systems

- Structured or unstructured?
- Addressing?
- Blocking/non-blocking?
- Buffered or unbuffered?
- Reliable or unreliable?
- Server architecture
- Scalability
- Push or pull?
- Group communication



# Today: Remote Procedure Calls

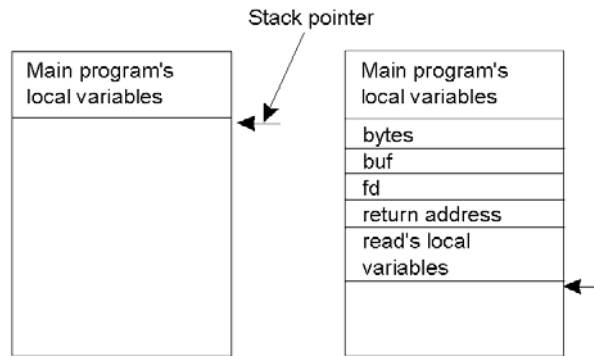
- Goal: Make distributed computing look like centralized computing
- Allow remote services to be called as procedures
  - Transparency with regard to location, implementation, language
- Issues
  - How to pass parameters
  - Bindings
  - Semantics in face of errors
- Two classes: integrated into prog, language and separate



# Conventional Procedure Call

a) Parameter passing in a local procedure call: the stack before the call to read

b) The stack while the called procedure is active

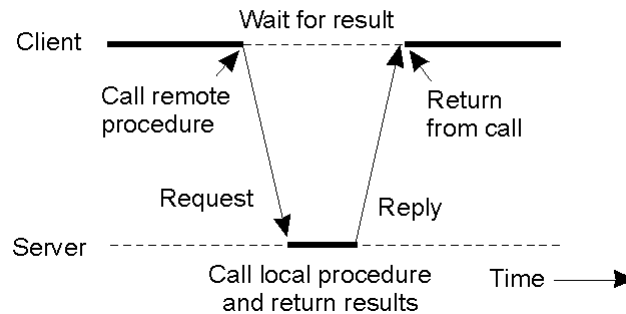


# Parameter Passing

- Local procedure parameter passing
  - Call-by-value
  - Call-by-reference: arrays, complex data structures
- Remote procedure calls simulate this through:
  - Stubs – proxies
  - Flattening – marshalling
- Related issue: global variables are not allowed in RPCs

# Client and Server Stubs

- Principle of RPC between a client and server program.



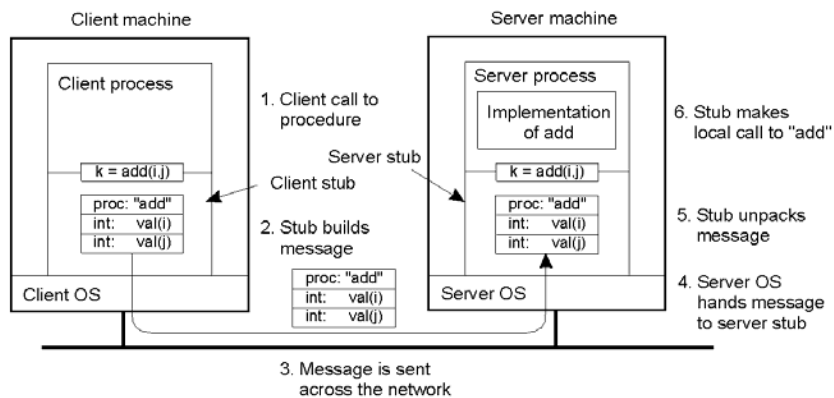
# Stubs

- Client makes procedure call (just like a local procedure call) to the client stub
- Server is written as a standard procedure
- Stubs take care of packaging arguments and sending messages
- Packaging is called *marshalling*
- Stub compiler generates stub automatically from specs in an Interface Definition Language (IDL)
  - Simplifies programmer task

# Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

# Example of an RPC



# Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a standard representation
  - Example: external data representation (XDR)
- Problem: how do we pass pointers?
  - If it points to a well-defined data structure, pass a copy and the server stub passes a pointer to the local copy
- What about data structures containing pointers?
  - Prohibit
  - Chase pointers over network
- Marshalling: transform parameters/results into a byte stream

# Binding

- Problem: how does a client locate a server?
  - Use Bindings
- Server
  - Export server interface during initialization
  - Send name, version no, unique identifier, handle (address) to binder
- Client
  - First RPC: send message to binder to import server interface
  - Binder: check to see if server has exported interface
    - Return handle and unique identifier to client

## Binding: Comments

- Exporting and importing incurs overheads
- Binder can be a bottleneck
  - Use multiple binders
- Binder can do load balancing

## Failure Semantics

- *Client unable to locate server*: return error
- *Lost request messages*: simple timeout mechanisms
- *Lost replies*: timeout mechanisms
  - Make operation idempotent
  - Use sequence numbers, mark retransmissions
- *Server failures*: did failure occur before or after operation?
  - At least once semantics (SUNRPC)
  - At most once
  - No guarantee
  - Exactly once: desirable but difficult to achieve

# Failure Semantics

- *Client failure*: what happens to the server computation?
  - Referred to as an *orphan*
  - *Extermination*: log at client stub and explicitly kill orphans
    - Overhead of maintaining disk logs
  - *Reincarnation*: Divide time into epochs between failures and delete computations from old epochs
  - *Gentle reincarnation*: upon a new epoch broadcast, try to locate owner first (delete only if no owner)
  - *Expiration*: give each RPC a fixed quantum  $T$ ; explicitly request extensions
    - Periodic checks with client during long computations



# Implementation Issues

- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to NIC and from server NIC to server
  - As many as 7 copies
    - Stack in stub – message buffer in stub – kernel – NIC – medium – NIC – kernel – stub – server
  - Scatter-gather operations can reduce overheads

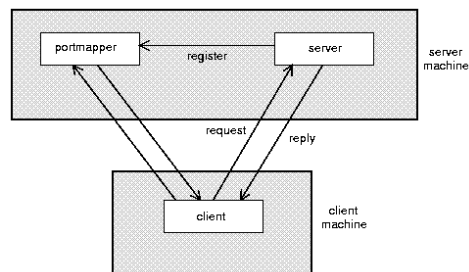


# Case Study: SUNRPC

- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8192 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple arguments marshaled into a single structure
- At-least-once semantics if reply received, at-least-zero semantics if no reply. With UDP tries at-most-once
- Use SUN's eXternal Data Representation (XDR)
  - Big endian order for 32 bit integers, handle arbitrarily large data structures

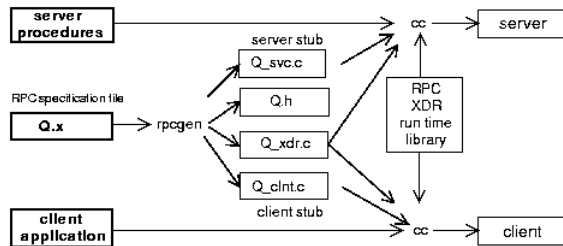
# Binder: Port Mapper

- Server start-up: create port
- Server stub calls *svc\_register* to register prog. #, version # with local port mapper
- Port mapper stores prog #, version #, and port
- Client start-up: call *clnt\_create* to locate server port
- Upon return, client can call procedures at the server





## *Rpcgen: generating stubs*



- Q\_xdr.c: do XDR conversion
- Detailed example: later in this course

## Summary

- RPCs make distributed computations look like local computations
- Issues:
  - Parameter passing
  - Binding
  - Failure handling
- Case Study: SUN RPC