Today's Class

- VM migration wrap-up
- Communication in distributed systems
- Remote Procedure Calls

Virtual Machine Migration Recap

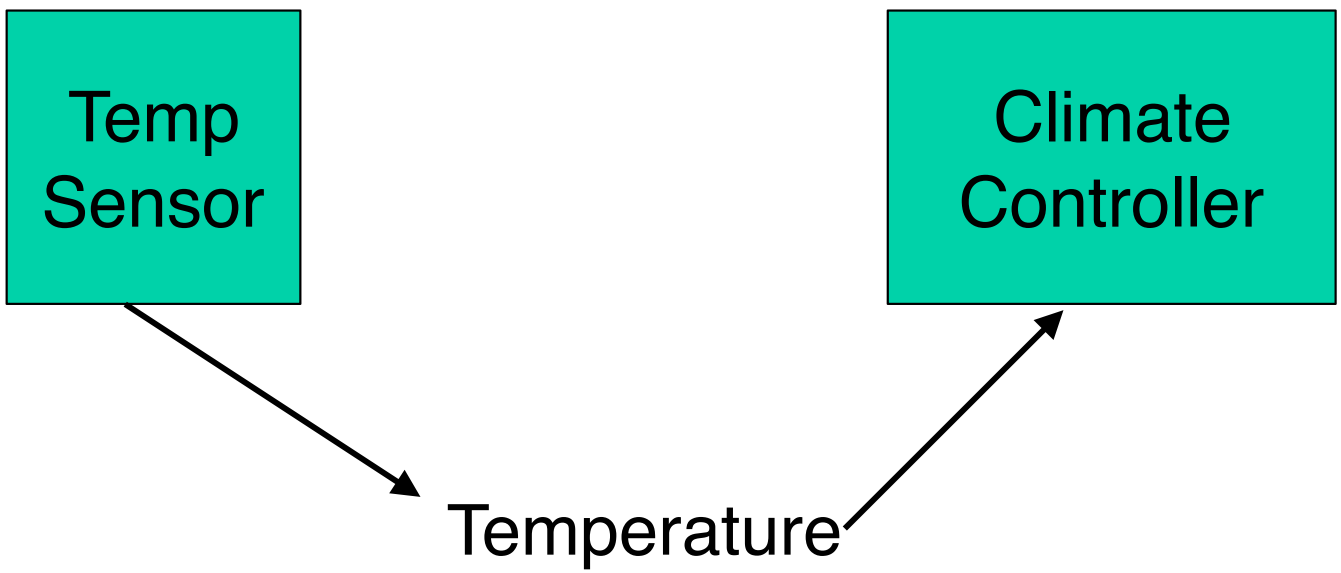
- Transfer VM state from one host to another
- VM state = CPU + **memory** + disk + network state
- Last time: memory state transfer using **pre-copy**
  - Memory state changes continuously
  - Changed memory state (dirty pages) are iteratively transferred
- Pre-copy : copy the VM state first, and then execute VM on destination host

Post-copy VM Migration

- Pre-copy : copy the VM state first, and then execute VM on destination host
- Post-copy: Begin VM execution on destination, and then copy VM state
- In post-copy, VM (almost) immediately begins running on destination
- Tradeoffs: immediacy of migration, performance,…

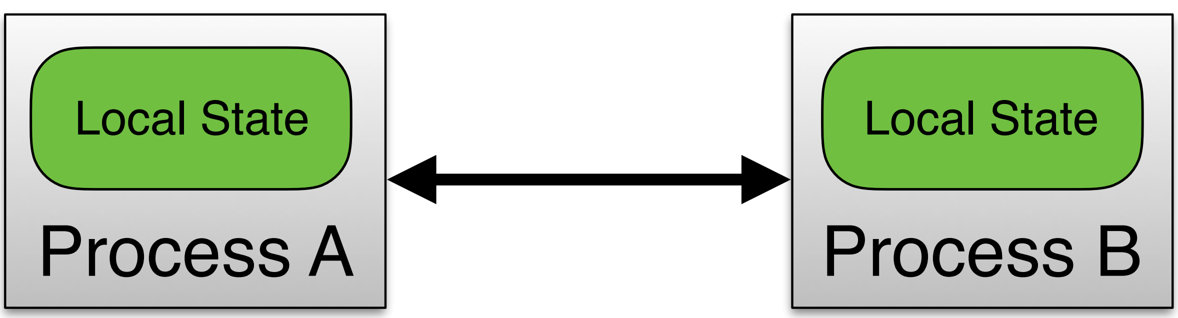
Communication in Distributed Systems

Fundamental problem: How to share information and state among distributed entities (processes) ?



Communication in Distributed Systems

- Components of distributed systems : processes
- Processes can run on different machines
  - Process execution is independent and decoupled
- **How do processes communicate with each other?**
  - Transfer of data (message passing)
  - Transfer of data and control



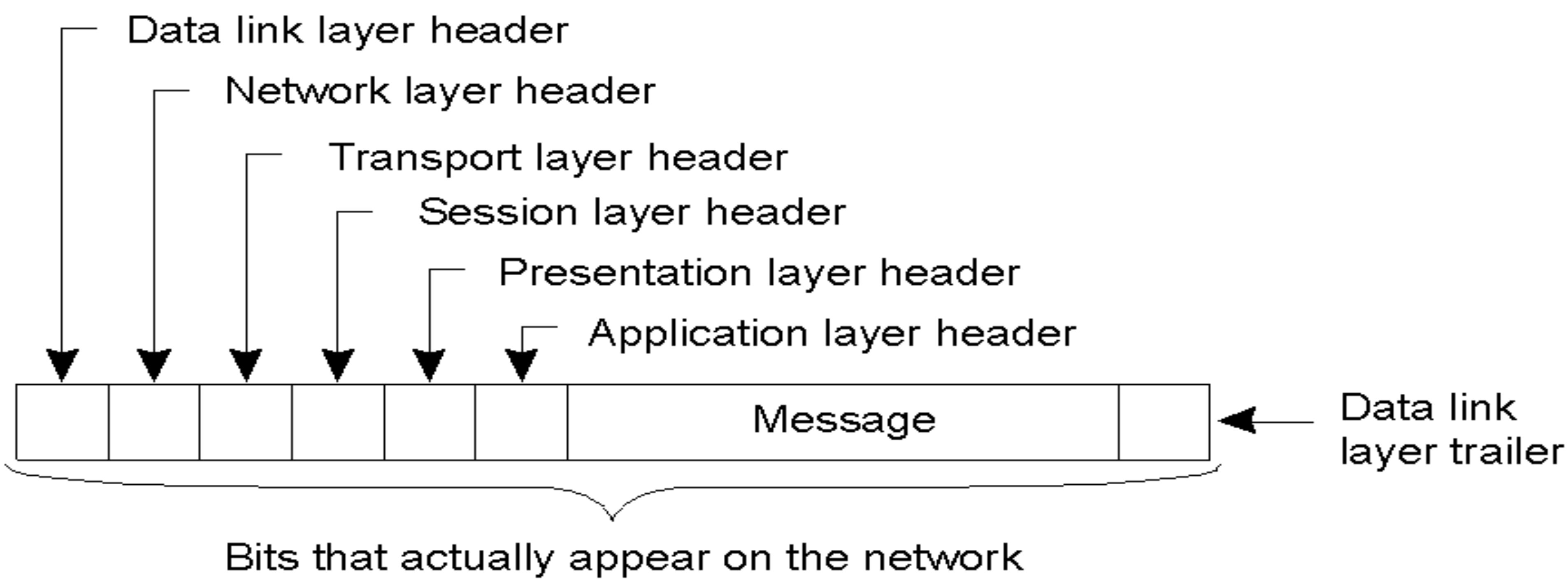
Communication Between Processes

- *Unstructured* communication
  - Use shared memory or shared data structures

- ***Structured* communication**
  - Use explicit messages (IPCs)
  - Communication may be over the network

Communication over the Network

- Processes communicate by sharing messages over a network
- A typical message as it appears on the network.



Messaging in Distributed Systems

- *Message-oriented Communication*
- *Remote Procedure Calls*
  - Transparency but poor for passing references
- Remote Method Invocation
  - RMIs are essentially RPCs but specific to remote objects
  - System wide references passed as parameters
- Stream-oriented Communication

Communication Patterns

- Client-pull architecture
  - Clients pull data from servers (by sending requests)
  - Example: HTTP
  - Pro: stateless servers, failures are easy to handle
  - Con: limited scalability
- Server-push architecture
  - Servers push data to client
  - Example: video streaming, stock tickers
  - Pro: more scalable, Con: stateful servers, less resilient to failure
- **When/how-often to push or pull?**

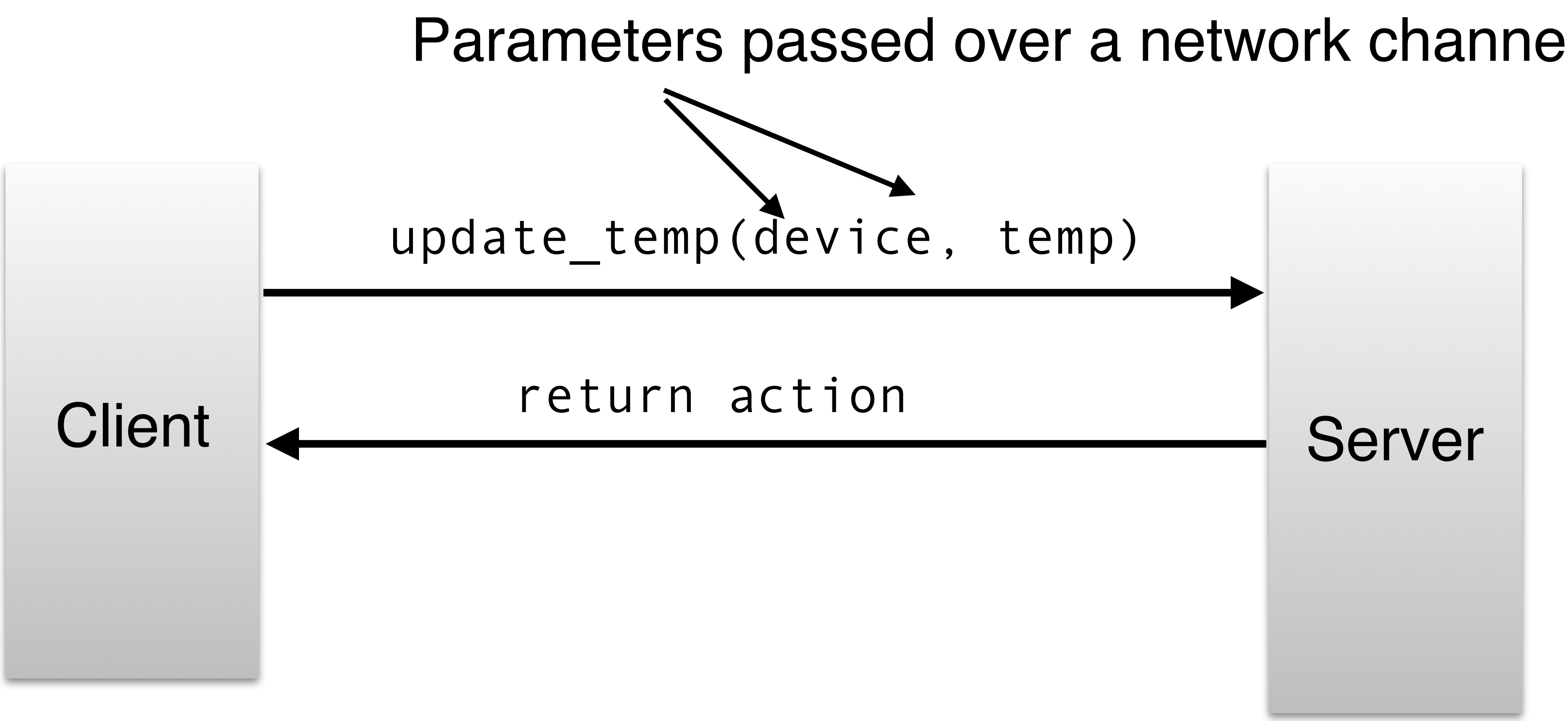
Group Communication

- One-to-many communication: useful for distributed applications
- Issues:
  - Group characteristics:
    - Static/dynamic, open/closed
  - Group addressing
    - Multicast, broadcast, application-level multicast (unicast)
  - Atomicity
  - Message ordering
  - Scalability

Remote Procedure Calls

- Procedure (function) calls a well known and understood mechanism for transfer of data and control within a program/process
- Remote Procedure Calls : extend conventional local calls to work *across* processes.
  - Processes may be running on different machines
  - Allows communication of data via function parameters and return values
  - RPC invocations also serve as notifications (transfer of control)

RPC Example


Parameters passed over a network channel
update_temp(device, temp)
return action
Client
Server

RPC Advantages

- Clean and simple to understand semantics similar to local procedure calls
- Generality: all languages have local procedure calls
  - RPC libraries augment the procedure call interface to make RPCs appear similar to local calls

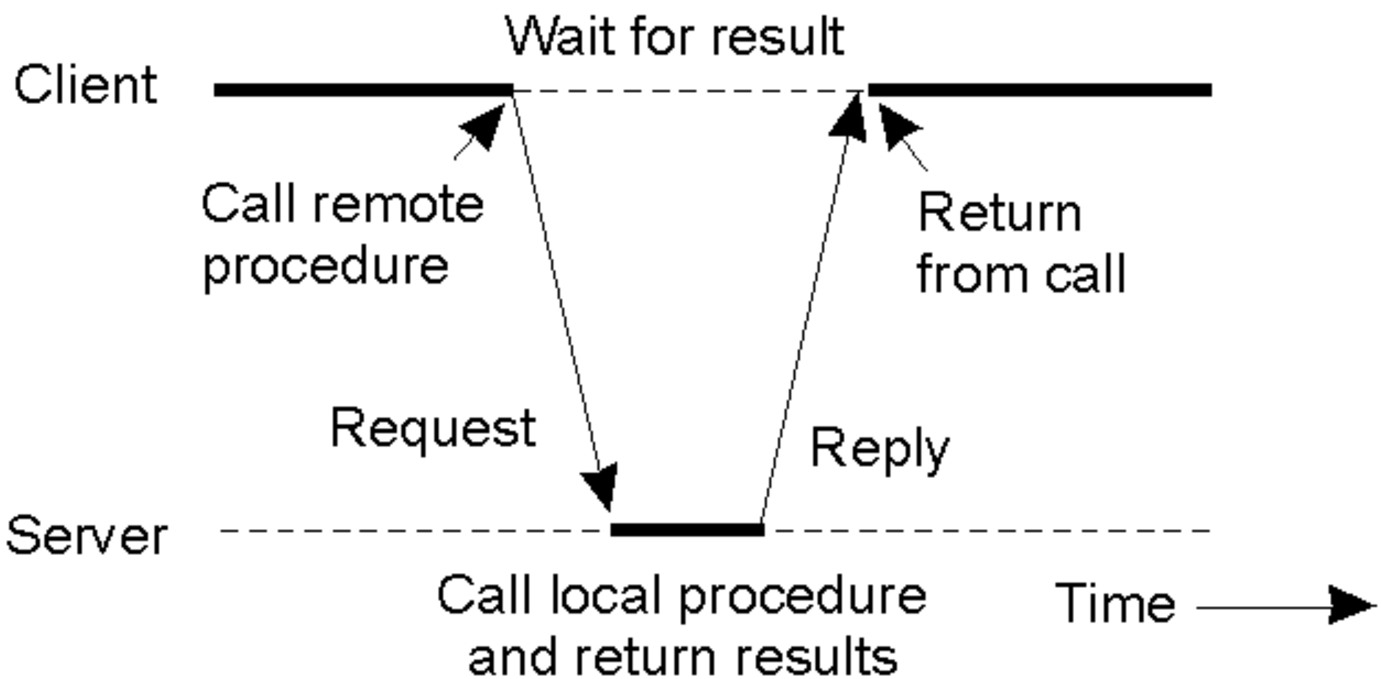
```
push_temp(name) {
    t = get_current_temp();
    return update_temp (name, t);   //RPC
    }
```

Challenges

- RPCs impose new challenges not faced in local calls
- How to pass parameters?
  - Passing data over a network raises issues like endian-ness
  - Pointers: machines may not share an address space
- How to deal with machine failures?
  - Local procedures are assumed to always run
  - A remote machine running an RPC may face crashes, network issues
  - Need to consider failure semantics in RPC implementations
- How to integrate RPCs with existing language runtimes?
  - Seamless local and remote calls
  - Integrate RPCs with language caller/callee interface

RPC Semantics

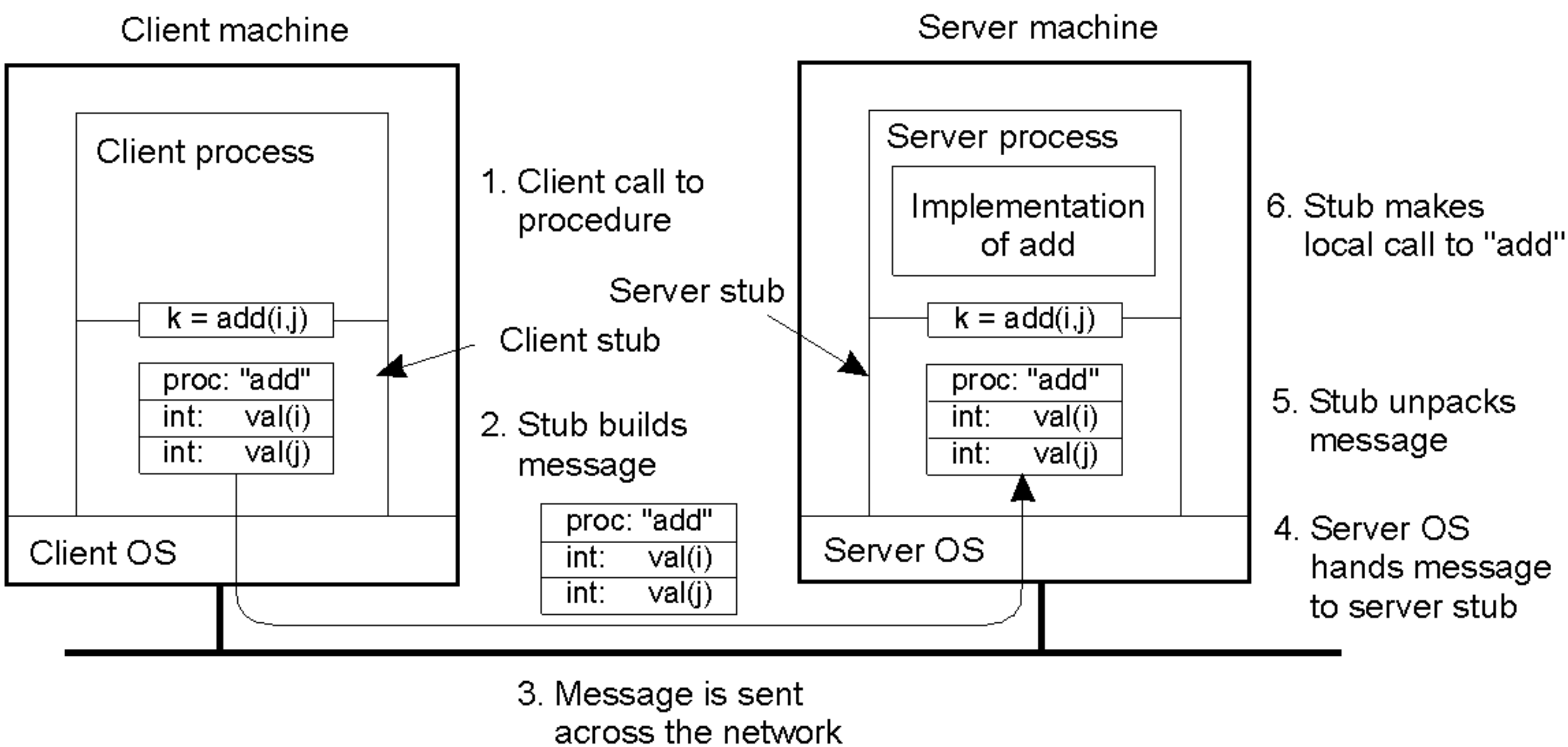
- Principle of RPC between a client and server program [Birrell&Nelson 1984]



How RPCs Work

- Each process has 2 additional components: code stubs and RPC runtime
- Code stubs "translate" local calls remote calls
  - Pack/unpack parameters
- RPC runtime transmits these translated calls over the network
  - Wait for result

How RPCs Work



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

- 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 parameters 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

Marshalling

- Problem: different machines have different data formats
  - Intel: little endian, SPARC: big endian
- Solution: use a cross-platform, general, 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 (serialization of parameters)

Binding

- Problem: how does a client locate a server?
  - How does caller code locate and call the callee
  - Use bindings (similar to how symbols are bound to variables during run-time in local programs)
- 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 Information

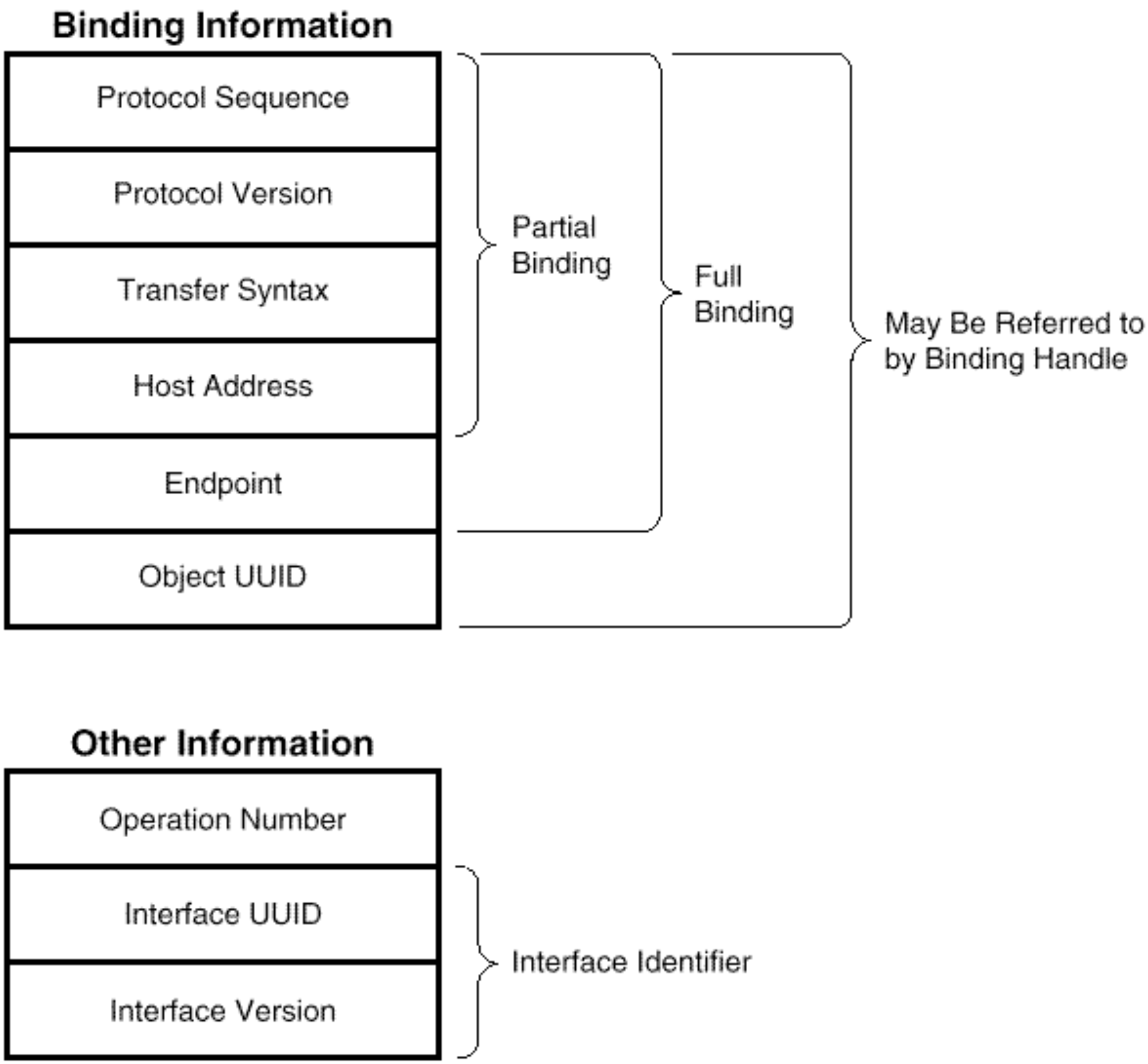


Figure 2-1 Information Required to Complete an RPC

Binding: Comments

- Binding is at **run-time**
  - Better handling of partial failures (clients can try other advertised end-points, protocols, etc.)
  - Increased dynamism
- 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 / Idempotent (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

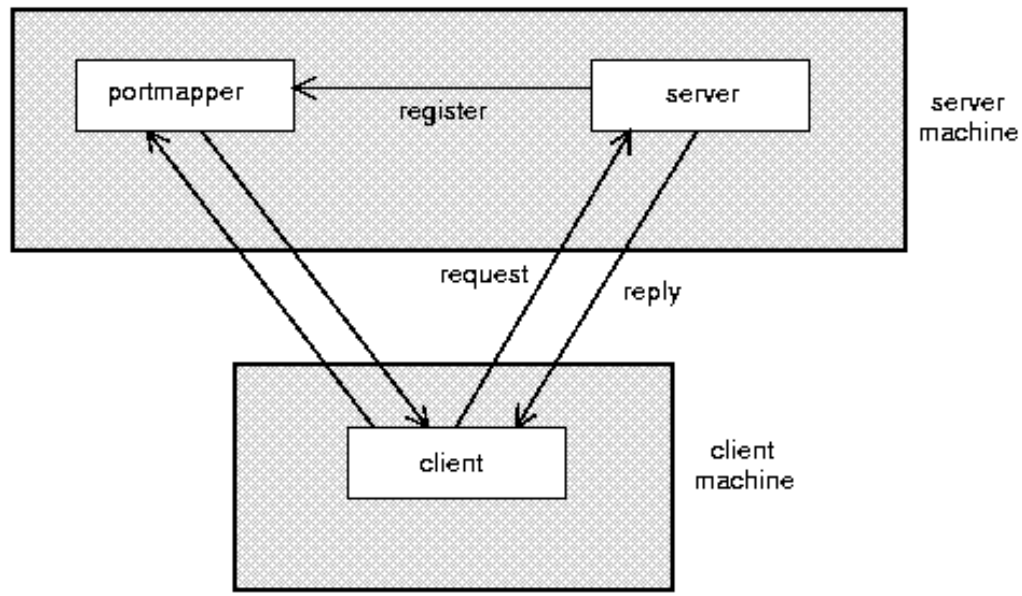
- Choice of protocol [affects communication costs]
  - Use existing protocol (UDP) or design from scratch
  - Packet size restrictions
  - Reliability in case of multiple packet messages
  - Flow control
- Copying costs are dominant overheads
  - Need at least 2 copies per message
    - From client to 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

Case Study: SUNRPC

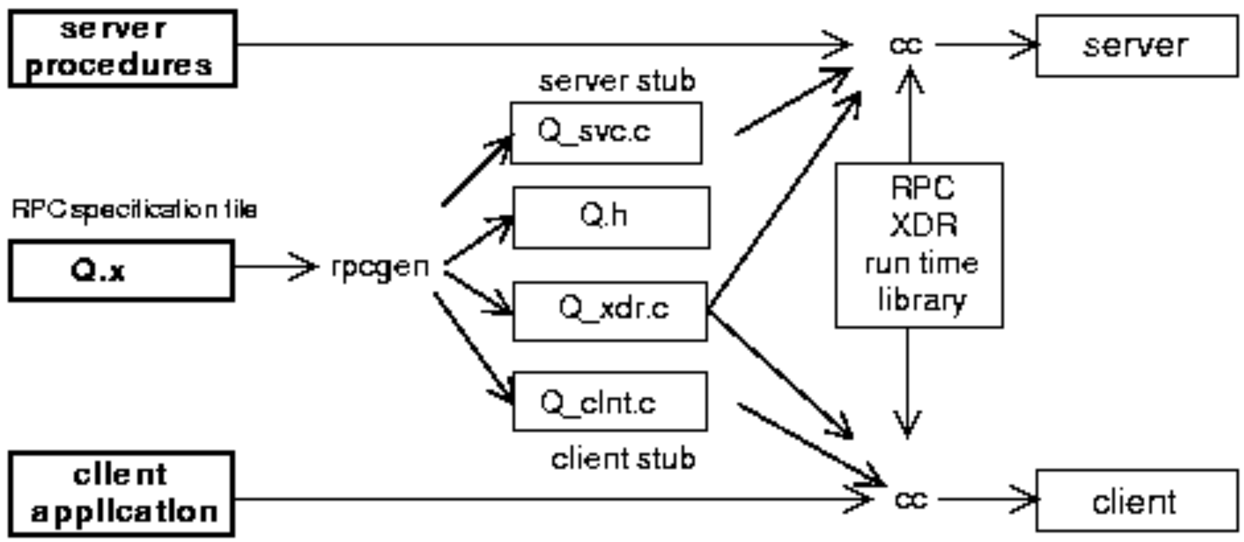
- One of the most widely used RPC systems
- Developed for use with NFS
- Built on top of UDP or TCP
  - TCP: stream is divided into records
  - UDP: max packet size < 8912 bytes
  - UDP: timeout plus limited number of retransmissions
  - TCP: return error if connection is terminated by server
- Multiple 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: add rpc

Modern RPCs & Protocol Buffers

- Many distributed systems use RPCs today (Mesos)
- Common paradigm: serialize function calls in some serialization format (XML, JSON,…) and send over HTTP (xmlrpclib, etc.)
- HTTP servers unpacks and executes the remote call
- For serialization, **protocol-buffers** are typically used
  - Compact, binary format
  - Faster to serialize and deserialize
  - Multi-language support.

Summary

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