Overview of Remote Procedure Calls (RPC)

Douglas C. Schmidt

Washington University, St. Louis

http://www.cs.wustl.edu/~schmidt/

schmidt@cs.wustl.edu

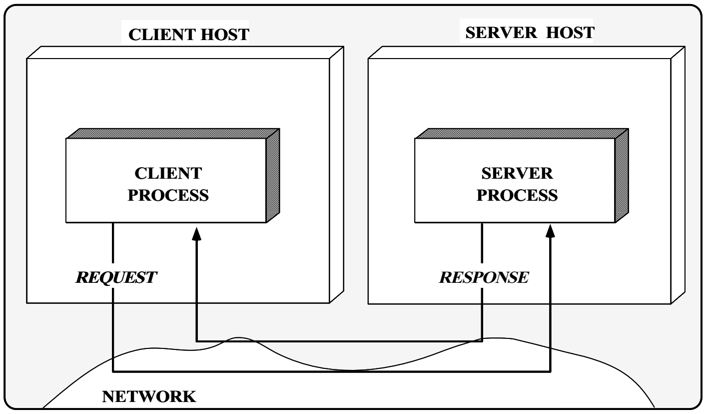
Introduction

- Remote Procedure Calls (RPC) are a popular model for building client/server applications
  - ONC RPC and OSF DCE are widely available RPC toolkits
- RPC forms the basis for many client/server applications
  - *e.g.*, NFS
- Distributed object computing (DOC) frameworks may be viewed as an extension of RPC (RPC on steriods)
  - *e.g.*, OMG CORBA
- RPC falls somewhere between the transport layer and application layer in the OSI model
  - *i.e.*, it contains elements of session and presentation layers

Motivation

- RPC tries to simplify distributed application programming by making distribution *transparent*
- RPC toolkits automatically handle
  - *Reliability*
    - ▹ *e.g.*, communication errors and transactions
  - *Platform heterogeneity*
    - ▹ *e.g.*, performs parameter "marshaling" of complex data structures and handles byte-ordering differences
  - *Service location and selection*
  - *Service activation and handler dispatching*
  - *Security*

IPC Overview



- Many applications require communication among multiple processes
  - Processes may be remote or local

Message Passing Model

- Message passing is a general technique for exchanging information between two or more processes
- Basically an extension to the `send`/`recv` I/O API
  - *e.g.*, UDP, VMTP
- Supports a number of different communication styles
  - *e.g.*, request/response, asynchronous oneway, multicast, broadcast, etc.
- May serve as the basis for higher-level communication mechanisms such as RPC

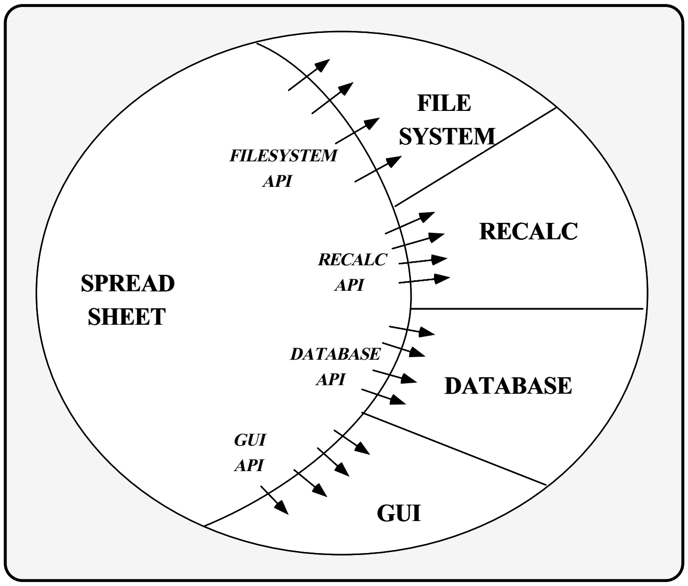
Message Passing Model (cont'd)

- In general, message passing does not make an effort to hide distribution
  - *e.g.*, network byte order, pointer linearization, addressing, and security must be dealt with explicitly
- This makes the model efficient and flexible, but also complicate and time consuming

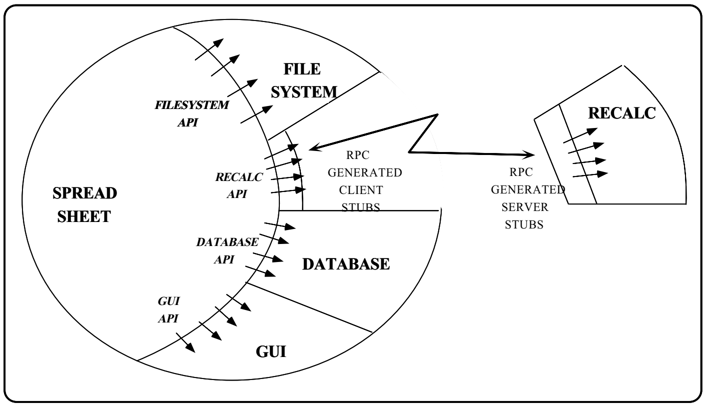
Message Passing Design Considerations

- *Blocking vs. nonblocking*
  - Affects reliablility, responsiveness, and program structure
- *Buffered vs. unbuffered*
  - Affects performance and reliability
- *Reliable vs. unreliable*
  - Affects performance and correctness

Monolithic Application Structure



RPC Application Structure



- Note, RPC generators automate most of the work involved in separating client and server functionality

Basic Principles of RPC

1. Use traditional programming style for distributed application development

2. Enable selective replacement of local procedure calls with remote procecure calls

   - *Local Procedure Call* (LPC)

     – A well-known method for transferring control from one part of a process to another

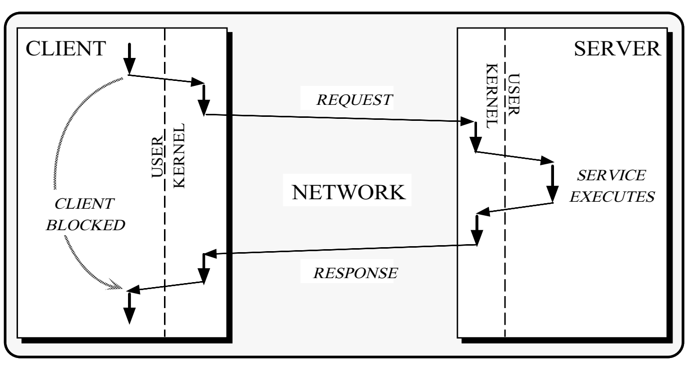
       ▹ Implies a subsequent return of control to the caller

   - *Remote Procedure Call* (RPC)

     – Similar LPC, except a local process invokes a procedure on a remote system

       ▹ *i.e.*, control is transferred *across* processes/hosts

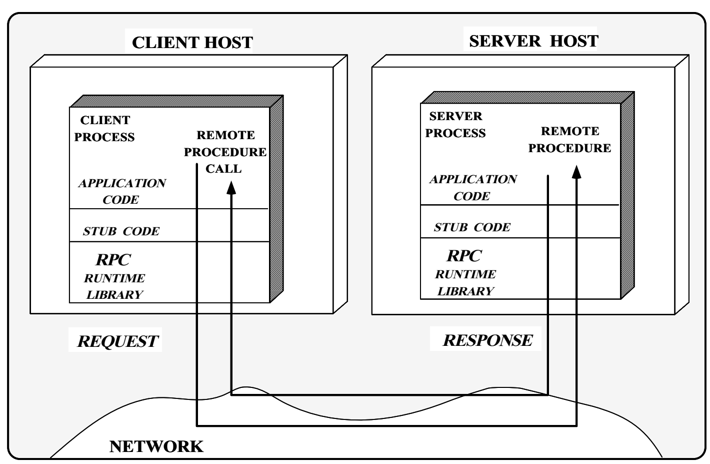
A Temporal View of RPC



- An RPC protocol contains two sides, the *sender* and the *receiver* (*i.e.*, *client* and *server*)

  – However, a server might also be a client of another server and so on...

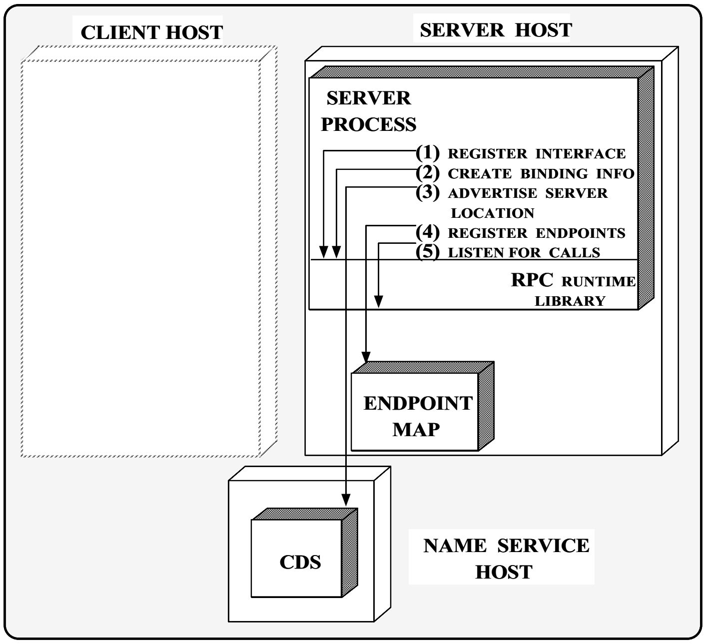
A Layered View of RPC



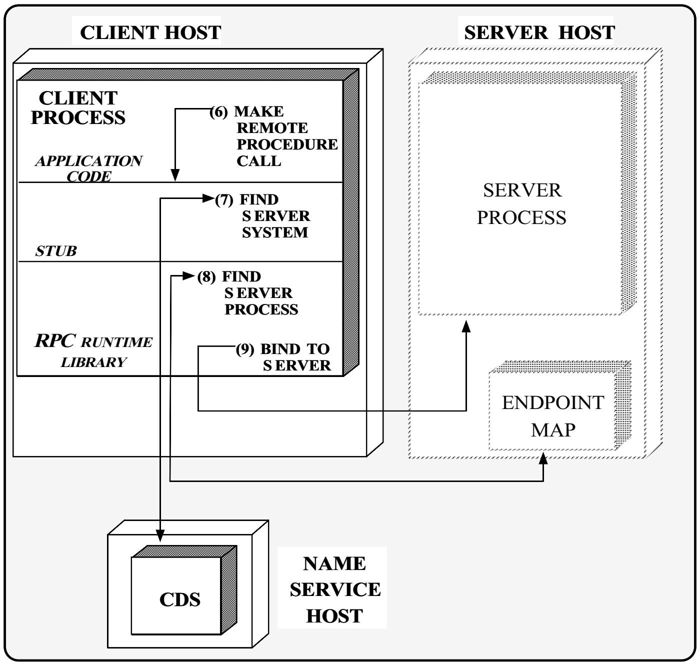
RPC Automation

- To help make distribution transparent, RPC hides all the network code in the client *stubs* and server *skeletons*
  - These are usually generated automatically...
- This shields application programs from networking details
  - *e.g.*, sockets, parameter marshalling, network byte order, timeouts, flow control, acknowledgements, retransmissions, etc.
- It also takes advantage of recurring communcation patterns in network servers to generate most of the stub/skeleton code automatically

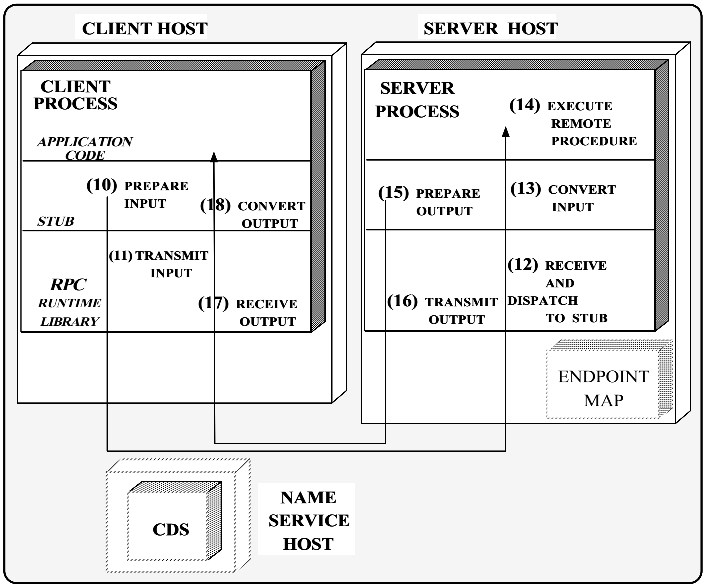
Typical Server Startup Behavior



Typical Client Startup Behavior



Typical Client/Server Interaction



RPC Models

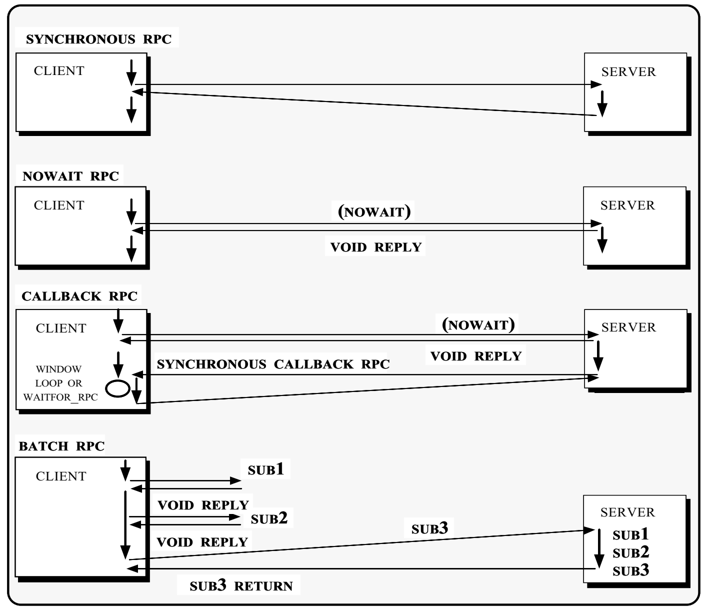
- There are several variations on the standard RPC "synchronous request/response" model

- Each model provides greater flexibility, at the cost of less transparency

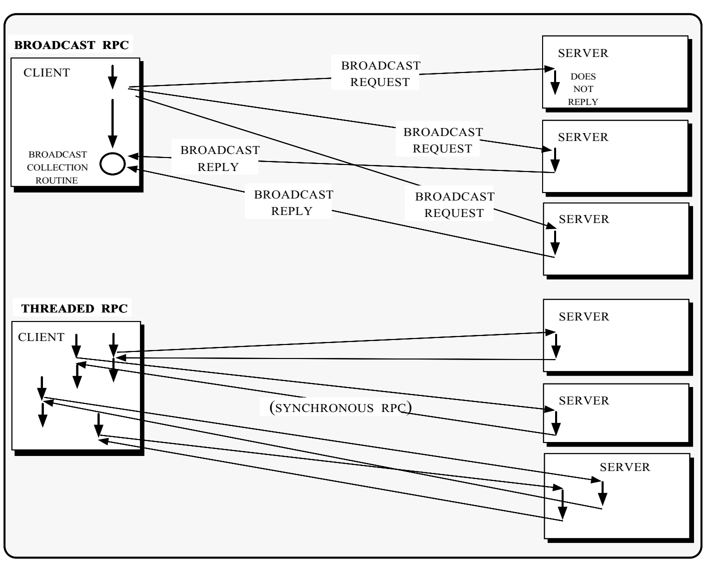
- Certain RPC toolkits support all the different models
  - *e.g.*, ONC RPC

- Other DOC frameworks do not (due to portability concerns)
  - *e.g.*, OMG CORBA and OSF DCE

RPC Models



RPC Models (cont'd)



Transparency Issues

- RPC has a number of limitations that must be understood to use the model effectively
  - Most of the limitations center around *transparency*

- Transforming a simple local procedure call into *system calls*, *data conversions*, and *network communications* increases the chance of something going wrong
  - *i.e.*, it reduces the *transparency* of distribution

Tranparency Issues (cont'd)

- Key Aspects of RPC Transparency
  1. *Parameter passing*
  2. *Data representation*
  3. *Binding*
  4. *Transport protocol*
  5. *Exception handling*
  6. *Call semantics*
  7. *Security*
  8. *Performance*

Parameter Passing

- Functions in an application that runs in a single process may collaborate via parameters and/or global variables
- Functions in an application that runs in multiple processes on the same host may collaborate via message passing and/or non-distributed shared memory
- However, passing parameters is typically the only way that RPC-based clients and servers share information
  - Hence, we have already given up one type of transparency...

Parameter Passing (cont'd)

- Passing parameters across process/host boundaries is surprisingly tricky...
- Parameters that are passed by value are fairly simple to handle
  - The client stub copies the value from the client and packages into a network message
  - Presentation issues are still important, however
- Parameters passed by reference are *much* harder
  - *e.g.*, in C when the address of a variable is passed
    - ▹ *e.g.*, passing arrays
  - Or more generally, handling pointer-based data structures
    - ▹ *e.g.*, pointers, lists, trees, stacks, graphs, etc.

Parameter Passing (cont'd)

- Typical solutions include:
  - Have the RPC protocol only allow the client to pass arguments by value
    - ▹ However, this reduces transparency even further!
  - Use a presentation data format where the user specifically defines what the input arguments are and what the return values are
    - ▹ *e.g.*, Sun's XDR routines
  - RPC facilities typically provide an "interface definition language" to handle this
    - ▹ *e.g.*, CORBA or DCE IDL

Data Representation

- RPC systems intended for heterogeneous environments must be sensitive to byte-ordering differences
  - They typically provide tools for automatically performing data conversion (*e.g.*, `rpcgen` or `idl`)
- Examples:
  - *Sun RPC (XDR)*
    - ▷ Imposes "canonical" big-endian byte-ordering
    - ▷ Minimum size of any field is 32 bits
  - *Xerox Courier*
    - ▷ Uses big-endian
    - ▷ Minimum size of any field is 16 bits

Data Representation (cont'd)

- Examples (cont'd)
  - *DCE RPC (NDR)*
    - ▷ Supports multiple presentation layer formats
    - ▷ Supports "receiver makes it right" semantics...
      - · Allows the sender to use its own internal format, if it is supported
    - ▷ The receiver then converts this to the appropriate format, if different from the sender's format
      - · This is more efficient than "canonical" big-endian format for little-endian machines

Binding

- Binding is the process of mapping a request for a service onto a physical server somewhere in the network
  - Typically, the client contacts an appropriate name server or "location broker" that informs it which remote server contains the service
    - ▷ Similar to calling 411...
- If service migration is supported, it may be necessary to perform this operation multiple times
  - Also may be necessary to leave a "forwarding" address

Binding (cont'd)

- There are two components to binding:
  1. Finding a remote host for a desired service
  2. Finding the correct service on the host
     - *i.e.*, locating the "process" on a given host that is listening to a well-known port
- There are several techniques that clients use to locate a host that provides a given type of service
  - These techniques differ in terms of their performance, transparency, accuracy, and robustness

Binding (cont'd)

- "Hard-code" magic numbers into programs (ugh...;-))
- Another technique is to hard-code this information into a text file on the local host
  - *e.g.*, /etc/services
  - Obviously, this is not particularly scalable...
- Another technique requires the client to name the host they want to contact
  - This host then provides a "superserver" that knows the port number of any services that are available on that host
  - Some example super servers are:
    - ▷ **`inetd`** and **`listen`** -- ID by port number
    - ▷ **`tcpmux`** -- ID by name (*e.g.*, "ftp")

Binding (cont'd)

- Superserver: `inetd` and `listen`
  - *Motivation*
    - ▷ Originally, system daemon processes ran as separate processes that started when the system was booted
    - ▷ However, this increases the number of processes on the machine, most of which are idle much of the time
  - *Solution* → superserver
    - ▷ Instead of having multiple daemon processes asleep waiting for communication, **`inetd`** or **`listen`** listens on behalf of all of them and dynamically starts the appropriate one "on demand"
      - · *i.e.*, upon receipt of a service request

Binding (cont'd)

- Superservers (cont'd)
  - This reduces total number of system processes
  - It also simplifies writing of servers, since many start-up details are handled by **`inetd`**
    - ▷ *e.g.*, **`socket`**, **`bind`**, **`listen`**, **`accept`**
  - See /etc/inetd.conf for details...
  - Note that these super servers combine several activities
    - ▷ *e.g.*, binding and execution

Binding (cont'd)

- *Location brokers and traders*
  - These more general techniques maintain a distributed database of "service → server" mappings
  - Servers on any host in the network register their willingness to accept RPCs by sending a special *registration* message to a mapping authority, *e.g.*,

    `portmapper` -- ID by PROGRAM/VERSION number
    `orbixd` -- ID by "interface"
  - Clients contact the mapping authority to locate a particular service
    - ▷ Note, one extra level of indirection...

Binding (cont'd)

- *Location brokers and traders*
  - A location broker manages a hierarchy consisting of pairs of names and object references
    - ▹ The desired object reference can be found if its name is known
  - A trader service can locate a suitable object given a set of attributes for the object
    - ▹ *e.g.*, supported interface(s), average load and response times, or permissions and privileges
  - The location of a broker or trader may be set via a system administrator or determined via a name server discovery protocol
    - ▹ *e.g.*, may use broadcast or multicast to locate name server...

Transport Protocol

- Some RPC implementations use only a single transport layer protocol
  - Others allow protocol section either implicitly or explicitly
- Some examples:
  - *Sun RPC*
    - ▹ Earlier versions support only UDP, TCP
    - ▹ Recent versions are "transport independent"
  - *DCE RPC*
    - ▹ Runs over many, many protocol stacks
    - ▹ And other mechanisms that aren't stacks
      - · *e.g.*, shared memory
  - *Xerox Courier*
    - ▹ SPP

Transport Protocol (cont'd)

- When a connectionless protocol is used, the client and server stubs must explicitly handle the following:

1. Lost packet detection (*e.g.*, via timeouts)
2. Retransmissions
3. Duplicate detection

- This makes it difficult to ensure certain RPC reliability semantic guarantees
- A connection-oriented protocol handles some of these issues for the RPC library, but the overhead may be higher when a connection-oriented protocol is used
  - *e.g.*, due to the connection establishment and termination overhead

Exception Handling

- With a local procedure call there are a limited number of things that can go wrong, both with the call/return sequence and with the operations
  - *e.g.*, invalid memory reference, divide by zero, etc.
- With RPC, the possibility of something going wrong increases, *e.g.*,

1. The actual remote server procedure itself generate an error
2. The client stub or server stub can encounter network problems or machine crashes

- Two types of error codes are necessary to handle two types of problems

1. Communication infrastructure failures
2. Service failures

Exception Handling (cont'd)

- Both clients and servers may fail independently
  - If the client process terminates after invoking a remote procedure but before obtaining its result, the server reply is termed an *orphan*
- Important question: "how does the server indicate the problems back to the client?"
- Another exception condition is a request by the client to stop the server during a computation

Exception Handling (cont'd)

- DCE and CORBA define a set of standard "communication infrastructure errors"
- For C++ mappings, these errors are often translated into C++ exceptions
- In addition, DCE provides a set of C macros for use with programs that don't support exception handling

Call Semantics

- When a local procedure is called, there is never any question as to how many times the procedure executed
- With a remote procedure, however, if you do not get a response after a certain interval, clients may not know how many times the remote procedure was executed
  - *i.e.*, this depends on the "call semantics"
  - Of course, whether this is a problem or not is "application-defined"

Call Semantics (cont'd)

- When an RPC can be executed any number of times, with no harm done, it is said to be *idempotent*.
  - *i.e.*, there are no harmful side-effects...
  - Some examples of idempotent RPCs are:
    - ▹ Returning time of day
    - ▹ Calculating square root
    - ▹ Reading the first 512 bytes of a disk file
    - ▹ Returning the current balance of a bank account
  - Some non-idempotent RPCs include:
    - ▹ A procedure to append 512 bytes to the end of a file
    - ▹ A procedure to subtract an amount from a bank account

Call Semantics (cont'd)

- Handling non-idempotent services typically requires the server to maintain *state*
- However, this leads to several additional complexities:
  1. When is it acceptable to relinquish the state?
  2. What happens if crashes occur?

Call Semantics (cont'd)

- There are three different forms of RPC call semantics:
  1. *Exactly once* (same as local IPC)
     - Hard/impossible to achieve, because of server crashes or network failures ...
  2. *At most once*
     - If normal return to caller occurs, the remote procedure was executed one time
     - If an error return is made, it is uncertain if remote procedure was executed one time or not at all
  3. *At least once*
     - Typical for idempotent procedures, client stub keeps retransmitting its request until a valid response arrives
     - If client must send its request more than once, there is a possibility that the remote procedure was executed more than once
       - ▷ Unless response is cached...

Call Semantics (cont'd)

- Note that if a connectionless transport protocol is used then achieving "at most once" semantics becomes more complicated
  - The RPC framework must use sequence numbers and cache responses to ensure that duplicate requests aren't executed multiple times
- Note that accurate distributed timestamps are useful for reducing the amount of state that a server must cache in order to detect duplicates

Security

- Typically, applications making local procedure calls do not have to worry about maintaining the integrity or security of the caller/callee
  - *i.e.*, calls are typically made in the same address space
    - ▷ Note that shared libraries may complicate this...
- Local security is usually handled via access control or special process privileges
- Remote security is handled via distributed authentication protocols
  - *e.g.*, Kerberos...

Performance

- Usually the performance loss from using RPC is an order of magnitude or more, compared with making a local procedure call due to

  1. *Protocol processing*
  2. *Context switching*
  3. *Data copying*
  4. *Network latency*
  5. *Congestion*

- Note, these sources of overhead are ubiquitous to networking...

Performance (cont'd)

- RPC also tends to be much slower than using lower-level remote IPC facilities such as sockets directly due to overhead from

  1. *Presentation conversion*
  2. *Data copying*
  3. *Flow control*
     - *e.g.*, stop-and-wait, synchronous client call behavior
  4. *Timer management*
     - Non-adaptive (consequence of LAN upbringing)

- Note, these sources of overhead are typical of RPC...

Performance (cont'd)

- Another important aspect of performance is how the server handles multiple simultaneous requests from clients
  - An iterative RPC server performs the following functionality:

    ```
    loop {
        wait for RPC request;
        receive RPC request;
        decode arguments;
        execute desired function;
        reply result to client;
    }
    ```

  - Thus the RPC server cannot accept new RPC requests while executing the function for the previous request
    - ▷ This is undesirable if the execution of the function takes a long time
      - · *e.g.*, clients will time out and retransmit, increasing network and host load

Performance (cont'd)

- In many situation, a concurrent RPC server should be used:

  ```
  loop {
      wait for RPC request;
      receive RPC request;
      decode arguments;
      spawn a process or thread {
          execute desired function;
          reply result to client;
      }

  }
  ```

- Threading is often preferred since it requires less resources to execute efficiently

Performance (cont'd)

- However, the primary justification for RPC is not just replacing local procedure calls
  - *i.e.*, it is a method for simplifying the development of distributed applications
- In addition, using distribution may provide higher-level improvements in:
  1. *Performance*
  2. *Functionality*
  3. *Reliability*

Performance (cont'd)

- Servers are often the bottleneck in distributed communication
- Therefore, another performance consideration is the technique used to invoke the server every time a client request arrives, *e.g.*,
  - *Iterative* -- server handles in the same process
    - ▷ May reduce throughput and increase latency
  - *Concurrent* -- server forks a new process or thread to handle each request
    - ▷ May require subtle synchronization, programming, and debugging techniques to work successfully
      - · Thread solutions may be non-portable
    - ▷ Note also that multi-threading removes the need for synchronous client behavior...

Summary

- RPC is one of several models for implementing distributed communication
  - It is particular useful for transparently supporting request/response-style applications
  - However, it is not appropriate for all applications due to its performance overhead and lack of flexibility
- Before deciding on a particular communication model it is crucial to carefully analyze the distributed requirements of the applications involved
  - Particularly the tradeoff of security for performance...