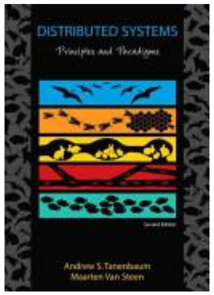
Chapter 4: COMMUNICATION
Part 1-2

Communications in Distributed Systems

Middleware and RPC



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

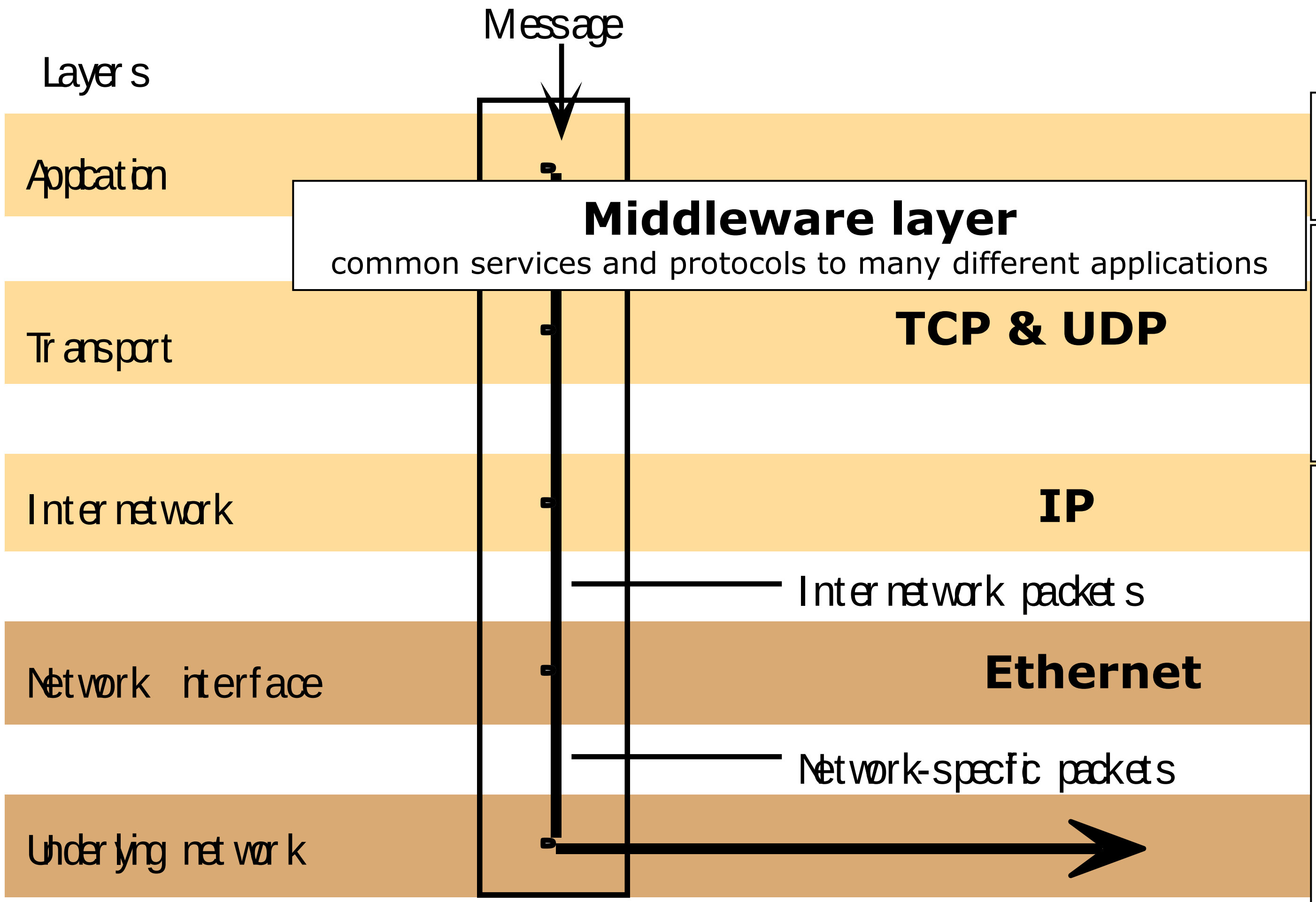
Chapter 4: Communications

- FUNDAMENTALS
  - Layered Protocols (chapter 0)
  - Middleware and Types of communications
- REMOTE PROCEDURE CALL
  - Basic RPC Operation
  - Parameter Passing
  - RPC operation
  - RPC Examples
  - Asynchronous RPC
  - RMI (some from chapter 10, most from web)
  - CORBA (some from chapter 10, most from web)
- MESSAGE-ORIENTED COMMUNICATION
  - Transient and Persistent Communication
- STREAM-ORIENTED COMMUNICATION
  - Support for Continuous Media and Quality of Service
  - Stream Synchronization
- MULTICAST COMMUNICATION
  - Application-Level Multicasting
  - Gossip-Based Data Dissemination

Objectives

- To understand how processes communicate (the heart of distributed systems)
- To understand computer networks and their layers
- To understand client-server paradigm and low-level message passing using **sockets**
- To learn higher-level communication mechanisms
  - RPC, RMI, CORBA
- To understand various forms of communications and their issues
  - Stream-oriented communication, multicast, etc.

Distributed Systems and Layer Structure



Middleware layer

common services and protocols to many different applications

TCP & UDP

IP

Ethernet

Higher-level app protocols

communication facilities for most **distributed systems**

Low-level Layers

An additional communication service in client-server computing

An intermediate (distributed) service in application-level communication

MIDDLEWARE

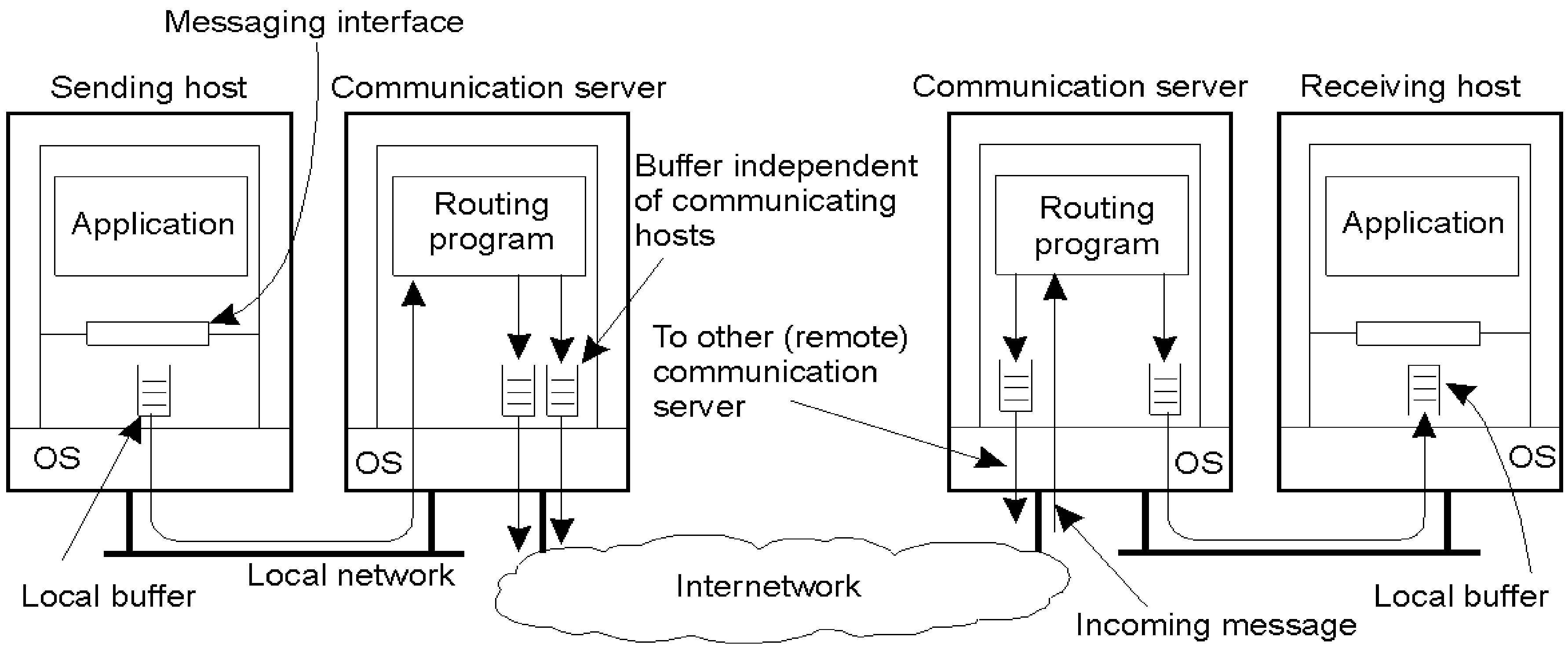
Middleware Layer

- Most network applications have application-specific protocols (FTP, HTTP, DNS) using low-level communication services (TCP, UDP)
- **Middleware Layer** is logically at the application layer but contains a rich set of general-purpose protocols and higher-level communication services (RPC) that can be used by different applications
  - (Un)marshaling of data, necessary for integrated systems
  - Naming protocols to easy sharing of resources (ch5)
  - Distributed locking and commit protocols (ch6, ch8)
  - Scaling mechanisms, such as replication (ch7)
  - Security protocols for secure communication (ch9)

Types of communications (1)
that middleware may offer

Persistent vs. Transient

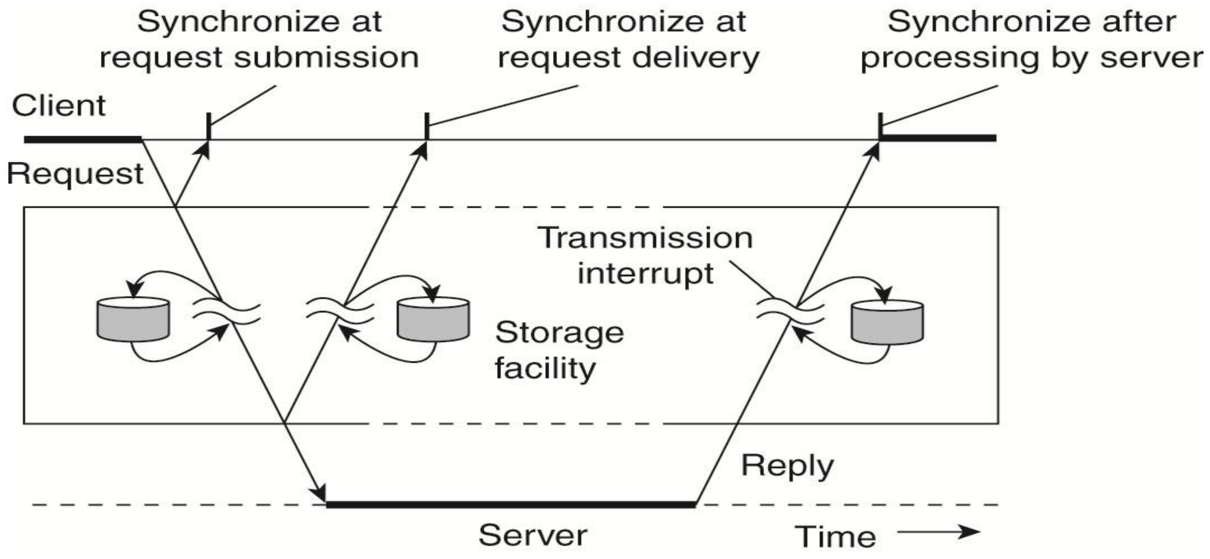
- A message is stored at a communication server as long as it takes to deliver it (e.g., e-mail).
- A message is stored as long as sender and receiver are working at the same time (TCP, UDP, IP routing)



Types of communication (2)

Asynchronous vs. Synchronous

- Sender **continues** immediately after it has submitted the message (Unblocked and Need a local buffer at the sender)
- Sender **blocks** until the sender receives an OK to continue (This OK may come from three places for synchronization)


Synchronize at request submission
Synchronize at request delivery
Synchronize after processing by server
Client
Request
Transmission interrupt
Storage facility
Reply
Server
Time

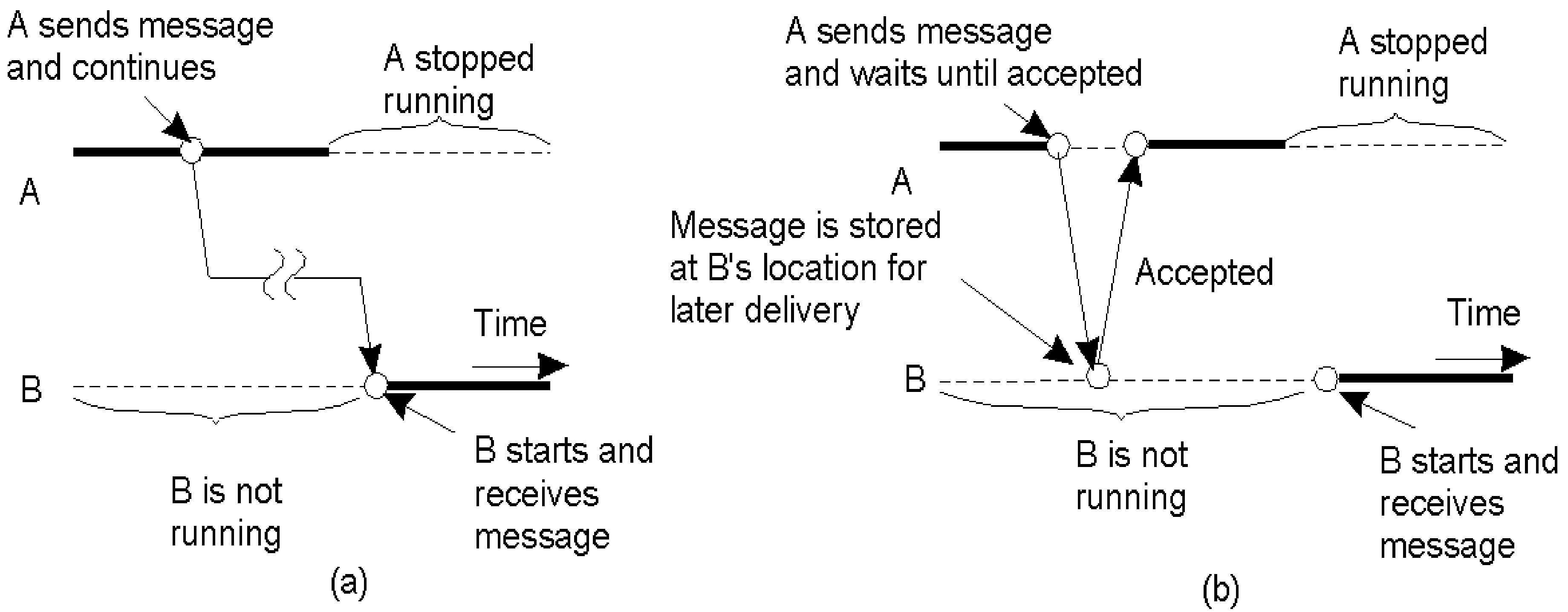
Discrete vs. Streaming

Combinations

	Asynchronous	**Synchronous at** Submission, delivery, after service
Persistent	**Message-oriented middleware (MOM)** • Processes send each other messages (queued) • Sender does not need to wait for immediate reply • Middleware often ensures fault tolerance	
Transient		**Client/Server, RPC, TCP** • 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)

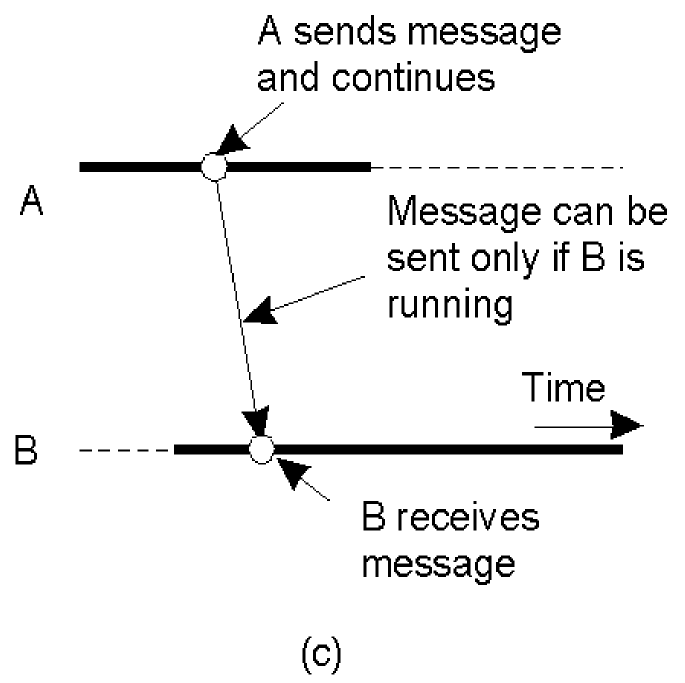
Persistent with Async and Sync

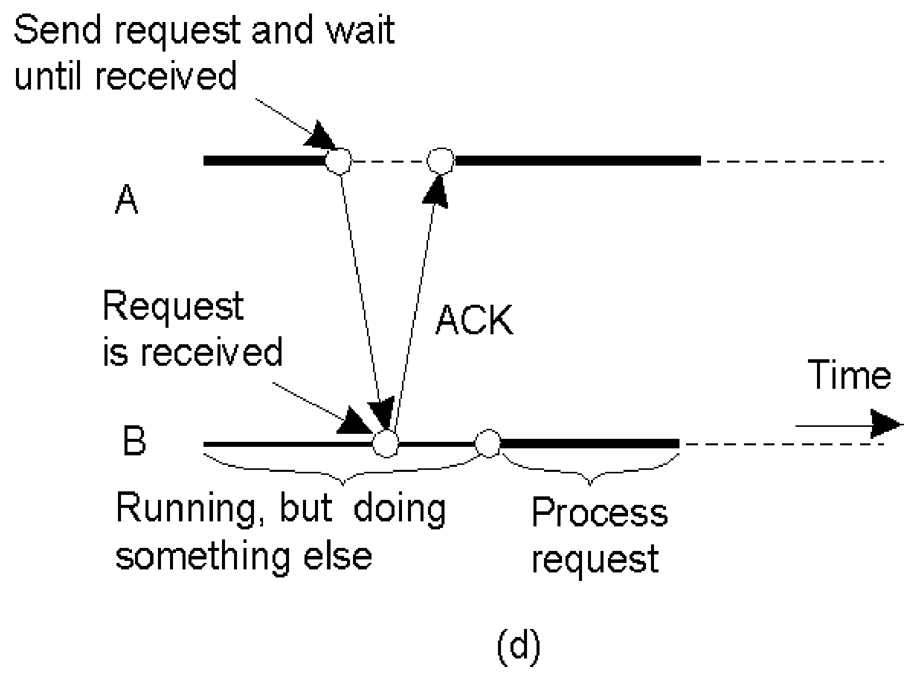

A sends message and continues
A stopped running
A
Time
B
B starts and receives message
B is not running
(a)
A sends message and waits until accepted
A stopped running
A
Message is stored at B's location for later delivery
Accepted
Time
B
B is not running
B starts and receives message
(b)

a) Persistent asynchronous communication (e.g., email)

b) Persistent synchronous communication

Transient with Async and Sync

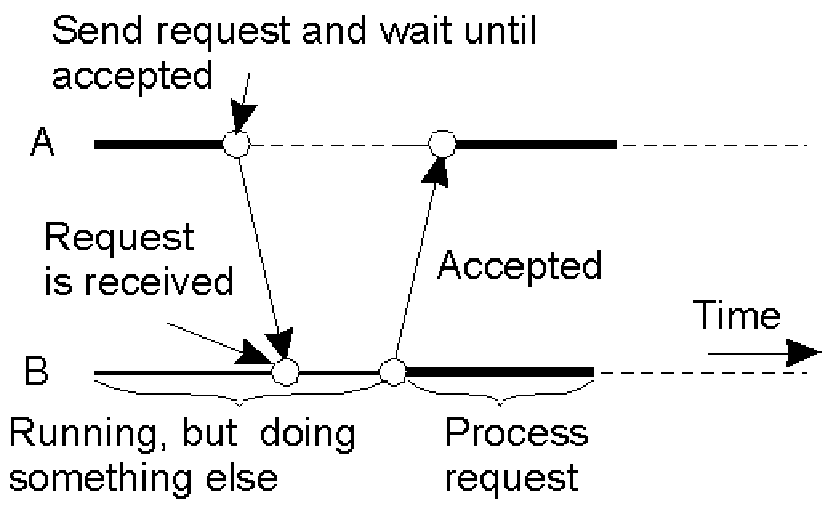

A sends message and continues
A
Message can be sent only if B is running
Time
B
B receives message
(c)


Send request and wait until received
A
Request is received
ACK
Time
B
Running, but doing something else
Process request
(d)

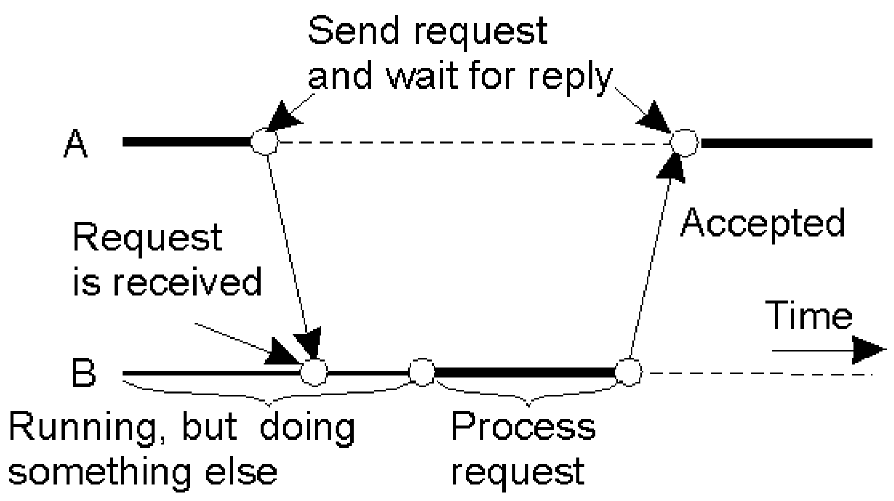
c) Transient asynchronous communication (e.g., UDP)

d) Receipt-based transient synchronous communication (e.g., TCP)

Transient and Sync


Send request and wait until accepted
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(e)


Send request and wait for reply
A
Request is received
Accepted
Time
B
Running, but doing something else
Process request

(f)

e) Delivery-based transient synchronous communication at message delivery (e.g., asynchronous RCP)

f) Response-based transient synchronous communication (RPC)

Remote Procedure Call (RPC) is a high-level model for client-sever communication.

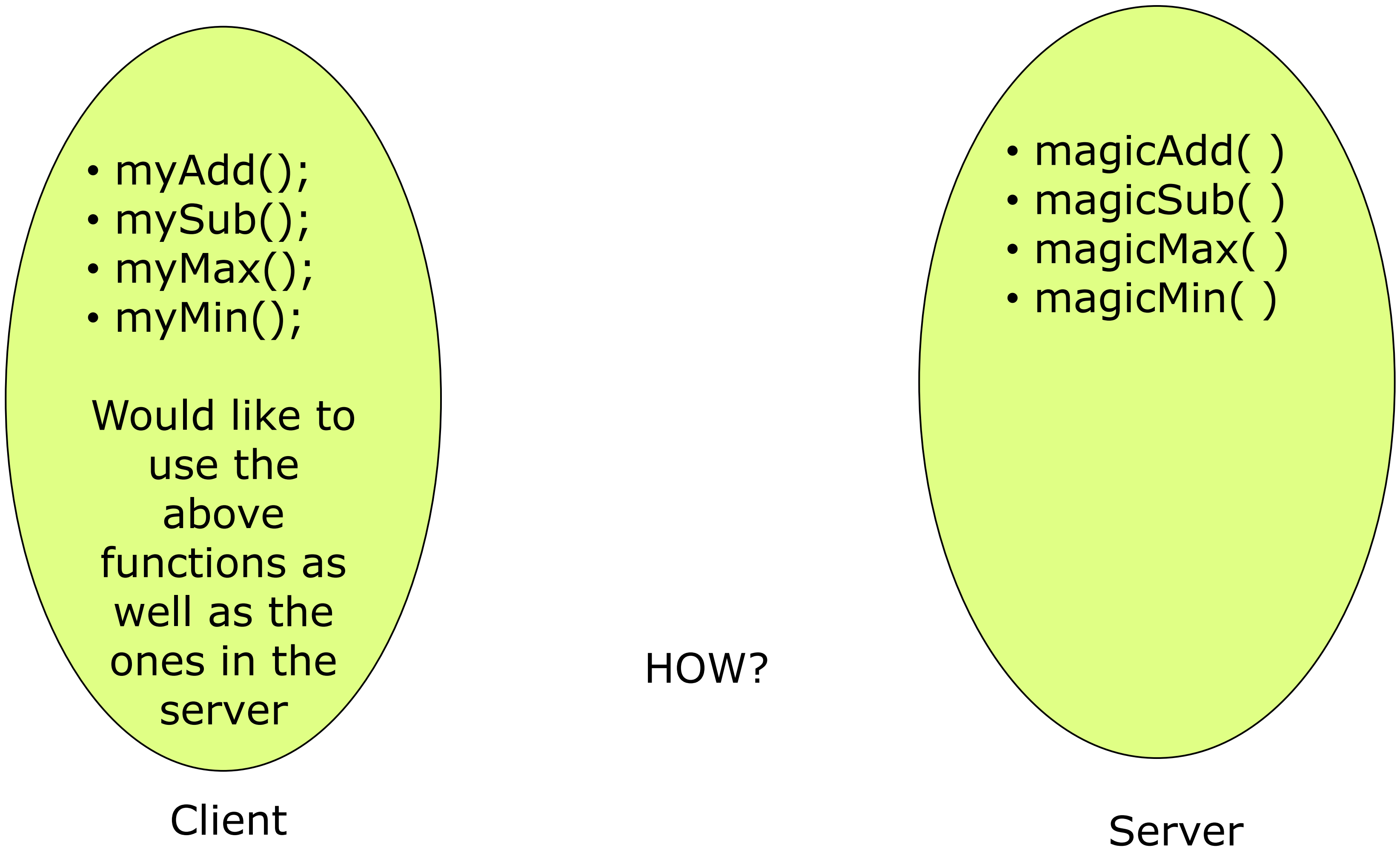
It provides the programmers with a familiar mechanism (e.g., **function calls**) for building distributed systems.

REMOTE PROCEDURE CALLS (RPC)

Remote Procedure Calls

- Send and receive are at the heart of DS, but they fail to achieve transparency
- A new paradigm is needed to hide communications from application programmers
  - Application developers are familiar with simple "*procedure call*" model
  - Well-engineered procedures operate in isolation (black box)
- In 1984, Birrell and Nelson proposed a completely different way of handling communication by allowing programs to call procedures located on other machines (RPC)
  - There is no fundamental reason not to execute procedures on separate machines
  - RPC is integrated into programming languages
  - Makes distributed computing look like centralized computing by allowing remote services to be called as procedures
  - Issues: How to pass parameters, Bindings, Semantics in face of errors, etc…

Remote Procedure Call (RPC)



- myAdd();
- mySub();
- myMax();
- myMin();

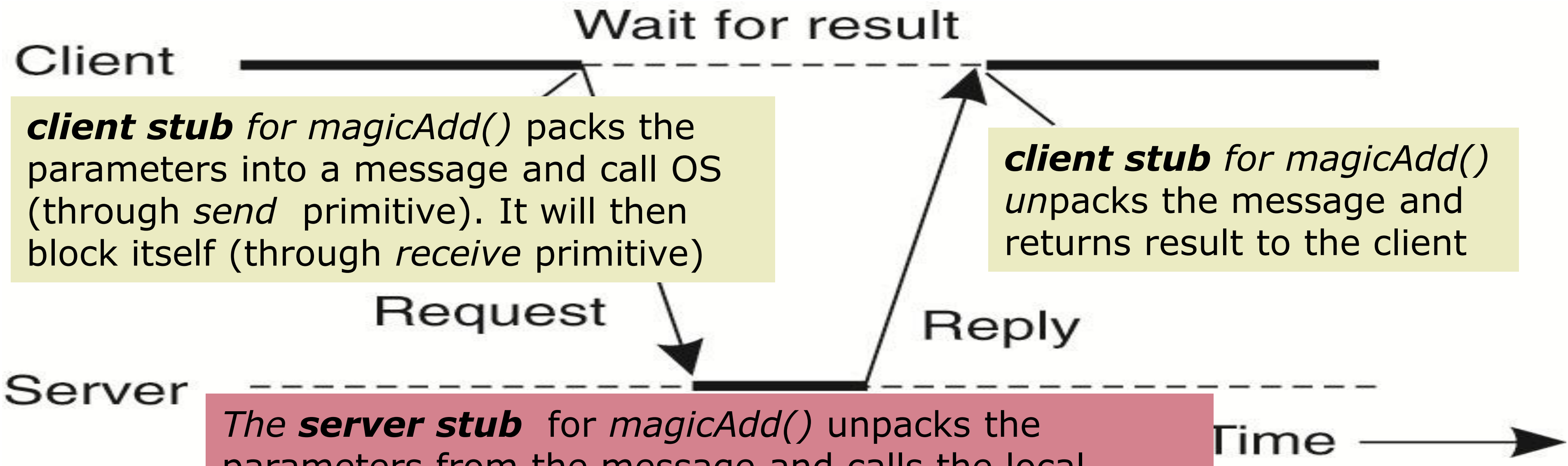
Would like to use the above functions as well as the ones in the server

- magicAdd( )
- magicSub( )
- magicMax( )
- magicMin( )

Client and Server Stubs

- Make remote procedure call look like local call
- So that the client can call myAdd() and magicAdd() in the same way

```
result1 = myAdd(para_x, para_y);
result2 = magicAdd(para_x, para_y);
```



***client stub** for magicAdd()* packs the parameters into a message and call OS (through *send* primitive). It will then block itself (through *receive* primitive)

***client stub** for magicAdd()* unpacks the message and returns result to the client

*The **server stub*** for *magicAdd()* unpacks the parameters from the message and calls the local procedure in a usual way. When completes, the server stub will pack the result and return it to the client

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

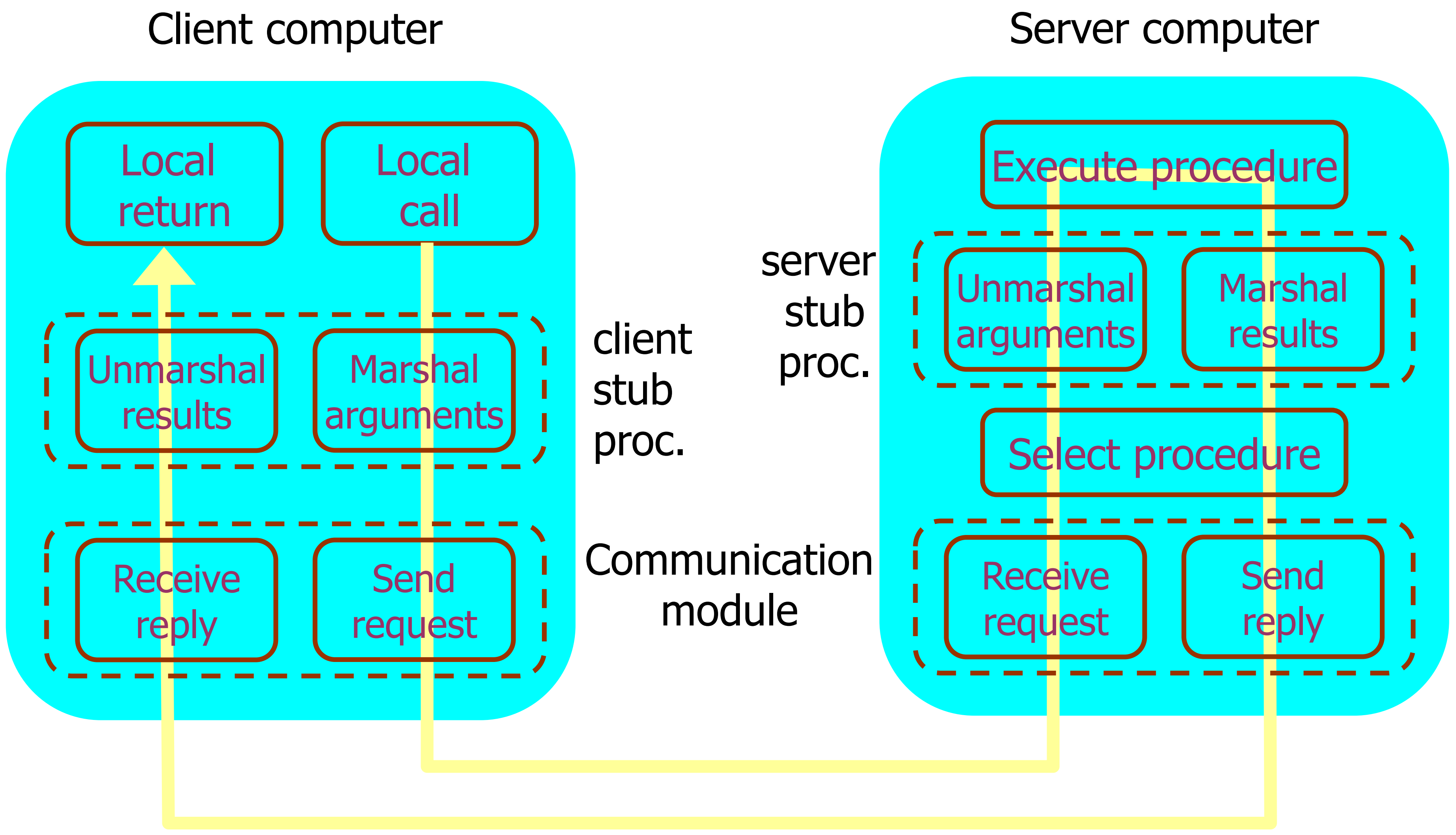
RPC Steps

A remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and calls the local operating system.
3. The client's OS sends the message to the remote OS.
4. The remote OS gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and calls its local OS.
8. The server's OS sends the message to the client's OS.
9. The client's OS gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

Sounds simple but there are several issues…

RPC Mechanism



Simulate “Local procedure parameter passing” through:

- Stubs – proxies
- Flattening – marshalling

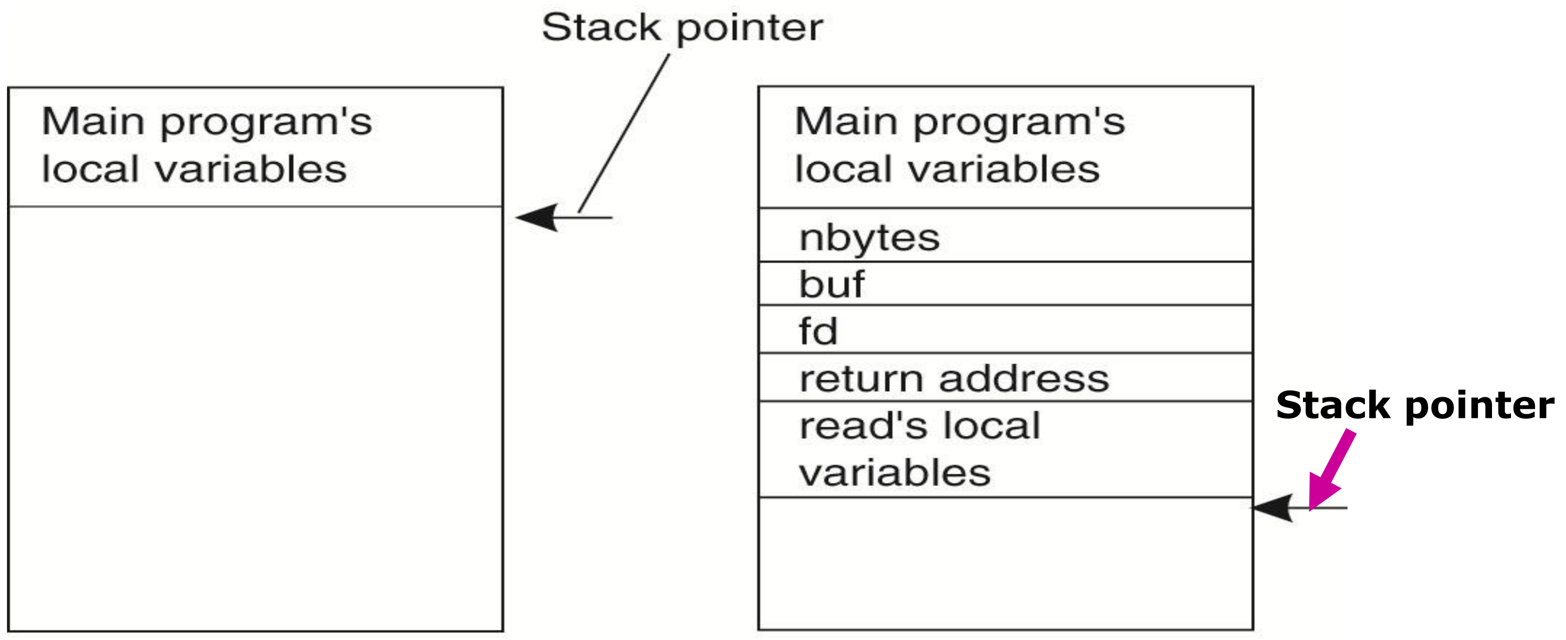
RPC PARAMETER PASSING

Procedure Call: Local

- Consider a call in C:

  result = read(fd, buf, nbytes);

- What happens?
- Caller pushes parameters onto the stack and then…



(a) stack before call (b) stack during call

Parameters Passing in Local Calls

- **Call-by-value**

  the parameter value copied to the stack. Modifications do not affect the calling side

- **Call-by-reference**

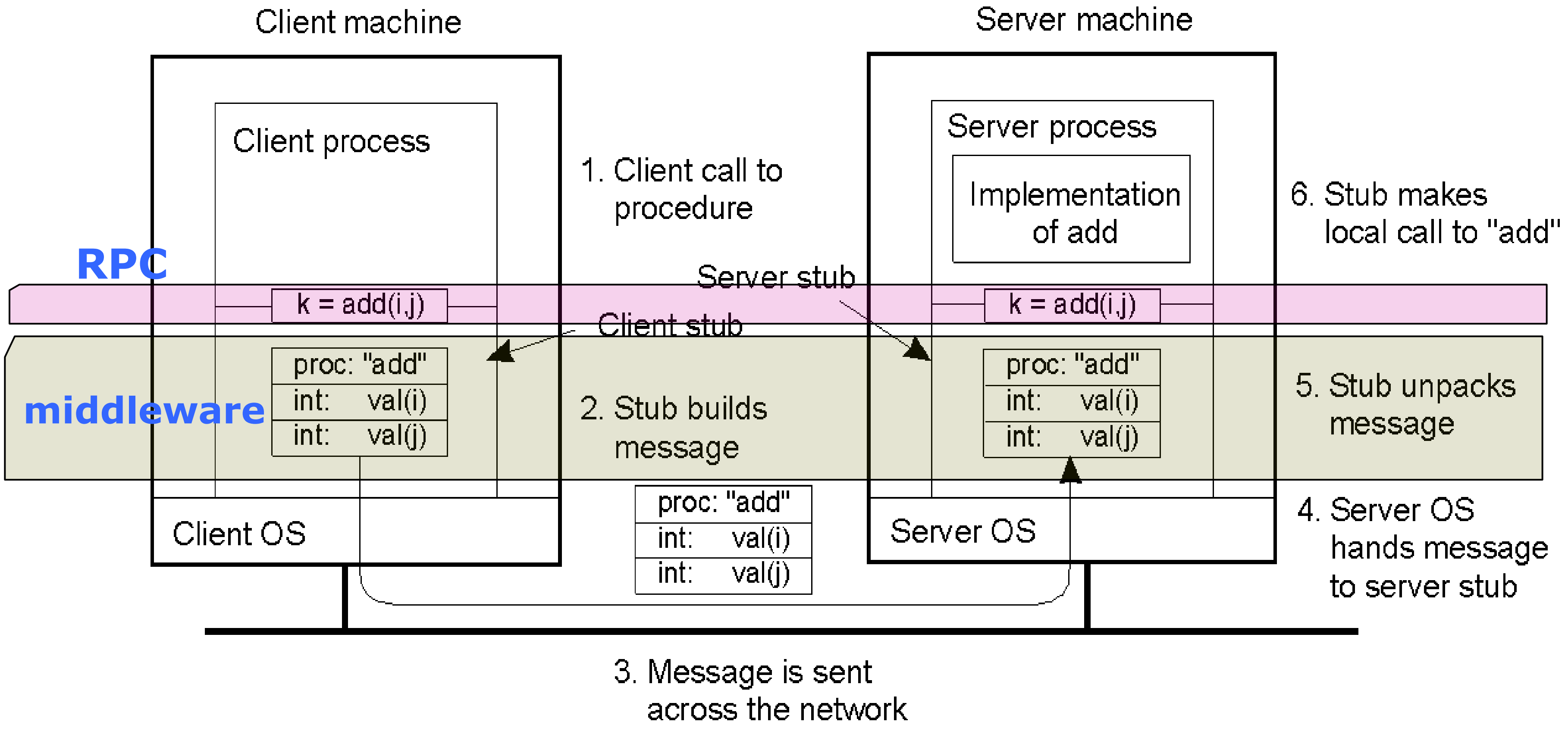
  the address of the parameter is pushed onto the stack (e.g., pointers in C; obj ref in Java). Any modification affects the variable at the calling side.

- **Call-by-copy/restore**

  Copy the variable first to the stack (as in call-by-value), and then copy back after the call, overwriting the caller’s original value.

  - In many cases, the same behavior as “call-by-reference”
  - When a given variable appears multiple times in the parameter list, its behavior may be different than that of “call-by-reference”

Passing Value Parameter in RPC
call-by-value

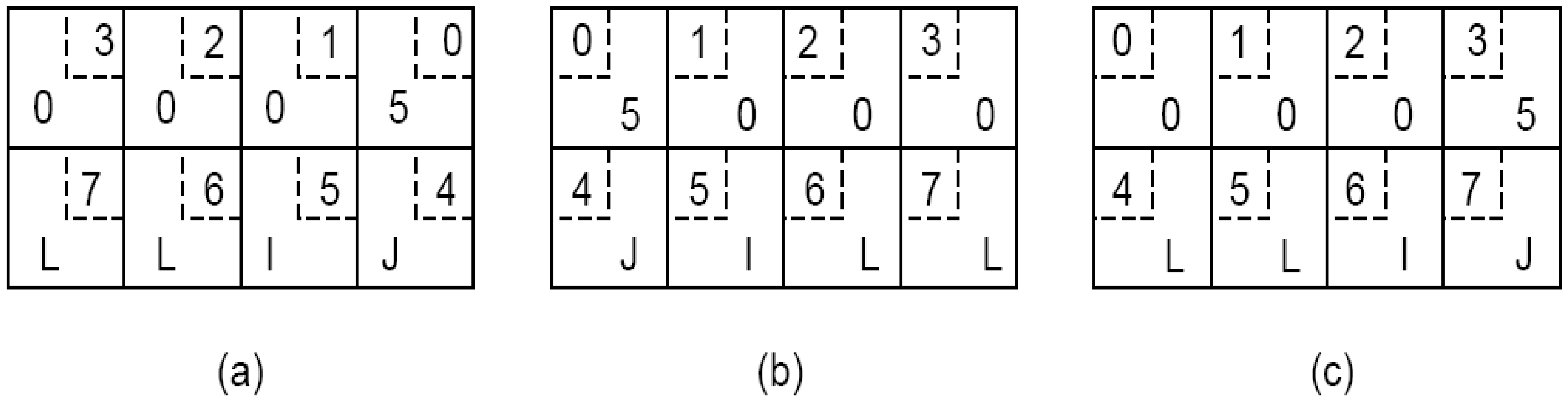


pack value parameters into a message and send it to the server,

would it be that easy?

Problem: different data representations

- A process on an Intel machine send a message of an integer and four-character string ("5, JILL") to another process on a Sun SPARC machine



- (a) original message on Intel (x86, Little Endian)
- (b) receipt message on SPARC (Big Endian) "5000, JILL"
- (c) simple **reverse**: message after converted "5, LLIJ"

The little numbers in boxes indicate the address of each byte

Approaches for Exchanging Information

How can clients make servers on different machines understand them?

How people from different countries communicate with each other?

Speaker uses listener's language

- English, Chinese, India(?)…, //hundreds

→ Clients send information in servers' data representation

Both use a common language

- English

→ Client and server use a common external data representation

- Language defined representation (e.g., Java, CORBA CDR)
- External self-descriptive data representation (e.g., XML, Web Services)

Parameter marshaling

- More than just wrapping parameters into a message
- Client and server machines may have different data representations (think of byte ordering)
- Client and server have to agree on a standard representation (e.g., external data representation (XDR))
  - 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, and transform them into machine-dependent representations.

Passing Reference Parameters in RPC
Difficulty Problem!

Solutions

- Forbid reference parameters!
- Client stub can copy the entire data structure
  - E.g., an entire array may be sent if the size is known
  - Server stub saves changes and sends it back

    (call-by-reference is replaced by call-by-copy/restore)
  - If we know it was just **in** or **out**, we can avoid one copy
- How to handle open-ended data structures (e.g., link list, graphs)?
  - Prohibit
  - A remote reference (chase pointers on network)
    - Remote reference offers unified access to remote data
    - Remote references can be passed as parameter in RPCs (Java RMI)

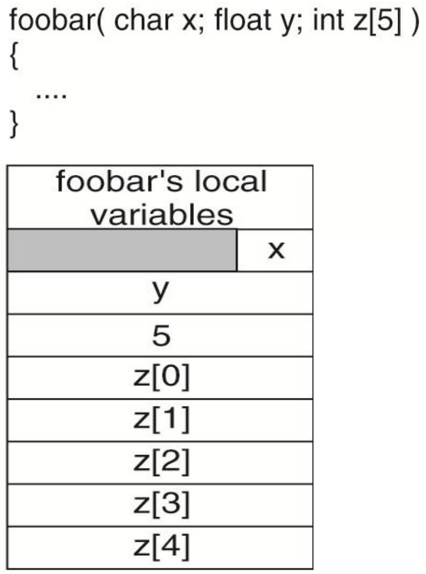
Full access **transparency** cannot be realized.

Parameter Passing Semantics in RPC

- RPC assumes **copy in/copy out** semantics
  - while procedure is executed, nothing can be assumed about parameter values
- RPC assumes all data that is to be operated on is passed by parameters.
- Global variables are not allowed in RPCs
- **Conclusion:**
  - full access transparency cannot be realized.

Parameter Specification and Stub Generation

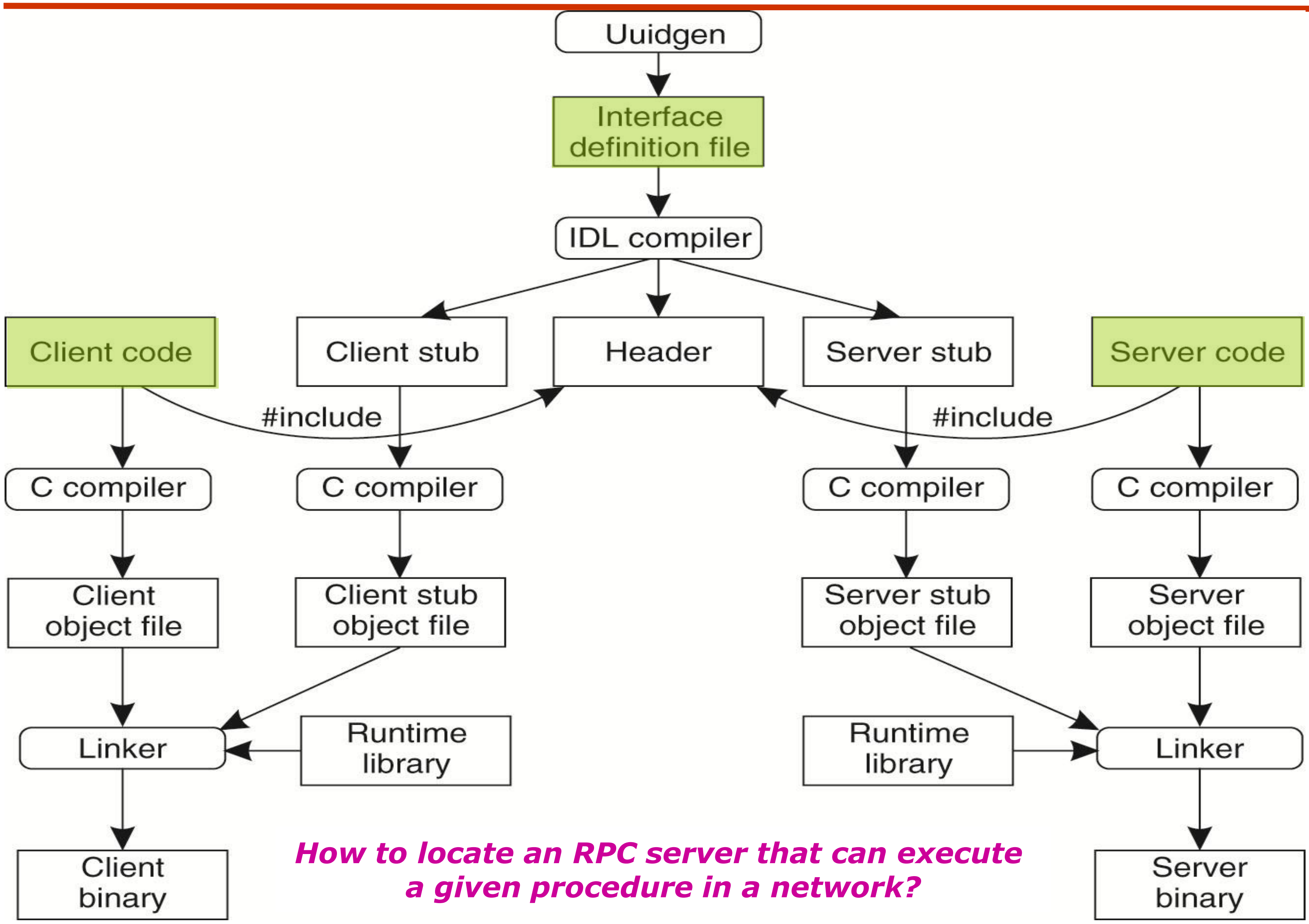
- Both client and server must follow the same protocol when passing complex data structures
  - Agree on format and representation
- Stubs take care of (un)packaging arguments and sending messages
- Programmers just define interfaces using Interface Definition Language (IDL)
- To simplify programmer's task, Stub compiler generates stub automatically from specs in IDL



(b)

We will see an example later!

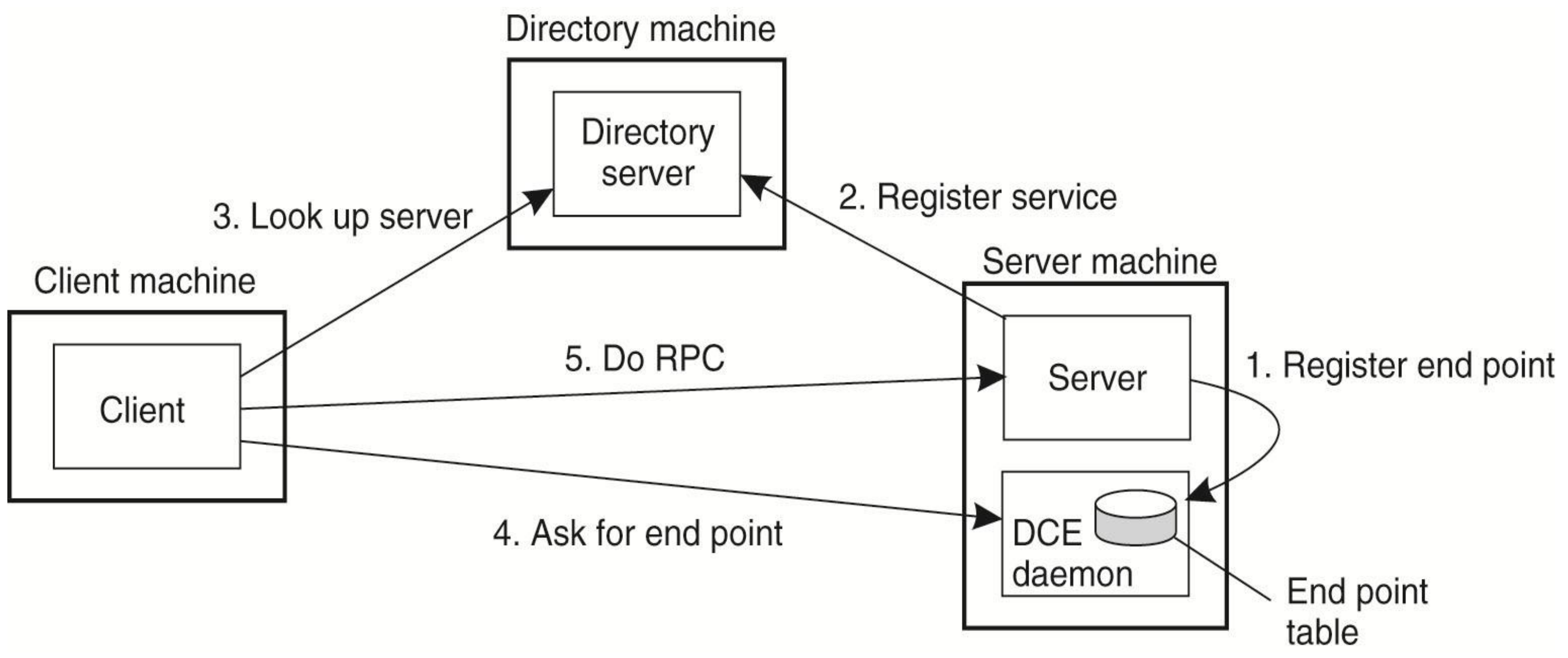
RPC in Practice



How to locate an RPC server that can execute a given procedure in a network?

Binding a Client to a Server

- Client must locate the server's machine and locate server on that machine (how?)
- Registration of a server makes it possible for a client to locate the server and bind to it



Binding

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

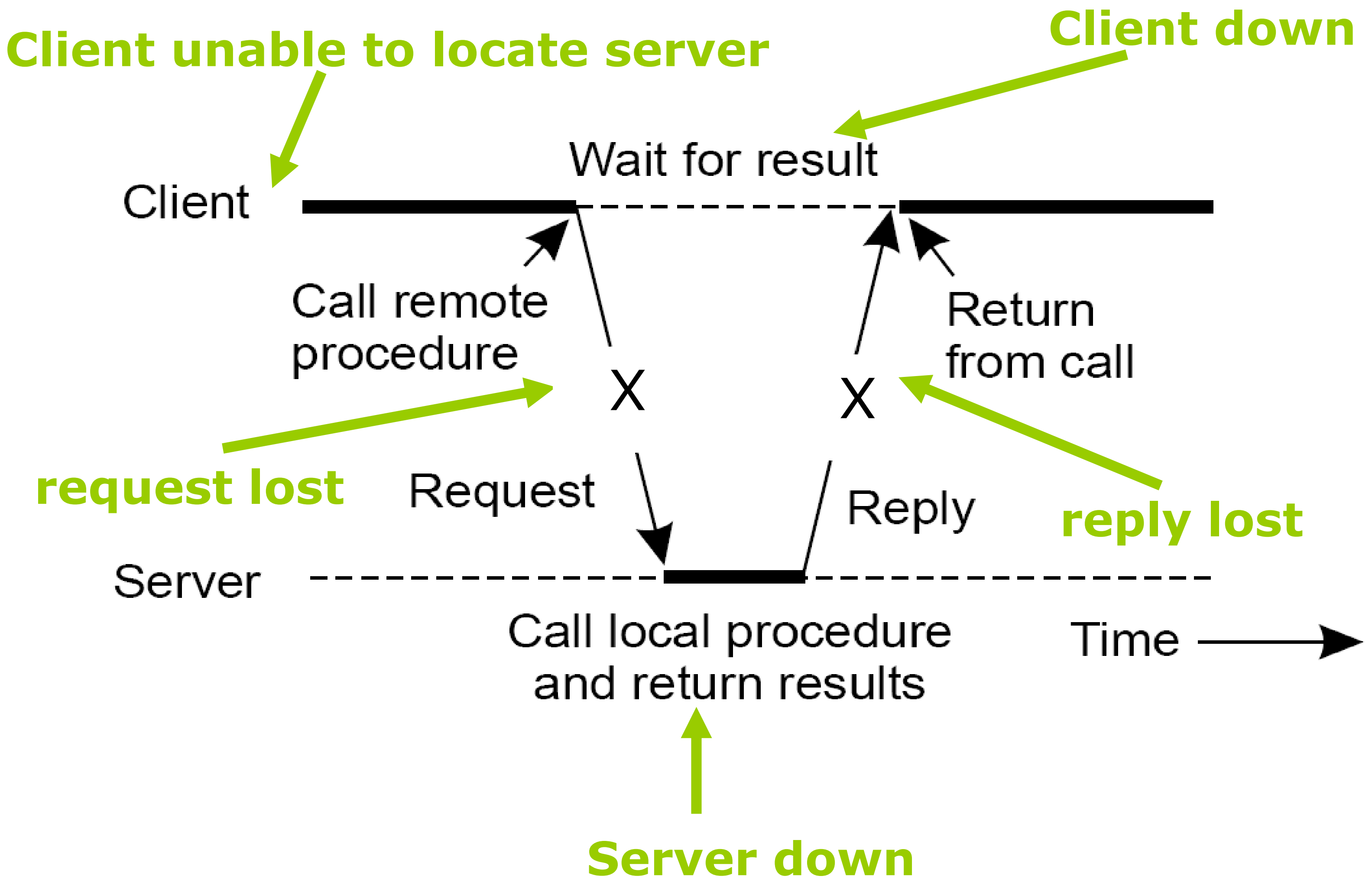
What may go wrong?

What to do when there is a failure?

(ch 8)

RPC OPERATION IN CASE OF FAILURES

What may go wrong in Request-Reply protocol (RR)?



Failure Semantics

- *Client unable to locate server*:
  - return error
- *Lost request messages*:
  - simple timeout mechanisms, resend?
- *Lost replies*:
  - timeout mechanisms, resend?

- What are the problems with resending?

Resend?

- Client: time out → retry request
  - If **timeout** and no reply received, it **resends** request
  - If client does not receive reply after N attempts it *assumes* that server has failed and gives up
- Server: duplicated requests
  - **Re-execute & resend** results;
    - Can the requested operation be re-executed on server?
    - Add **sequence numbers** to detect duplicate requests
  - **Store requests & results**
    - How long should you store the results?

Should Servers Re-Do Operations?

- **Idempotent** operations*:*
  - can be performed **repeatedly** with the **same** effect*.*

- For idempotent operations: no state needs to be maintained on the server
- Are the following operations idempotent?
  - HTTP GET … ← **yes**
  - UNIX file operations: read, write etc. ← **NO**

Server failure
did failure occur before or after operation?

- **Exactly once**: ideal case, same as local, but difficult to achieve
- **At least once**: will guarantee that RPC has been carried out at least once, but possibly more
  - Acceptable only if the server's operations are **idempotent**. That is f(x) = f(f(x)).
- **At most once**: Will guarantee that RPC has been carried out at most once, but possibly none at all
  - Implemented by the server's filtering of duplicate requests
- <u>No guarantees:</u> When a server crashes, the client gets no help and no promises about what happened
    - The partial execution may lead to erroneous results.
    - In this case, we want the effect that the RP has not been executed at all.

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

Invocation Semantics

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	***Maybe***
Yes	No	Re-execute procedure	***At-least-once***
Yes	Yes	Retransmit reply	***At-most-once***

3.11 Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the “at most once” or “exactly once” semantic. Describe possible uses for a mechanism that has neither of these guarantees.

Answer: If an RPC mechanism cannot support either the “at most once” or “at least once” semantics, then the RPC server cannot guarantee that a remote procedure will not be invoked multiple occurrences. Consider if a remote procedure were withdrawing money from a bank account on a system that did not support these semantics. It is possible that a single invocation of the remote procedure might lead to multiple withdrawals on the server. For a system to support either of these semantics generally requires the server maintain some form of client state such as the timestamp described in the text. If a system were unable to support either of these semantics, then such a system could only safely provide remote procedures that do not alter data or provide time-sensitive results. Using our bank account as an example, we certainly require “at most once” or “at least once” semantics for performing a withdrawal (or deposit!). However, an inquiry into an account balance or other account information such as name, address, etc. does not require these semantics.

EXAMPLES: SUN RPC

Sun RPC Interface

- Type/data definitions (like C).
- Component is described as a PROGRAM
  - Procedures have a result type, a parameter list and a number,
  - Procedure can be called remotely
- Used by client or server directly:
  - Locating servers: static vs. dynamic binding
  - Choosing a transport protocol.
  - Authentication and security.
  - Invoking RPCs dynamically.
- Used by stubs for:
  - Generating unique message IDs.
  - Sending messages.
  - Maintaining message history

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

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

```
/* pi.x: Remote pi calculation protocol */
program PIPROG {
   version CALCU_PIVERS {
     double CALCU_PI() = 1;
   } = 1;
} = 0x39876543;
```

```
/** pi_server.c **/
#include <rpc/rpc.h> /* always needed */
#include "pi.h"

double *calcu_pi_1_svc(void *argp,
                                   struct svc_req *rqstp)
{
                static double  pi;

                double sum = 0;
                int i;
                int sign;

                for (i=1; i<10000000; i++ ){
                   sign = (i+1) % 2;
                   if ( sign == 0 )
                      sign = 1;
                   else
                      sign = -1;

                    sum += 1.0 / (2*(double)i -1) *
                                 (double)sign;
                }

                pi = 4 * sum;

                return (&pi);
}
```

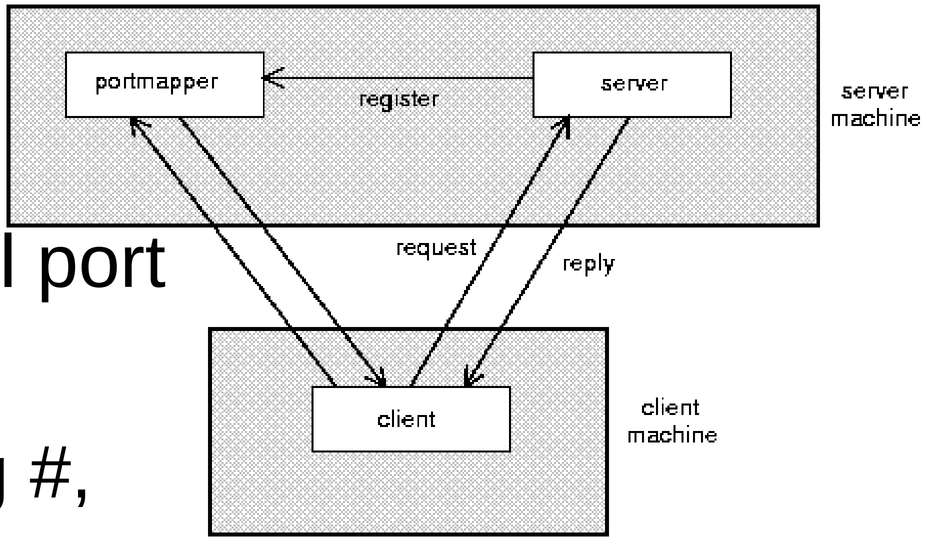
```
/** pi_client.c */
#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "pi.h"

main(int argc, char *argv[])
{
   CLIENT *clnt;
   double  *result_1;
   char *host;
   char *calcu_pi_1_arg;

   /* must have two arguments */
   if (argc < 2) {
      printf("usage:  %s server_host\n", argv[0]); exit(1);
   }
   host = argv[1]; /* server host name */
   clnt = clnt_create(host, PIPROG,CALCU_PIVERS, "tcp");
   if (clnt == (CLIENT *) NULL) {
      clnt_pcreateerror(host);
      exit(1);
   }
   /* call remote procedure */
   result_1 = calcu_pi_1((void *)&calcu_pi_1_arg, clnt);
   if (result_1 == (double *) NULL) {
      clnt_perror(clnt, "call failed");
   }
   /* print the pi value */
   printf("PI is %f\n" , *result_1);
   clnt_destroy(clnt);
   exit(0);
}
```

```
rpcgen  pi.x
cc -g    -o pi_client  pi_clnt.c pi_client.c -lnsl
cc -g    -o pi_server pi_svc.c pi_server.c -lnsl
```

- pi_server &
- pi_client localhost

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



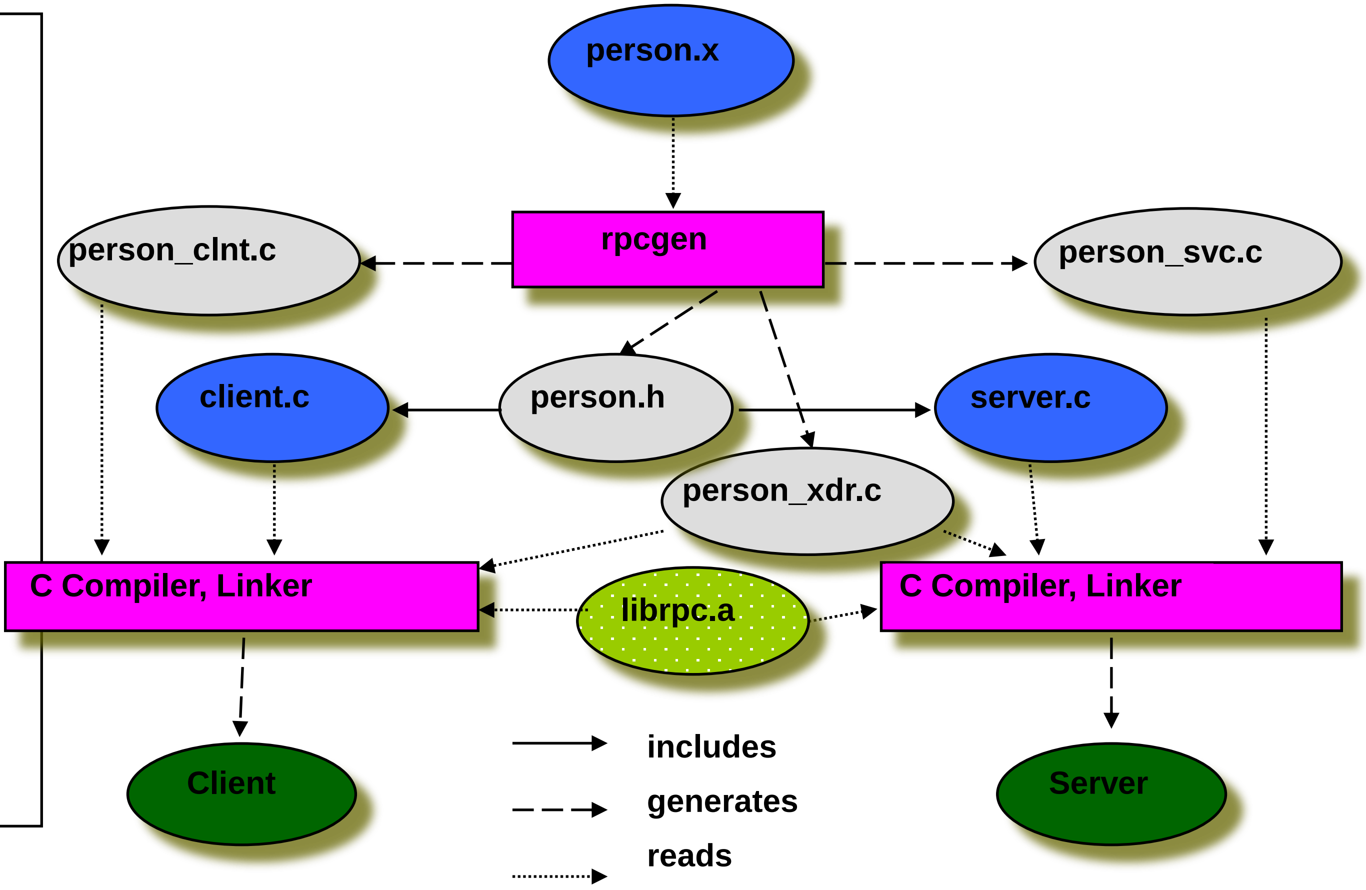
`/sbin/rpcbind`

The `rpcbind` utility is a server that converts RPC program numbers into universal addresses.

It must be running on the host to be able to make RPC calls on a server on that machine.

Example 2

```
/* person.x */
const NL=64;
enum sex_type {
 FEMALE = 1, MALE = 2};
struct Person {
  string first_name<NL>;
  string last_name<NL>;
  sex_type sex;
  string city<NL>;
};
program PERSONPROG {
 version PERSONVERS {
  void PRINT(Person)=0;
  int STORE(Person)=1;
  Person LOAD(int)=2;
 } = 1;
} = 1234567;
```



```
rpcgen  person.x
cc -g   -o person_client  person_clnt.c person_client.c person_xdr.c -lnsl
cc -g   -o person_server person_svc.c person_server.c person_xdr.c -lnsl

person_server &
person_client localhost
```

Example 2 (cont'd)

```
/* person_server.c */
#include <rpc/rpc.h> /* always needed */
#include "person.h"

Person pers = {"ABC name", "ABC lastname",
               MALE, "ABC city"};
int a=5;
void *print_1_svc(Person *argp,
           struct svc_req *rqstp)
{
  static char *result;
  printf("PRINT:%s %s\n%s\n\n",
      argp->first_name,
      argp->last_name,
      argp->city);
  return((void *) &result);
}
int *store_1_svc(Person *argp,
           struct svc_req *rqstp)
{
  printf("STORE: %s %s\n%s\n\n",
      argp->first_name,
      argp->last_name,
      argp->city);
  return &a;
}
Person *load_1_svc(int *num,
           struct svc_req *rqstp)
{
  printf("LOAD: Server got %d \n", *num);
 return &pers;
}
```

```
/* person_client.c */
#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "person.h"

main(int argc, char *argv[])
{
  CLIENT *clnt;
  char *host;
  Person pers = {"Person Name", "Lastname",
                  MALE, "San Antonio"};
  Person *p2;
  int *i, a=8;
  if (argc < 2) {/* must have two arguments */
    printf("usage:  %s server_host\n", argv[0]); exit(1);
  }
  host = argv[1]; /* server host name */
  clnt = clnt_create(host, PERSONPROG,
              PERSONVERS, "udp");
   if (clnt == (CLIENT *) NULL) {exit(1);}
   if (print_1(&pers, clnt)==NULL)
          clnt_perror(clnt, "call failed");
   if ((p2=load_1(&a, clnt))==NULL)
          clnt_perror(clnt, "call failed");
  printf("%s\n", p2->last_name);
  if (print_1(p2, clnt)==NULL)
          clnt_perror(clnt, "call failed");
  if ((i=store_1(&pers, clnt))==NULL)
          clnt_perror(clnt, "call failed");

 clnt_destroy(clnt);
}
```

OPTIONAL

EXTRAS

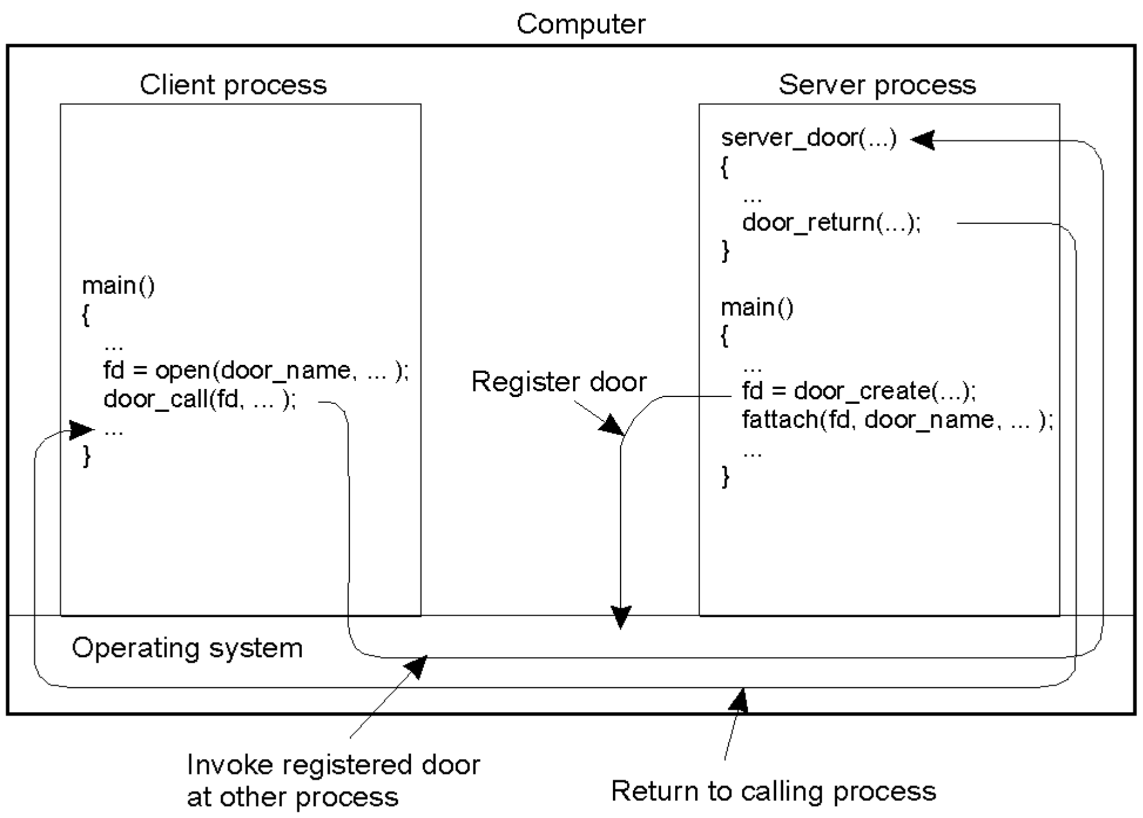
Lightweight RPCs

- Many RPCs occur between client and server on same machine
  - Need to optimize RPCs for this special case => use a lightweight RPC mechanism (LRPC)
- Server *S* exports interface to remote procedures
- Client *C* on same machine imports interface
- OS kernel creates data structures including an argument stack shared between *S* and *C*

Lightweight RPCs

- RPC execution
  - Push arguments onto stack
  - Trap to kernel
  - Kernel changes mem map of client to server address space
  - Client thread executes procedure (OS upcall)
  - Thread traps to kernel upon completion
  - Kernel changes the address space back and returns control to client
- Called “doors” in Solaris

Doors



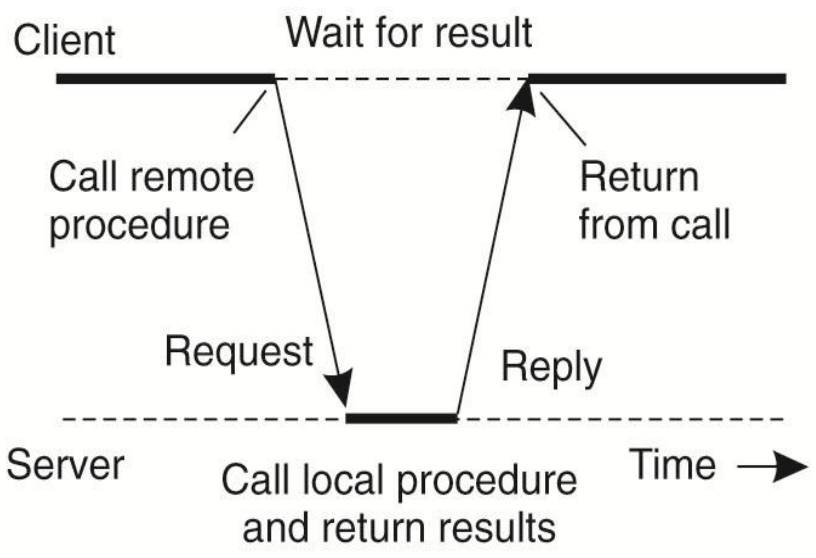
- Which RPC to use? - run-time bit allows stub to choose between LRPC and RPC

Other RPC Models

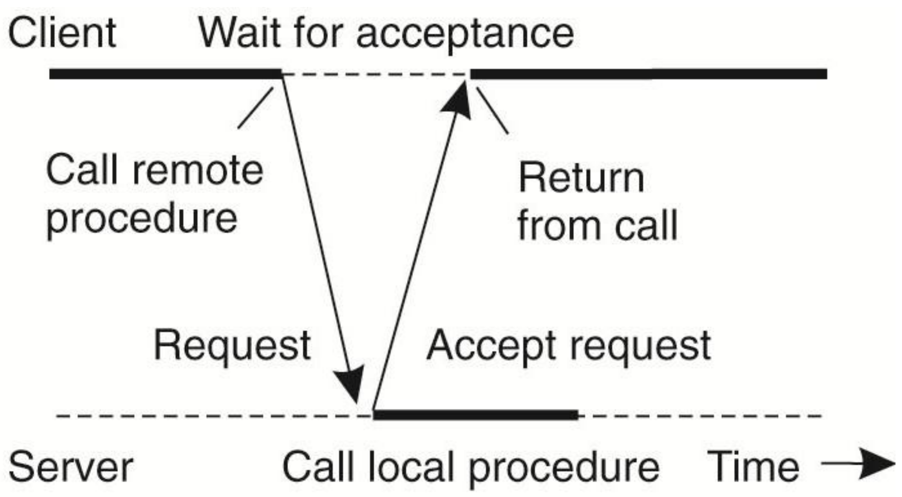
- Asynchronous RPC
  - Request-reply behavior often not needed
  - Server can reply as soon as request is received and execute procedure later
- Deferred-synchronous RPC
  - Use two asynchronous RPCs
  - Client needs a reply but can't wait for it; server sends reply via another asynchronous RPC
- One-way RPC
  - Client does not even wait for an ACK from the server
  - Limitation: reliability not guaranteed (Client does not know if procedure was executed by the server).

Traditional RPC vs. Asynchronous RPC

- Try to get rid of the strict request-reply behavior, and let the client continue without waiting for an answer from the server.



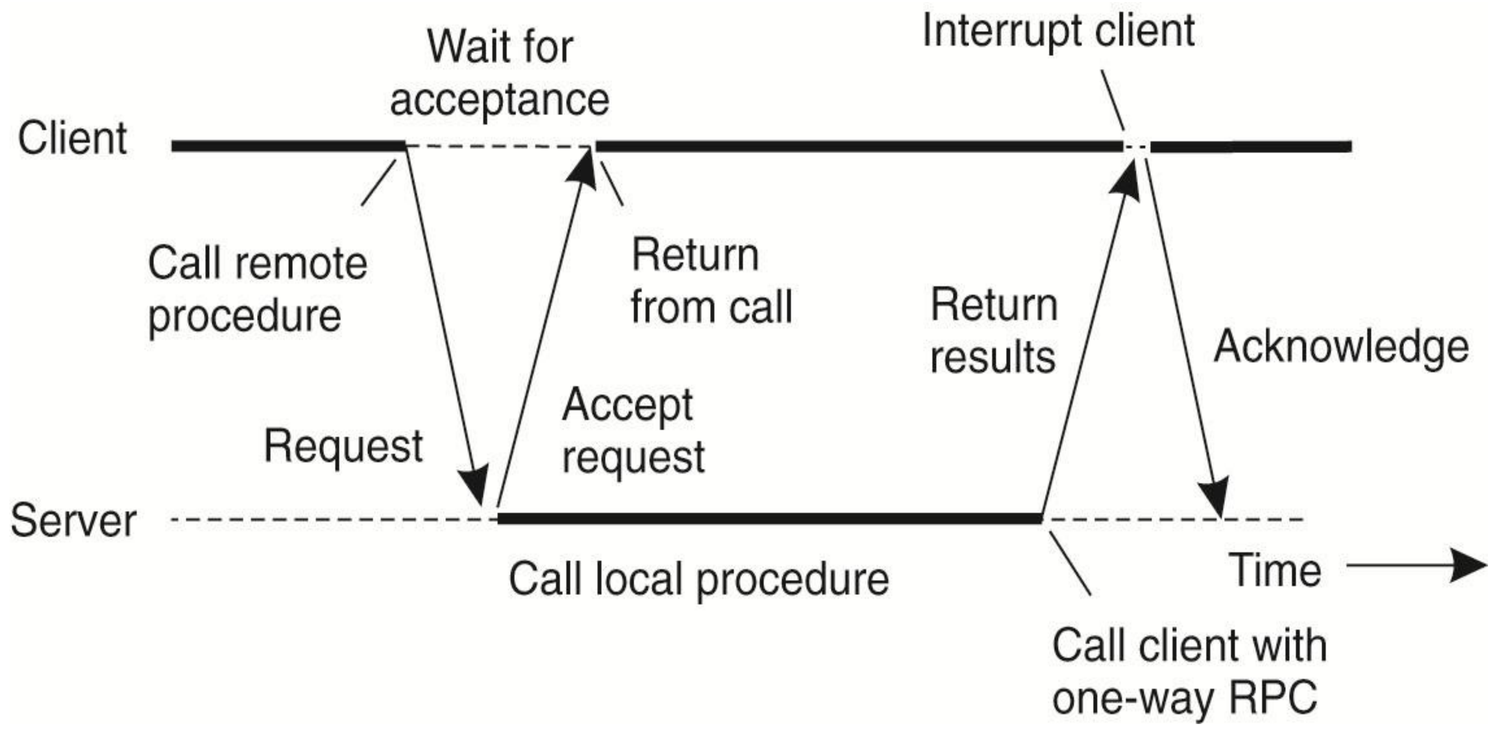
Traditional synchronous RPC



asynchronous RPC server stub immediately sends a reply to client.

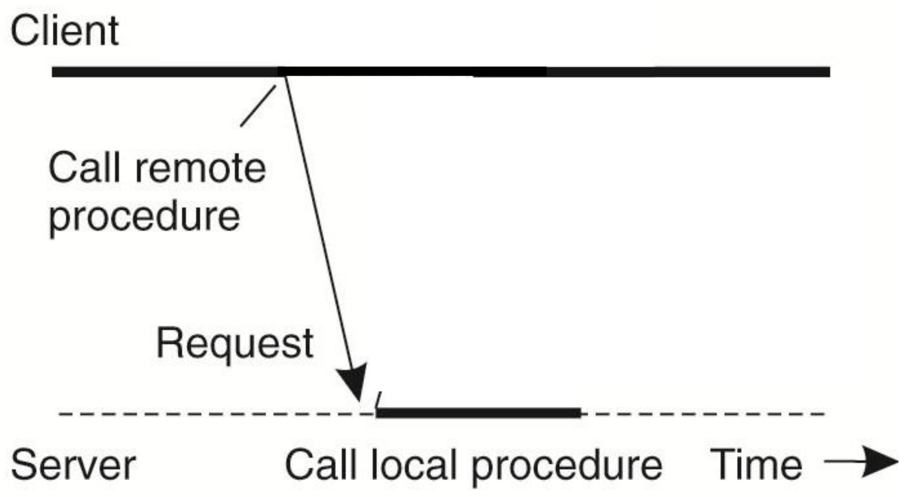
Deferred Synchronous RPC

- Client can also do a (non)blocking poll at the server to see whether results are available.
- Through two asynchronous RPCs



One-Way RPC

- Client does not know if the request is accepted or not (ch 8 fault tolerance)



RPC Limitations

- Parameters passed by values only and pointer values are not allowed.

- Speed: remote procedure calling (and return) time (i.e., overheads) can be significantly (1 - 3 orders of magnitude) slower than that for local procedure.
  - This may affect real-time design and the programmer should be aware of its impact.

RPC Limitations

- Failure: RPC is more vulnerable to failure (since it involves communication system, another machine and another process).
  - The programmer should be aware of the call semantics, i.e. programs that make use of RPC must have the capability of handling errors that cannot occur in local procedure calls.

Design Issues

- Exception handling
  - Necessary because of possibility of network and nodes failures;
  - RPC uses return value to indicate errors;
- Transparency
  - Syntactic $\rightarrow$ achievable, exactly the same syntax as a local procedure call;
  - Semantic $\rightarrow$ impossible because of RPC limitation: failure (similar but not exactly the same);