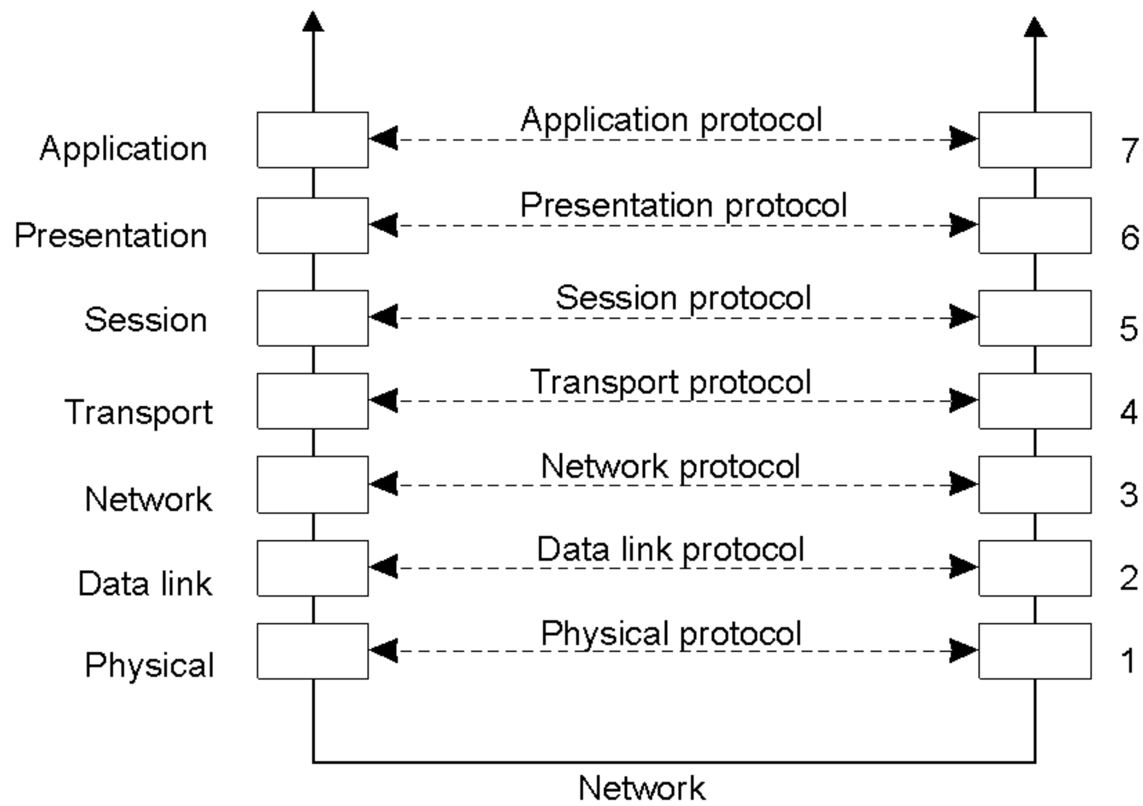
Communication

Layered Protocols

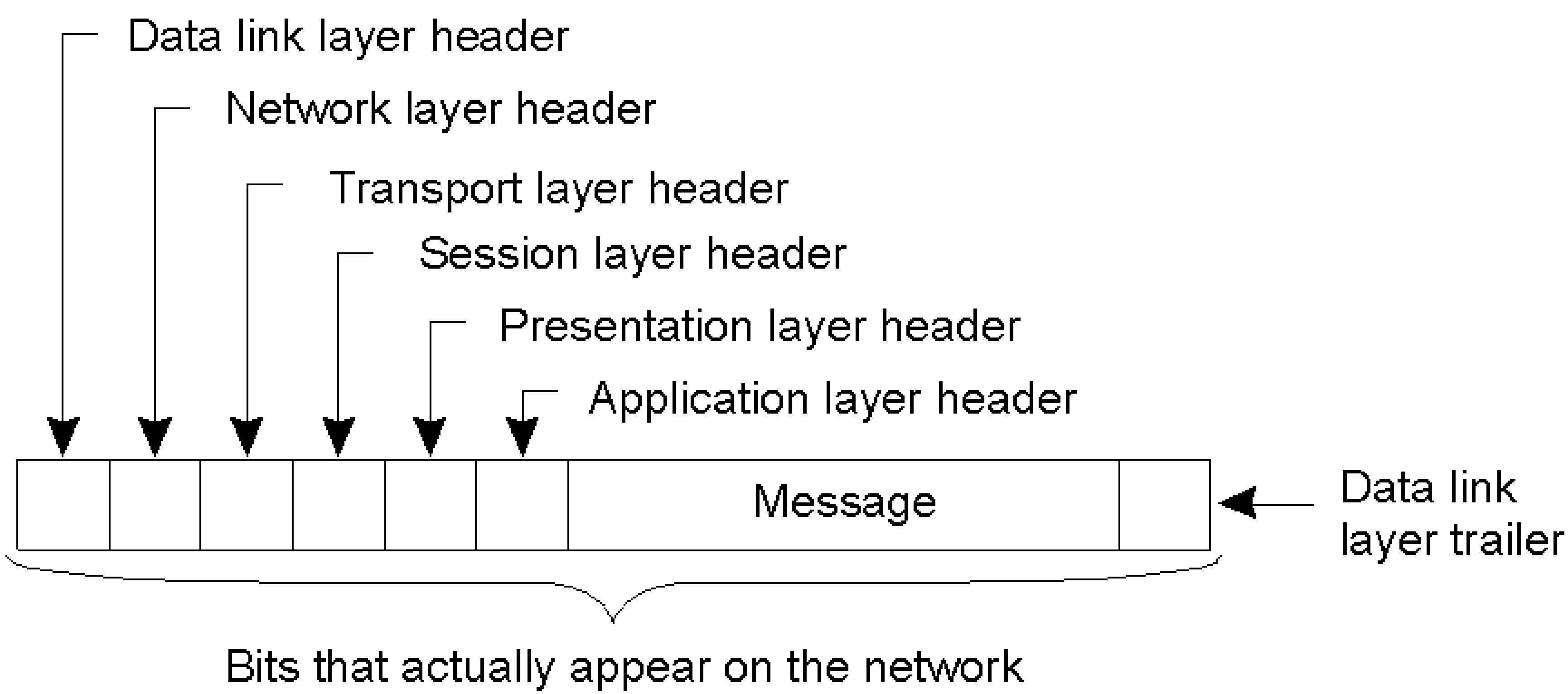
- Layers, interfaces, and protocols in the OSI model.



Layered Protocols

- International Standards Organization (ISO)
  - Developed a reference model that clearly identifies the various levels involved, gives them standard names, and points out which level should do which job.
- Open systems Interconnection Reference Model
- Protocols: rules determining how an open system can communicate with another open system.
- Connection-oriented protocols: Before exchanging data the sender and receiver first explicitly establish a connection, and possibly negotiate the protocol they will use
- Connectionless protocols: No setup in advance is needed.
  - Dropping a letter in a mailbox

Layered Protocols



Layered Protocols

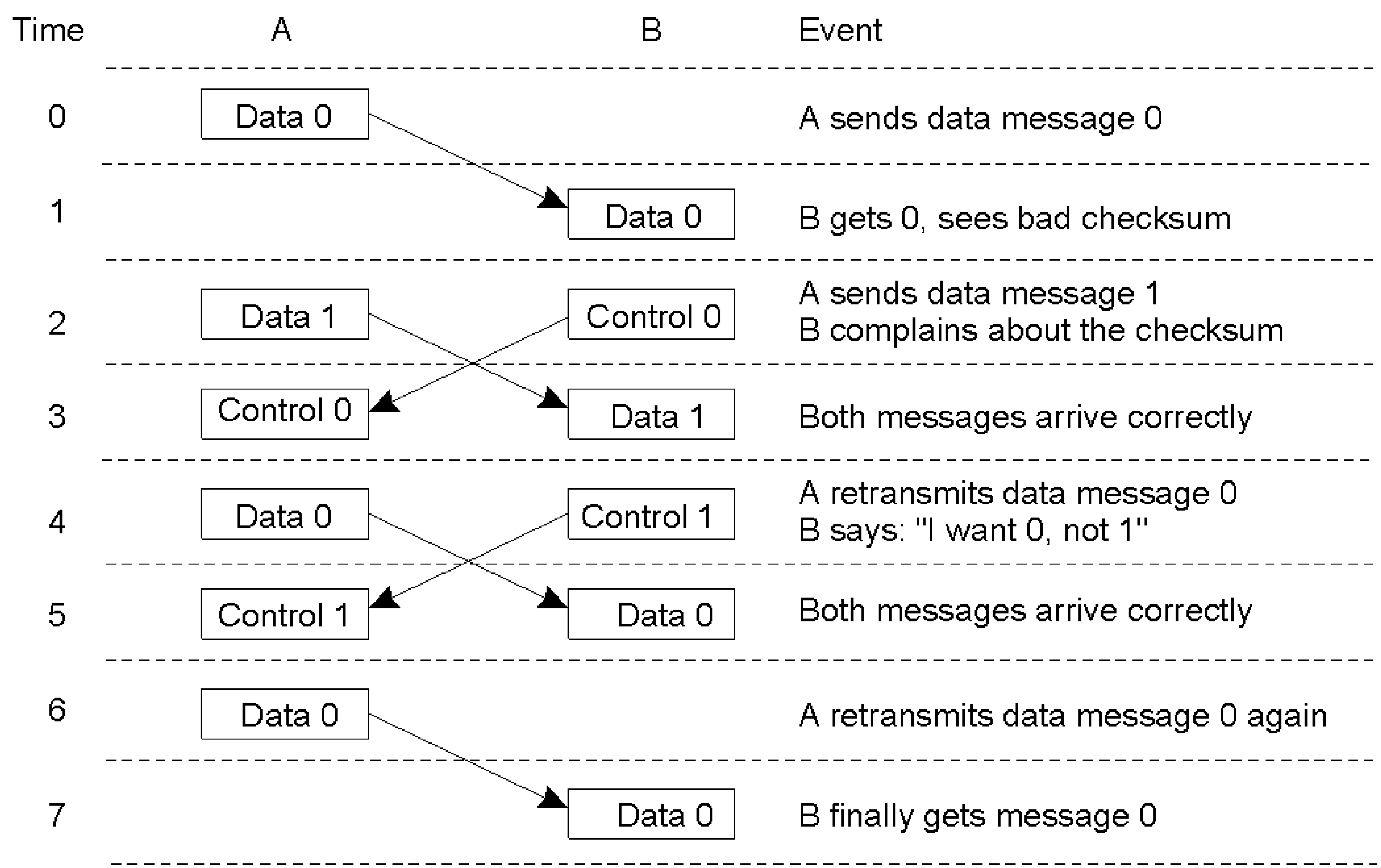
Physical Layer

- Undertakes the actual transmission of the message.
  - How many volts to use for 0 and 1?
  - How many bits per second can be sent?
  - Can transmission take place in both directions simultaneously?
  - Size and shape of the network connector
  - Number of pins and meanings of each

Data Link Layer

- Provide mechanisms to detect and correct errors during the bit transmission
  - Bits are grouped into units called frames.
  - A special bit pattern is placed at the beginning and at the end of each frame to mark it.
  - A checksum is computed by adding up all the bytes in the frame in a certain way.
  - Frames are assigned sequence numbers

Data Link Layer



- Discussion between a receiver and a sender in the data link layer.

Layered Protocols

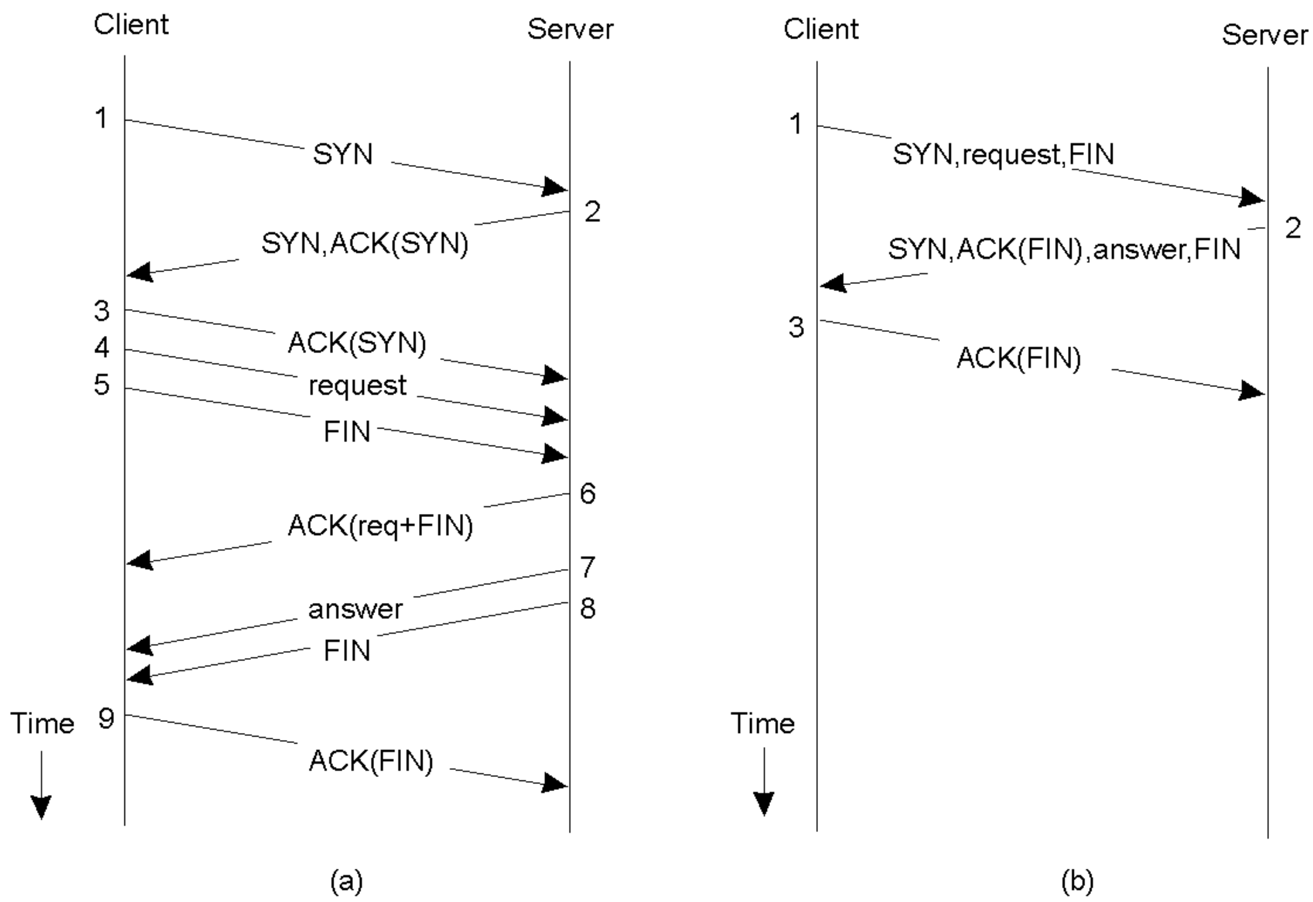
Network Layer

- The choosing of the best path from the sender to the receiver is called routing.
- Routing is the major task of the network layer.
- Example of Connectionless protocol: IP
  - No connection setup.
  - Each IP packet is routed to its destination independently of all others.
  - No internal path is selected and remembered.
- Example of Connection-oriented protocol – ATM Virtual Channels
  - A unidirectional connection from the source to the destination is established.
  - A collection of virtual channels between two hosts comprise a virtual path.

Transport Protocols

- Delivers messages without loss.
- Each message is broken into pieces called packets
- A sequence number is assigned to each packet
- Transport Layer Header
  - Which packets have been sent
  - Which have been received
  - How many more the receiver has room to accept
  - Which should be retransmitted

TCP & Client-Server TCP (TCP for Transactions, T/TCP)



Higher Level Protocols

Session Layer

- Provides dialog control, to keep track of which party is currently talking
- Provides synchronization facilities
  - Insert checkpoints into long transfers, so that in the event of a crash, it is necessary to go back only to the last checkpoint

Presentation Layer

- Is concerned with the meaning of the bits
  - It is possible to define records such a name, address, amount of money, and other valuable info that might be contained in a message, and have the sender notify the receiver that a message contains a particular record in a certain format.
  - Makes communication easier between machines with different internal representations

Application Protocols

- Contains a collection of standard network applications, like e-mail, file transfer, terminal emulation, etc.
- FTP
  - FTP protocol versus ftp program
- HTTP (HyperText Transfer Protocol)
  - Remotely manage and handle the transfer of Web pages
    - Web browsers and web servers
    - JAVA RMI

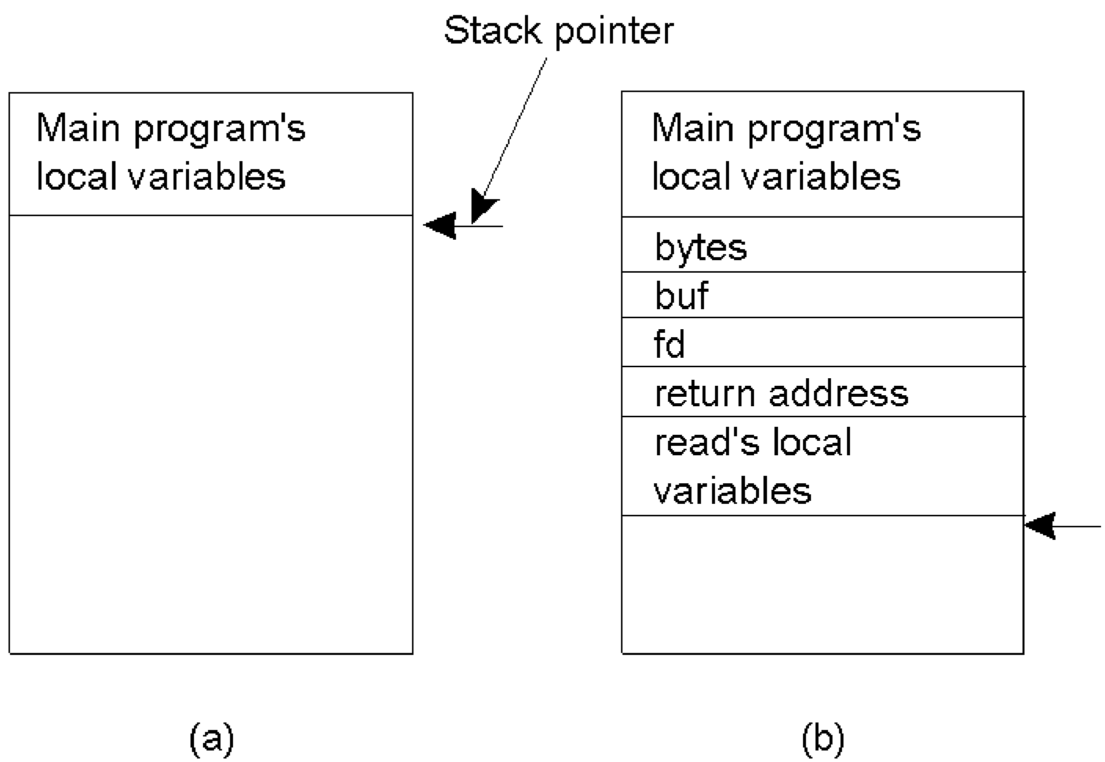
Remote Procedure Call - RPC

- Explicit message exchange using send and receive does not conceal communication.

Main Idea behind RPCs

- Allow programs to call procedures located on other machines
- Subtle problems exist, since the calling and called procedures:
  - run on different machines
  - execute on different address spaces
  - failures may occur

Conventional Procedure Call



C call

count = read(fd,buf,nbytes)

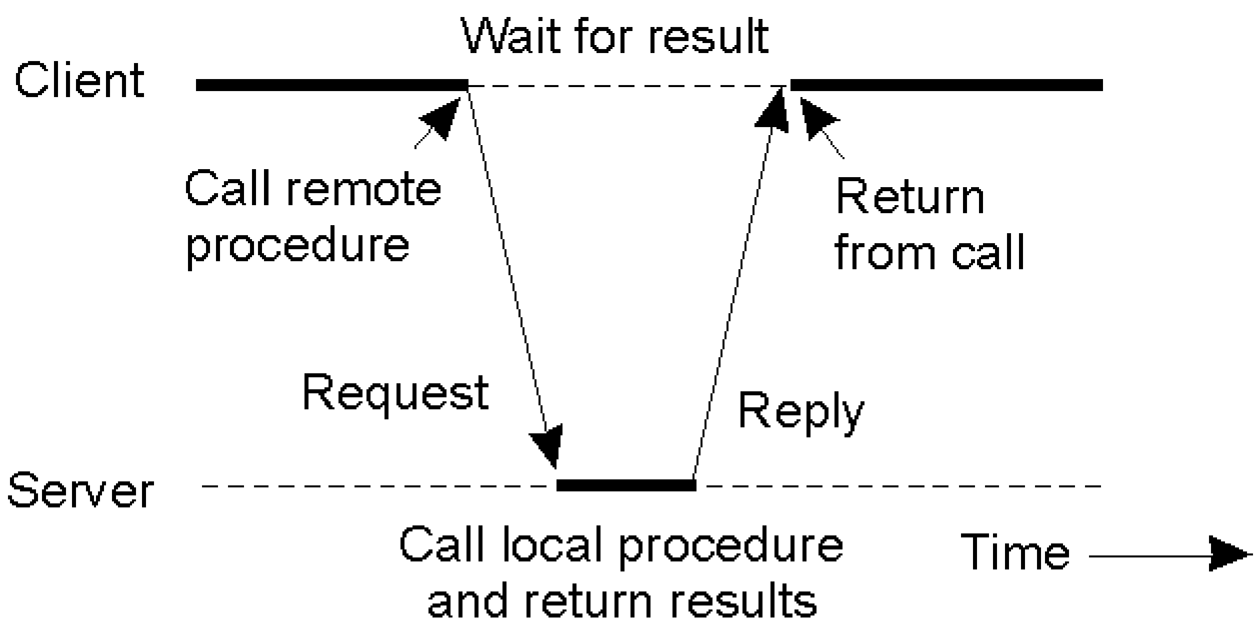
- **Call by value**
- **Call by reference**
- **Call by copy/restore**

a) Parameter passing in a local procedure call: the stack before the call to read
b) The stack while the called procedure is active

RPC versus Calling a System Call - Read data from a file

Local file	Remote file
▪ The programmer calls read ▪ The read routine is extracted from the library by the linker ▪ It is a short procedure that calls a system call ▪ The read procedure is a kind of interface between the user code and the local OS.	▪ The programmer calls read ▪ the read routine is extracted from the library by the linker (it is now called client stub) ▪ **it packs the parameters into a message and requests that message to be sent to the server** ▪ When the message arrives at the server, the OS passes it up to a server stub. ▪ The server stub unpacks the parameters and calls the server procedure in the usual way.

Client and Server Stubs



- Principle of RPC between a client and server program.

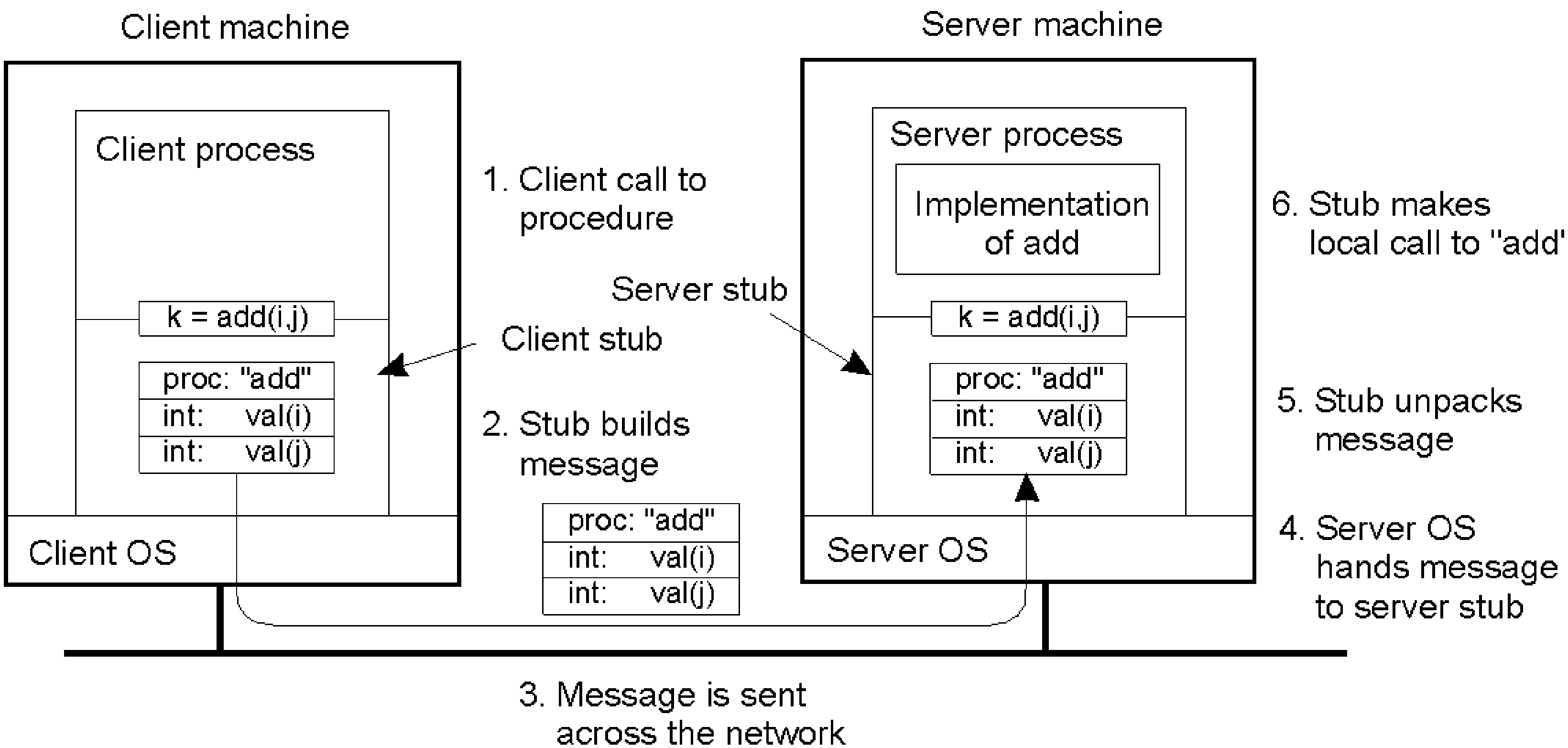
Client and Server Stubs

- Client's side:
  - read is called placing appropriate arguments in the stack (in the conventional way)
  - the read library routine does a call to the OS (as happens in the conventional call)
  - Unlike the original one, it does not ask the OS to perform a system call but to send a message to the server
- Server's side:
  - the server procedure is called in the usual way; the parameters and the return address are all on the stack and nothing seems unusual.
- The transparency achieved is the main beauty of the scheme!

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

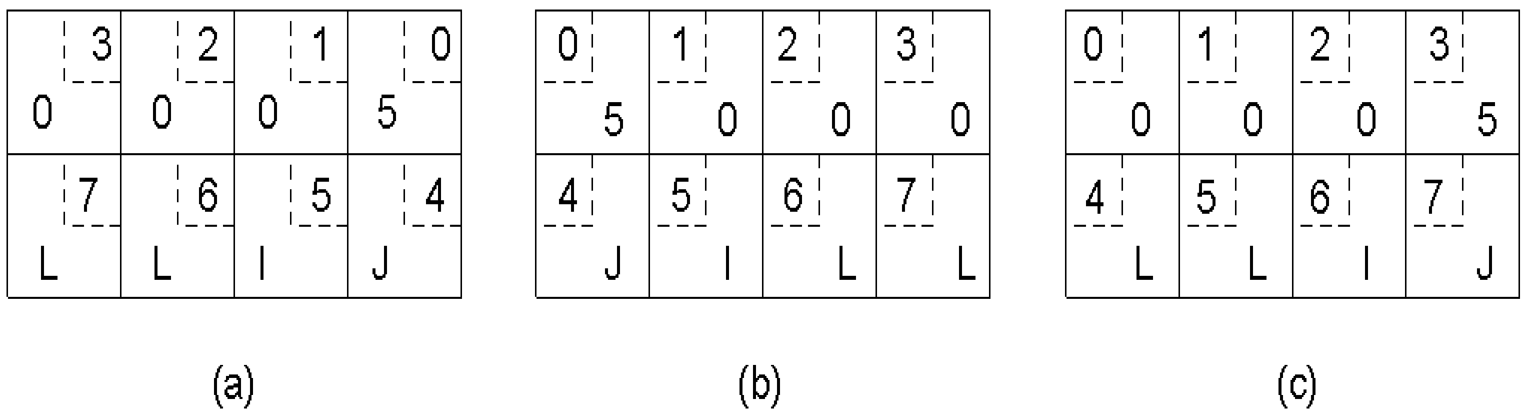
Passing Value Parameters (1)



Packing parameters into a message is called parameter marshalling.

Passing Value Parameters (2)

- Each machine has its own representation for numbers, characters, and other data items. IBM mainframes use the EBCDIC character code, whereas IBM PCs use ASCII
- Intel Pentium machines number their bytes from right to left (little endian), Sun SPARC number them the other way (big endian).



a) Original message on the Pentium
b) The message after receipt on the SPARC
c) The message after being inverted. The little numbers in boxes indicate the address of each byte

Passing Reference Parameters

- How are pointers or in general references passed?
  - With the greatest of difficulty, if at all ☹

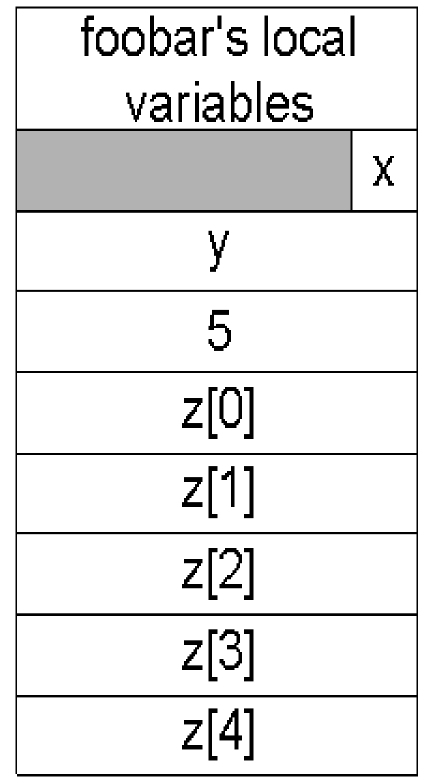
- Possible Solutions
  1. Forbid pointers
  2. Copy the object pointed to by the pointer into the message and send it to the server.
     - The server copies it at some place in its memory space, and calls the routine passing a pointer to it.
     - When the routine ends, the server copies back the object's value into one parameter of the message and sends the message to the client.
     - If the stubs know whether the object is an input or output parameter to the server, one of the two copies can be avoided.
     - The above approach does not work with complex objects (i.e, dynamic arbitrary data structures).

Parameter Specification and Stub Generation

a) A procedure
b) The corresponding message:
   - A character is placed in the rightmost byte of a word
   - A float is transmitted as a whole word
   - An array as a group of words equal to the array length, preceded by a word giving the length.

```
foobar( char x; float y; int z[5] )
{
   ....
}
```

(a)



(b)

Parameter Specification and Stub Generation

The caller and the callee agree on:

- The format of the messages
- The representation of simple data structures
  - integers are represented in two's complement, characters in 16-bit Unicode, floats in IEEE standard #754 format, etc.
- The actual exchange of the message
  - TCP/IP?
  - UDP?
- Stubs for the same RPC protocol but different procedures generally differ only in their interface to the applications.
- An interface consists of a collection of procedures that can be called by a client, and which are implemented by a server.
- Interfaces are often specified by means of an **Interface Definition Language (IDL),**
  - then compiled into a client stub and a server stub, along with the appropriate compile-time or run-time interfaces.

Asynchronous RPC

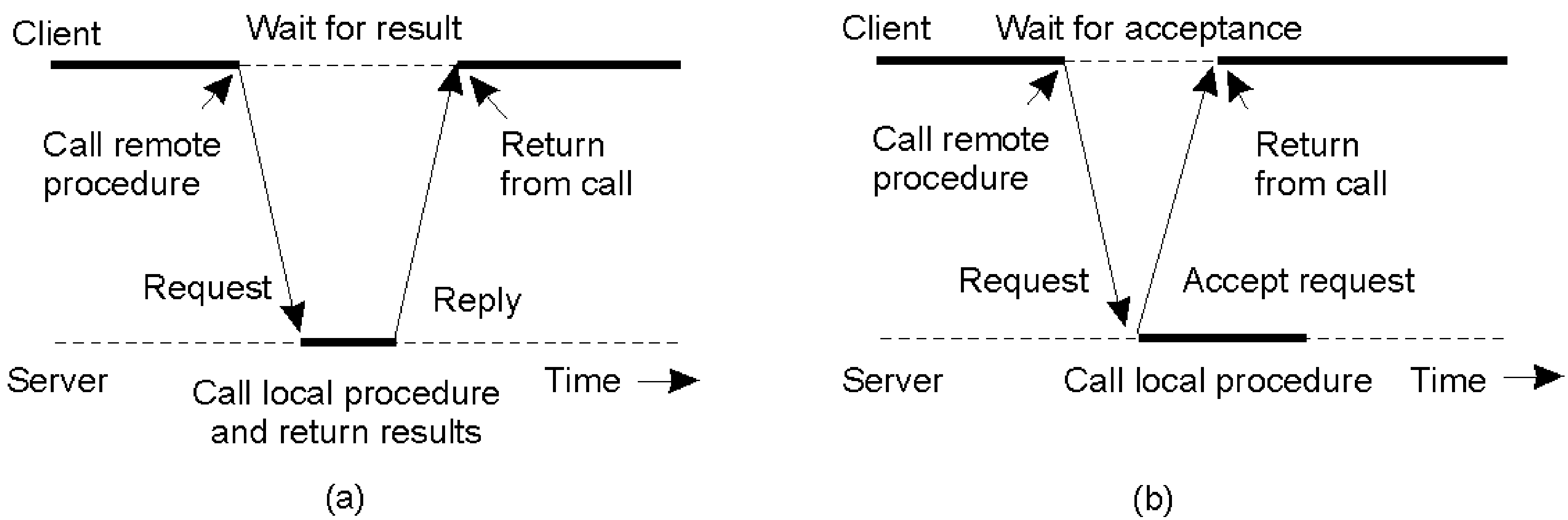
Examples where blocking is not necessary

- Transferring money from one account to another
- Adding entries into a database
- Starting remote services
- Batch processing

Asynchronous RPCs

- A client immediately continues after issuing the RPC request.

Asynchronous RPC (1)

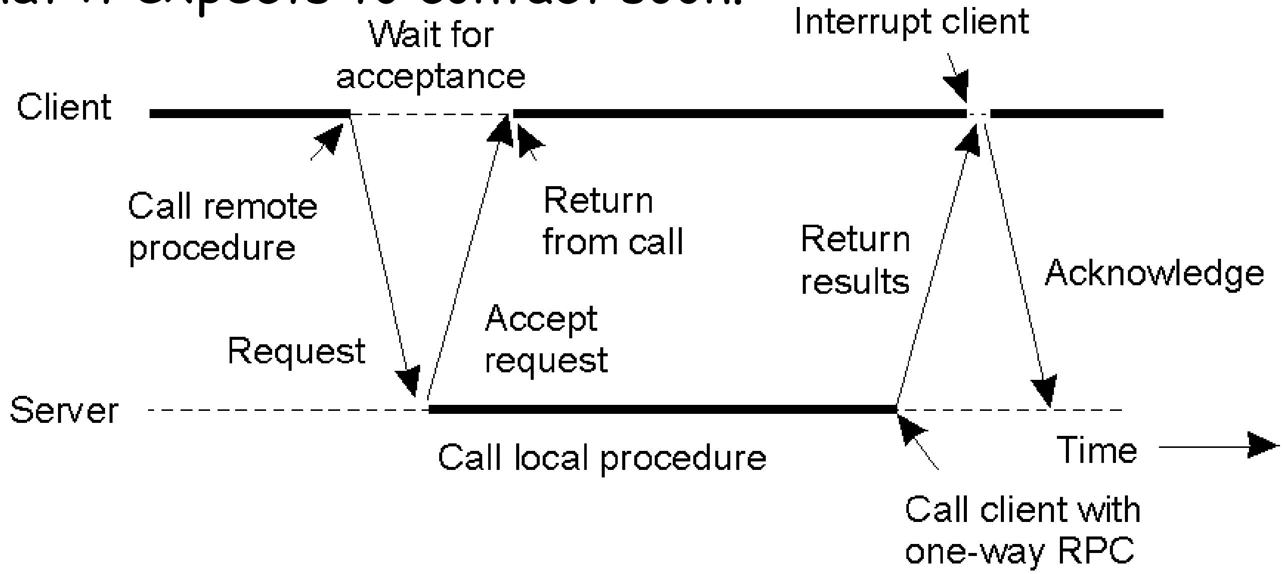


a) The interconnection between client and server in a traditional RPC
b) The interaction using asynchronous RPC

Asynchronous RPC (2)

Deferred Synchronous RPCs

• A client may want to pre-fetch the network addresses of a set of hosts that it expects to contact soon.

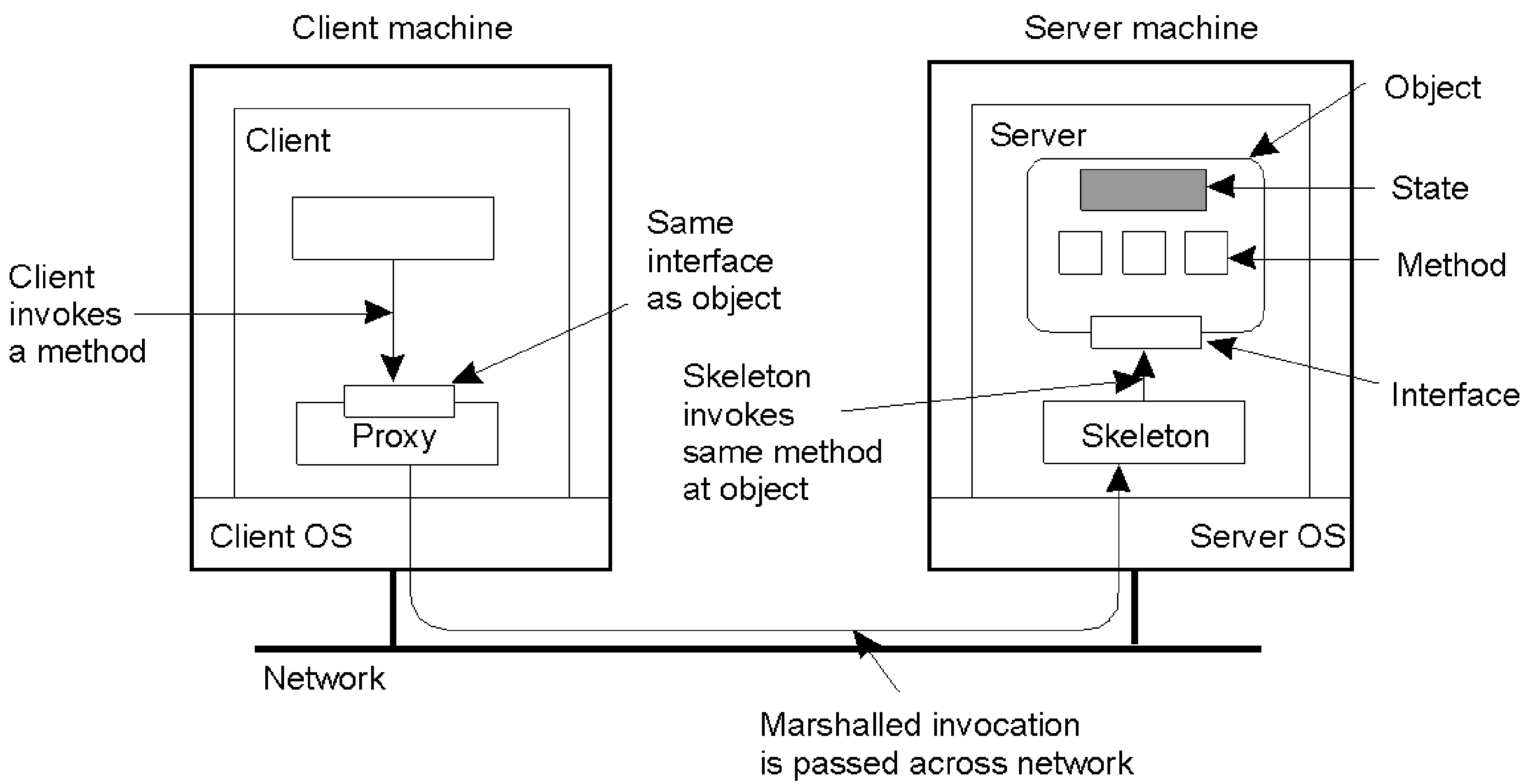


- A client and server interacting through two asynchronous RPCs
- Combining two asynchronous RPCs is referred as a deferred synchronous RPC.
- Completely Asynchronous RPCs

Distributed Objects

- The key feature of an object is that it encapsulates data, called state, and the operations on those data, called methods.
- Methods are made available through an interface.
- When a client binds to a distributed object, an implementation of the object's interface, called proxy, is loaded into the client's address space.
  - It marshals method invocations into messages and unmarshals reply messages to return the result to the client
  - Proxy = client stub
  - Skeleton = server stub
- Simple remote object: its state is not distributed, but its interface might be.
- Persistent Objects: continues to exist even if it is currently not contained in the address space of a server process
- Transient Objects: exists only as long as the server that manages it is active.

Distributed Objects



Binding a Client to an Object

- Object references are supported by RMI systems
- When a process holds an object reference, it must first bind to the reference's object before invoking any of its methods.
  - Binding results in a proxy being installed in the process's address space.
- **Implicit Binding**: binding is done automatically
  - The client is offered a mechanism that allows it to directly invoke methods using only a reference to the object.
- **Explicit Binding**: more transparent to the client
  - The client first calls a special function to bind to the object and then invokes any method.

Implementation of Object References

- An object reference should provide the following information:
  - The network address of the machine where the state of the object resides
  - An endpoint (port) identifying the server that manages the object
  - An id identifying which object in this server.
- If a server crashes and recovers, a new endpoint might be assigned to it
  - All object references become invalid
  - ✓ Have a local daemon per machine listening to a well-known endpoint, and keep track of the server-to-endpoint assignments in an endpoint table.
  - ✓ Replace the endpoint with an id in the object reference

Implementation of Object References

- Encoding the network address of the server within the object's reference is also not a good idea
  - Location servers
- Assumption of using the same protocol stack can be dropped
  - Add more information in the object reference
    - identification of the protocol
  - proxy implementation handle

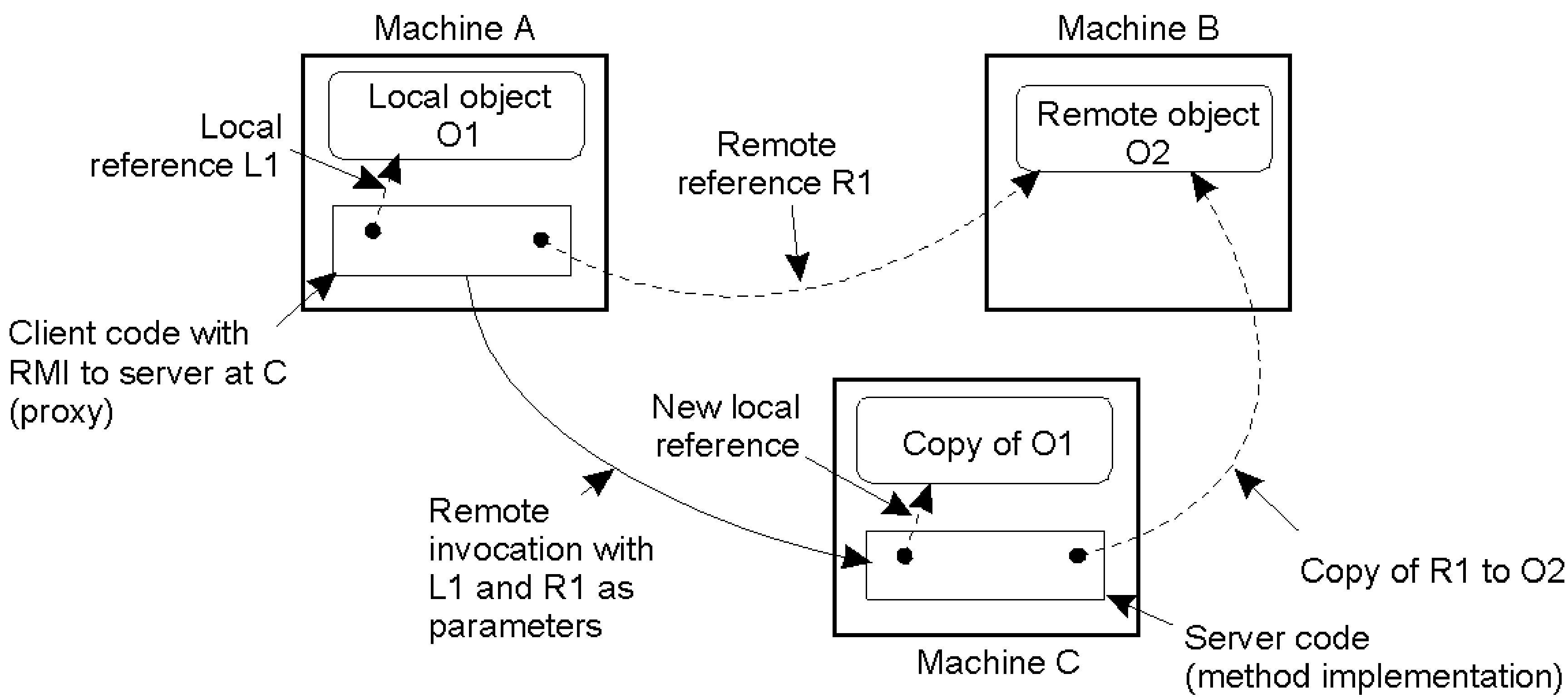
Static versus Dynamic Remote Memory Invocations

- Static Invocation
  - Use predefined inteface definitions
    - make use of an object-based language (e.g., JAVA) that will handle stub generation automatically.
    - **Example**: fobject.append(int);
- Dynamic Invocation
  - compose a method invocation at run time
  - **Example**: invoke(fobject,id(append), int);
- Usefulness of Dynamic Invocations
  - browser that is used to examine sets of objects and supports remote object invocations
  - dynamic invocations in a batch processing service where invocations can be handled along with a time determining when the invocation should be done

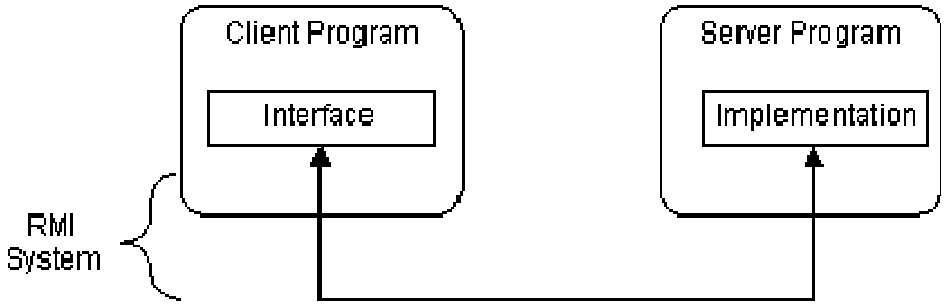
Parameter Passing

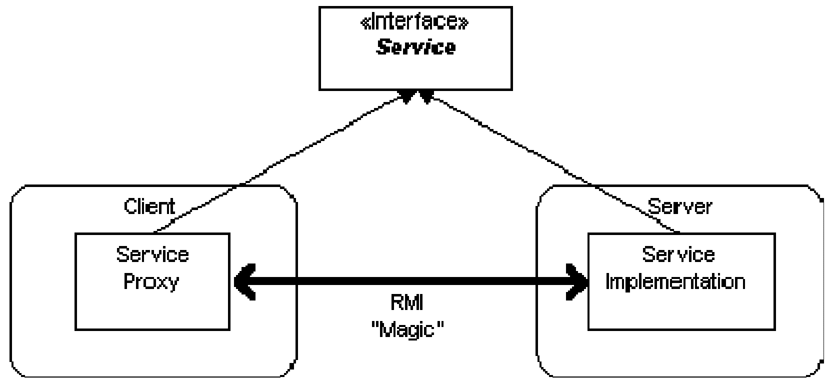
- Object references can be used as parameters to method invocations
- References are passed by value
  - They are copied from one machine to another
- References to local objects are treated differently for efficiency
  - The referenced local object may be copied as a whole and passed along with the invocation

Parameter Passing



JAVA RMI Architecture



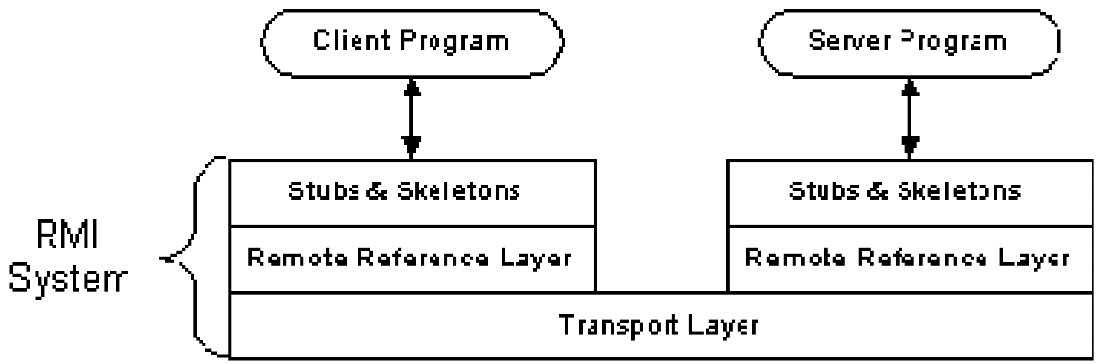


The definition of behavior and the implementation of that behavior are separate concepts:

- The definition of a remote service is coded using a Java interface.
- The implementation of the remote service is coded in a class.
- RMI supports two classes that implement the same interface.
  - The first class is the implementation of the behavior, and it runs on the server.
  - The second class acts as a proxy for the remote service and it runs on the client.

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

RMI Architecture Layers



- The Stubs and Skeleton layer intercepts method calls made by the client to the interface reference variable and redirects these calls to a remote RMI service. Implements the stub and skeletons needed.
- The Remote Reference layer connects clients to remote service objects that are running and exported on a server. Defines and supports the invocation semantics of the RMI connection.
- The transport layer is based on TCP/IP connections between machines in a network. A client is connected to a a remote service implementation by establishing a unicast point-to-point connection.

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Naming Remote Objects

- How does a client find an RMI remote service?
  - A naming or directory service is run on a well-known host and port number.
- How does a client obtains a reference to a service object?
  - RMI includes a simple service called the RMI registry, which runs on each machine that hosts remote service objects and accepts queries for services by default on port 1099.
  - Each object should be exported and registered.
  - The registry provides a remote reference to a service object (a URL is used to describe the service object).

Using RMI - Example

- Parts composing a working RMI
  - **Interface definitions for the remote services**
  - **Implementations of the remote services**
  - **Stub and Skeleton files**
  - **A server to host the remote services**
  - **An RMI Naming service that allows clients to find the remote services**
  - **A class file provider (an HTTP or FTP server)**
  - **A client program that needs the remote services**

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Using RMI - Example

- **Steps to build a system:**
  1. **Write and compile Java code for interfaces**
  2. **Write and compile Java code for implementation classes**
  3. **Generate Stub and Skeleton class files from the implementation classes**
  4. **Write Java code for a remote service host program**
  5. **Develop Java code for RMI client program**
  6. **Install and run RMI system**

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Example -Interface

```
public interface Calculator
                     extends java.rmi.Remote {
   public long add(long a, long b)
        throws java.rmi.RemoteException;
   public long sub(long a, long b)
        throws java.rmi.RemoteException;
   public long mul(long a, long b)
        throws java.rmi.RemoteException;
   public long div(long a, long b)
        throws java.rmi.RemoteException;
}
```

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Example - Implementation

```
public class CalculatorImpl
        extends java.rmi.server.UnicastRemoteObject implements Calculator {
    // Implementations must have an explicit constructor  in order to declare the
    // RemoteException exception

    public CalculatorImpl()  throws java.rmi.RemoteException
    {
    super();
    }
    public long add(long a, long b) throws java.rmi.RemoteException {
        return a + b;
    }
    public long sub(long a, long b) throws java.rmi.RemoteException {
        return a - b;
    }
    public long mul(long a, long b) throws java.rmi.RemoteException {
        return a * b;
    }
    public long div(long a, long b) throws java.rmi.RemoteException {
        return a / b;
    }
}
```

When the constructor calls super(), it activates code that performs the RMI linking and remote object initialization.

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Example – Stubs and Skeletons & Host Server

- The stub and skeleton files are created using the RMI compiler, rmic (rmic CalculatorImpl)
- This generates the Calculator_Stub.class and Calculator_Skel.class

Host Server

```
public class CalculatorServer {
    public CalculatorServer() {
        try {
            Calculator c = new CalculatorImpl();
            Naming.rebind("rmi://localhost:1099/CalculatorService", c);
        } catch (Exception e) {
            System.out.println("Trouble: " + e);
        }
    }
    public static void main(String args[]) {
        new CalculatorServer();
    }
}
```

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Example - Client

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.net.MalformedURLException;
import java.rmi.NotBoundException;
public class CalculatorClient {
    public static void main(String[] args) {
            try {
                Calculator c = (Calculator) Naming.lookup( "rmi://localhost/CalculatorService");
                System.out.println( c.sub(4, 3) );
                System.out.println( c.add(4, 5) );
                System.out.println( c.mul(3, 6) );
                System.out.println( c.div(9, 3) );
            } catch (MalformedURLException murle) {
                 System.out.println();      System.out.println( "MalformedURLException");
                 System.out.println(murle);
            } catch (RemoteException re) {
                 System.out.println();      System.out.println( "RemoteException");
                 System.out.println(re);
            } catch (NotBoundException nbe) {
                 System.out.println();      System.out.println( "NotBoundException");
                 System.out.println(nbe);
            } catch ( java.lang.ArithmeticException ae) {
                 System.out.println();      System.out.println( "java.lang.ArithmeticException");
                System.out.println(ae);
            }
        }
}
```

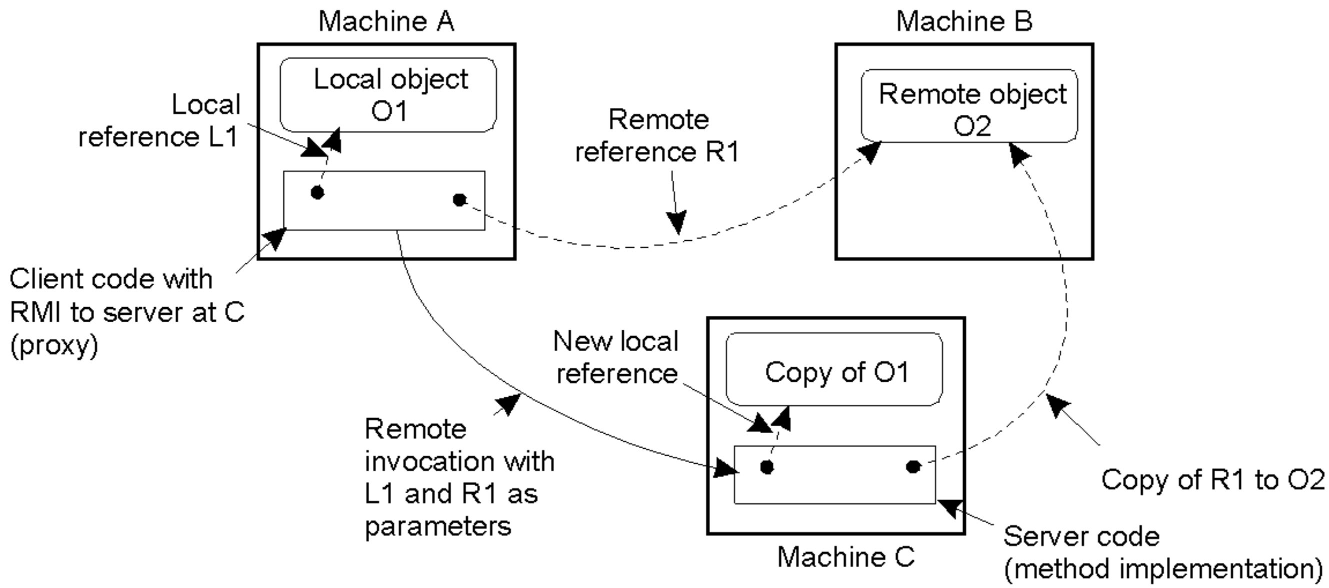
Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Parameter-Passing in RMI

- When a local object is passed to a remote method, the object itself is passed by value, not the reference to the object.
- When a remote method returns an object, a copy of the whole object is returned to the calling program.
- A Java object can be simple and self-contained, or it could refer to other Java objects in complex graph-like structure.
- Because different JVMs do not share heap memory, RMI must send the referenced object and all objects it references.
- RMI uses a technology called serialization to transform an object into a linear format that can then be sent over the network wire.
- Object serialization essentially flattens an object and any objects it references.

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Parameter-Passing in RMI



- A client program can obtain a reference to a remote object:
  - through the RMI Registry program, or
  - as the return value from a method call

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

Distributing and Installing RMI Software

- To run an RMI application, the supporting class files must be placed in locations that can be found by the server and the clients.
- For the server, the following classes must be available to its class loader:
  - Remote service interface definitions
  - Remote service implementations
  - Skeletons for the implementation classes
  - Stubs for the implementation classes
  - All other server classes
- For the client, the following classes must be available to its class loader:
  - Remote service interface definitions
  - Stubs for the remote service implementation classes
  - Server classes for objects used by the client (such as return values)
  - All other client classes

Material taken from http://java.sun.com/developer/onlineTraining/rmi/RMI.html

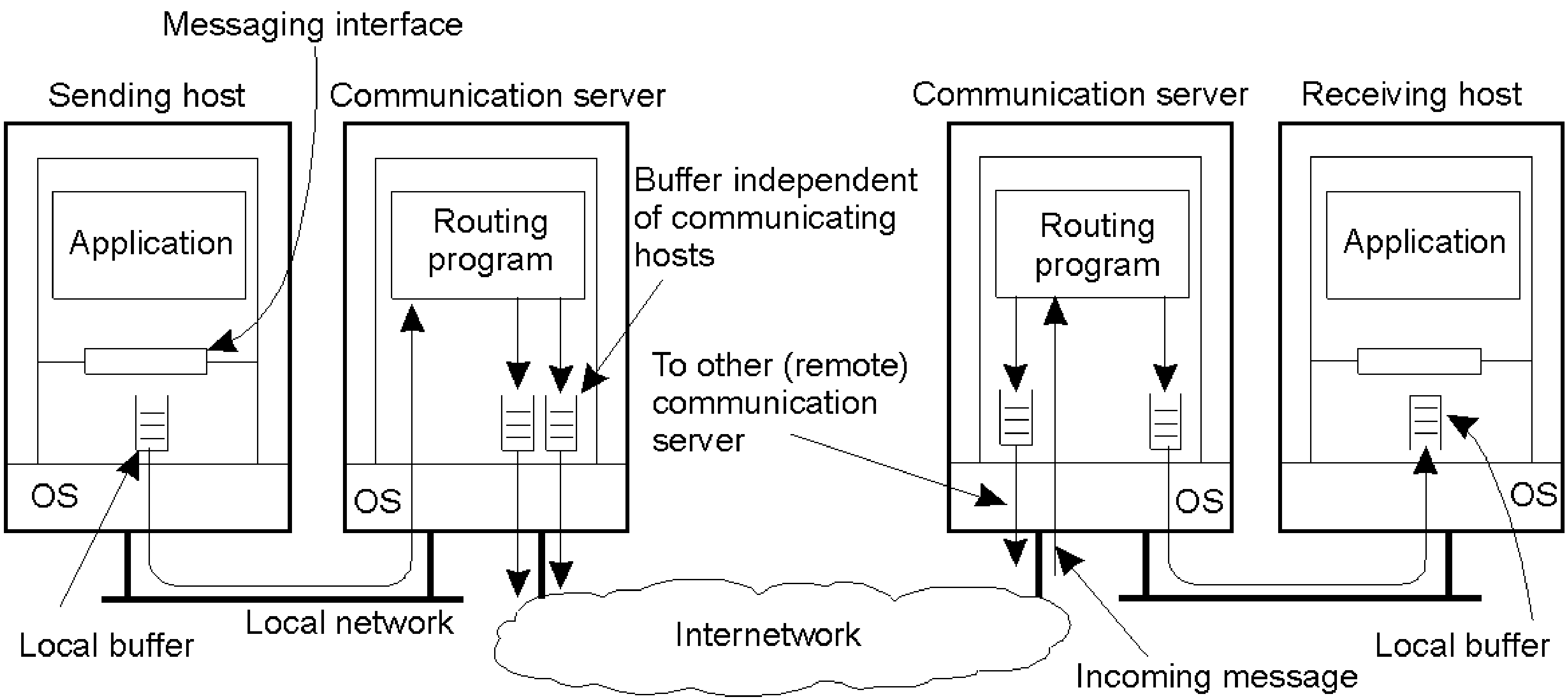
Distributing and Installing RMI Software

- The RMI supports loading of classes from FTP and HTTP servers (class RMIClassLoader).
  - classes can be deployed in one, or only a few places,
  - all nodes in a RMI system will be able to get the proper class files to operate.
- If the remote JVM needs to load a class file for an object, it looks for the embedded URL and contacts the server at that location for the file.
- When the property java.rmi.server.useCodebaseOnly is set to true, then the JVM will load classes from either a location specified by the CLASSPATH environment variable or the URL specified in this property.

Message-Oriented Communication

- Applications are executed on hosts
- The hosts are connected through a network of communication servers
- Each host is connected to some communication server
- **Example: Electronic Mail System**
  - Each host runs an application by which a user can compose, send, receive and read messages.
  - Each host is connected to a mail server
  - Each message is first stored in one of the output buffers of the local mail server.
  - The server removes messages from its buffers and sends them to their destination.
  - The target mail server stores the message in an input buffer for the designated receiver (in the receiver's mailbox).
  - The interface at the receiving host offers a service to the receiver's user agent by which the latter can regularly check for incoming mail.

Persistence and Synchronicity in Communication



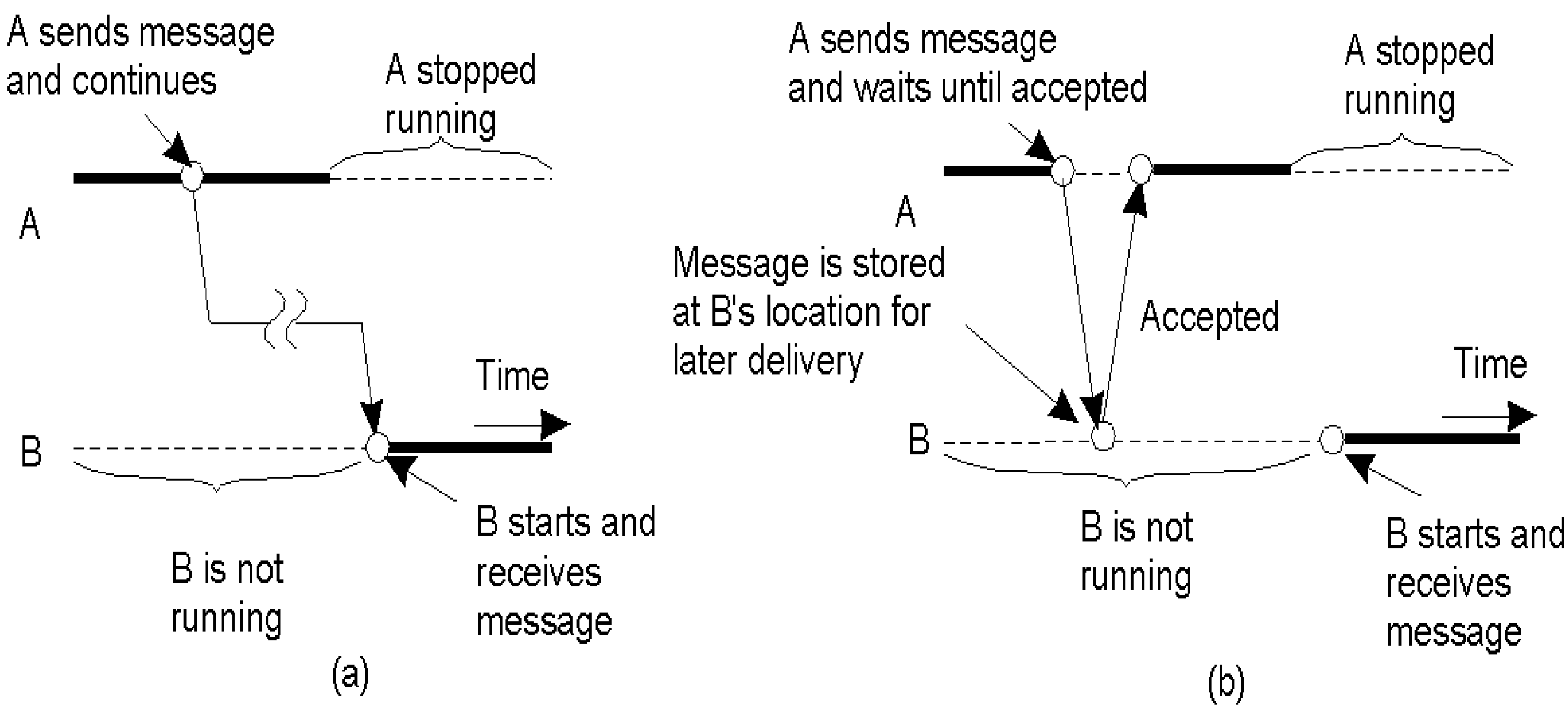
Persistence and Synchronicity in Communication

- Persistent communication
  - a message that has been submitted for transmission is stored by the communication system as long as it takes to deliver it to the receiver.
- Transient communication
  - a message is stored by the communication system only as long as the sending and receiving application are executing.
    - If a communication server cannot deliver a message to the next communication server or the receiver, the message will be discarded
    - It works like a traditional store-and-forward router
- Asynchronous Communication
  - A sender continues its execution immediately after it has submitted its message for transmission
- Synchronous Communication
  - The sender is blocked until its message is stored in a local buffer at the receiving host, or actually delivered to the receiver.

Persistence and Synchronicity in Communication

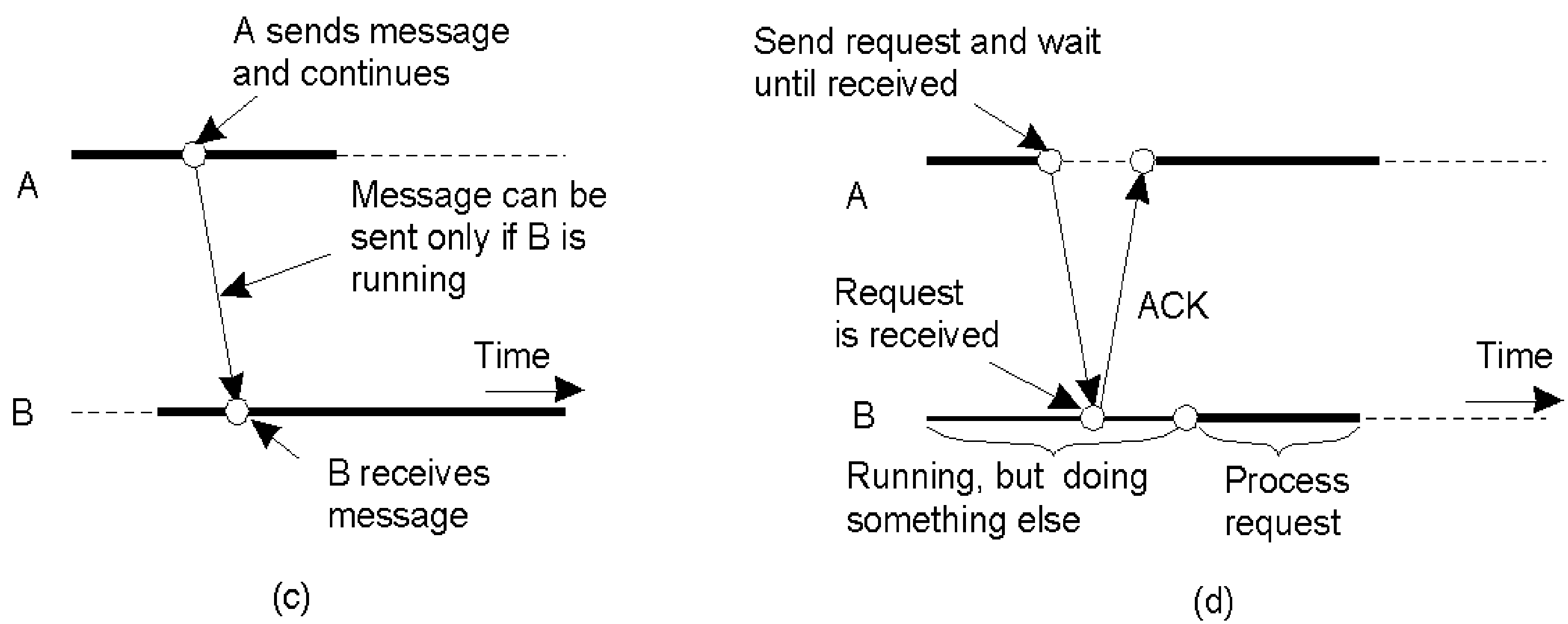
- Persistent Asynchronous Communication
  - Each message is either persistently stored in a buffer at the local host or at the first communication server.
    - A e-mail system is an example
- Persistent Synchronous Communication
  - Messages can be persistently stored at the receiving host and a sender is blocked until this happens
- Transient Asynchronous Communication
  - The message is temporarily stored at a local buffer at the sending host, after which the sender immediately continues
  - UDP is an example
- Transient Synchronous Communication
  - The sender is blocked until the message is stored in a local buffer at the receiving host, or
  - until the message is delivered to the receiver for further processing, or
  - until it receives a reply message from the other side (RPCs,RMIs)

Persistence and Synchronicity in Communication



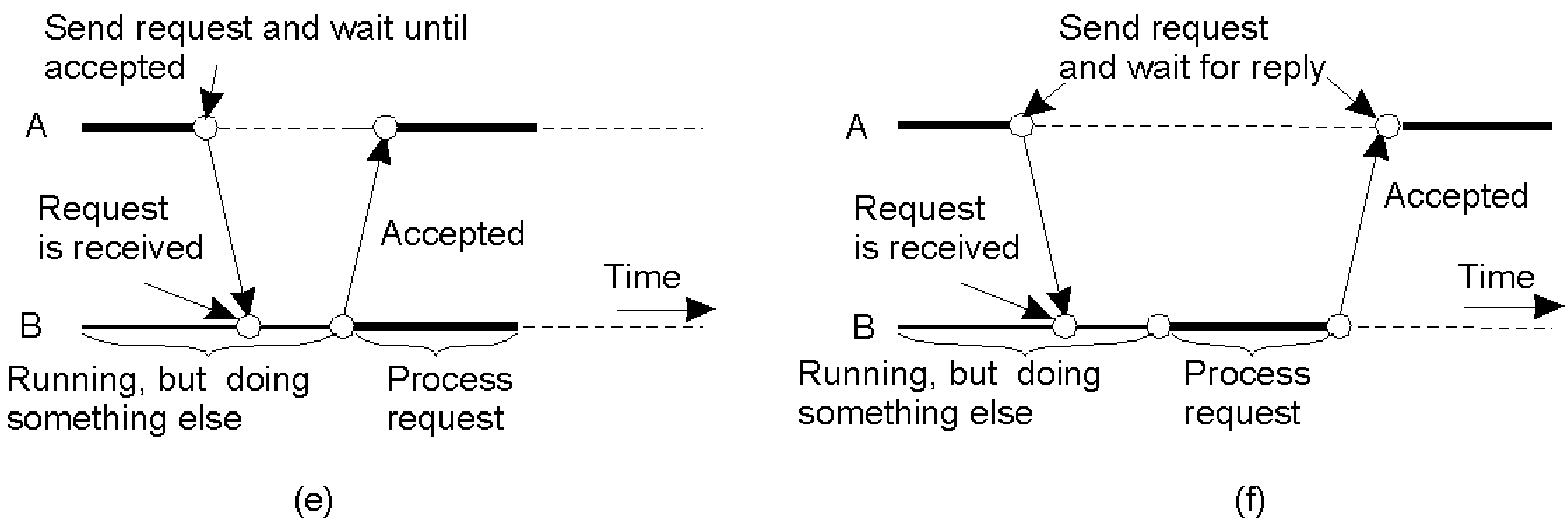
a) Persistent asynchronous communication
b) Persistent synchronous communication

Persistence and Synchronicity in Communication



c) Transient asynchronous communication
d) Receipt-based transient synchronous communication

Persistence and Synchronicity in Communication



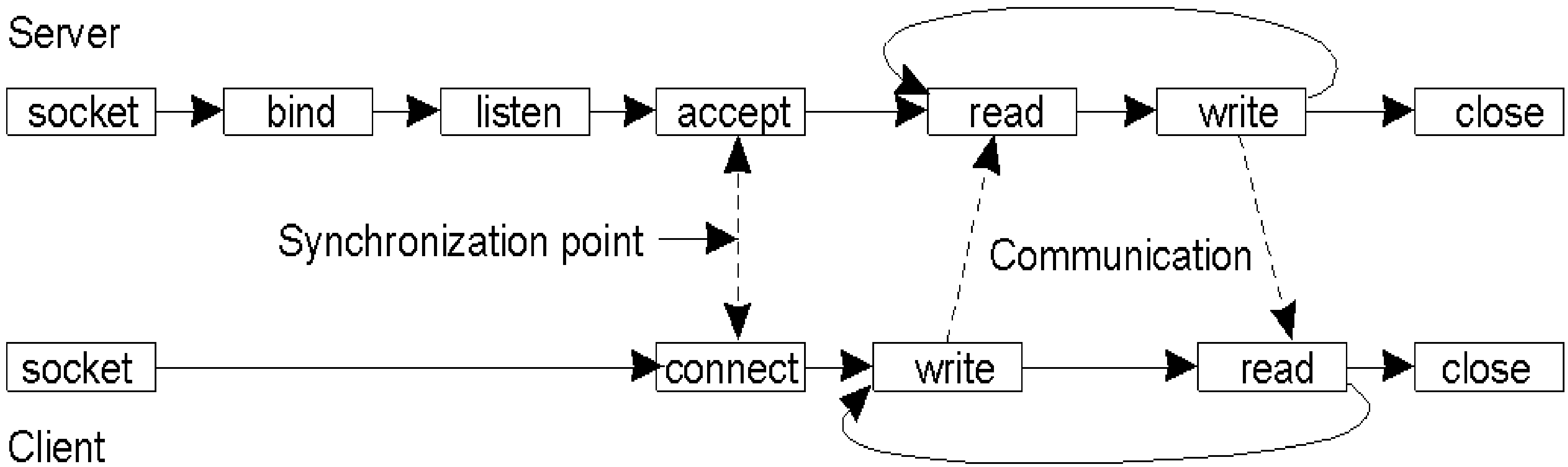
e) Delivery-based transient synchronous communication at message delivery
f) Response-based transient synchronous communication

Berkeley Sockets

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

- A socket is a communication endpoint to which an application can write data that are to be sent out over the underlying network, and from which incoming data can be read.
- Socket primitives for TCP/IP.

Berkeley Sockets



- Connection-oriented communication pattern using sockets.

Sockets

The BSD UNIX supports:

- The "UNIX system domain" for processes communicating on one machine
- The "Internat domain" for processes communicating over the Internet (using the DARPA [Defense Advanced Research Project] communications protocols)

Each socket has a type:

- Stream (default: TCP)
- Datagram (default: UDP)

bind(sd, <sockname>,length): associates the name <sockname> with the socket sd.

listen(sd,qlength): specifies the maximum length of the queue which stores incoming requests for connections to the socket

```
#include <sys/types.h>
#include <sys/socket.h>
#define BUF_LEN 256
#define ADDRESS "mysocket"

int main(void) {
        int sd,ns, len, fromlen;
        char buf[BUF_LEN];
        struct sockaddr_un sockaddr, clientsockaddr;

        sd = socket(AF_UNIX,SOCK_STREAM,0);

        sockaddr.sun_family = AF_UNIX;
        strcpy(sockaddr.sun_path, ADDRESS);
        len = sizeof(sockaddr.sun_family) +
      strlen(sockaddr.sun_path);

        bind(sd, (struct sockaddr *)&sockaddr, len);
        listen(sd,1);

        while  (1) {
                ns = accept(sd,&clientsockaddr,&fromlen);
                if (fork() == 0) {      // child code
                        close(sd);
                        read(ns,buf,sizeof(buf));
                        printf("server read '%s'\n", buf);
                        exit(0);
                }
                close(ns);
                sleep(3);
        }
}
```

A server process in the UNIX System Domain

Sockets

```
#include <sys/types.h>
#include <sys/socket.h>
#define BUF_LEN 256
#define ADDRESS "mysocket"

int main(void) {
      int sd, len;
      struct sockaddr_un sockaddr;

      sd = socket(AF_UNIX,SOCK_STREAM,0);

      sockaddr.sun_family = AF_UNIX;
      strcpy(sockaddr.sun_path, ADDRESS);

      len = sizeof(sockaddr.sun_family) + strlen(sockaddr.sun_path);

      if (connect(sd,&sockaddr, len) == -1)
              exit(1);
      write(sd,"hi guy",6);
      close(sd);
}
```

A client process in the UNIX System Domain

The Message-Passing Interface (MPI)

- Some of the most intuitive message-passing primitives of MPI.

Primitive	**Meaning**
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_recv	Receive a message; block if there are none
MPI_irecv	Check if there is an incoming message, but do not block

MPI – A Simple Example

```
/*The Parallel Hello World Program*/
#include <stdio.h>
#include <mpi.h>
int main(int argc, char **argv) {
     int node;
     char buf[64];
     FILE *fp;

     MPI_Init(&argc,&argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &node);
     sprintf(buf,"file%d",node);
      fp = fopen(buf,"r");
     fprintf(fp, "Hello World from Node %d\n",node);
      fclose(fp);
     MPI_Finalize();
}
```

MPI - Basic Concepts

- A communicator is a collection of processes that can send messages to each other.
- There is a default communicator whose group contains all initial processes, called **MPI_COMM_WORLD**.
- A process is identified by its *rank* in the group associated with a communicator.

MPI – A simple example with send-receive

```
int main(int argc, char **argv) {
    int rank, size;
    double x[10];
    MPI_Status status;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        for (int i=0; i<10 ; i++) x[i] = 0.1*i;
        MPI_Send(x, 10, MPI_DOUBLE, 1, 666, MPI_COMM_WORLD);
    }
    else if (rank == 1) {
        MPI_Recv(x, 10, MPI_DOUBLE, 0, 666, MPI_COMM_WORLD, &status);
    }
    MPI_Finalize();
}
```

Send an array from process 0 to 1

- int MPI_Send(void 8message, int count, MPI_Datatype datatype, int dest, int tag)
- int MPI_recv(void *message, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive.

Another Simple Example

```
# include <stdio.h>
#include “mpi.h”

int main(int argc, char **argv) {
      int my_rank;                    /* rank of process */
      int p;                          /* number of processes */
      int source;                     /* rank of sender */
      int dest;                       /* rand of receiver */
      int tag = 50;                   /* tag for message */
      char message[100];              /* storage for message */
      MPI_Status status;                          /* return status for receive */

      MPI_INIT(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
      MPI_Comm_size(MPI_COMM_WORLD,&p);
      if (my_rank != 0) {
            sprinf(message, “Greetings from process %d!”, my_rank);
            dest =0;
            MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
      else {
            for (source = 1; source < p; sopurce++) {
                        MPI_Recv(message,100, MPI_CHAR, source, tag, MPI_COMM_WORLD,
      &status);
                        printf(“%s\n”, message);
            }
      }
}
```

Introduction to Collective Operations in MPI

- Collective operations are called by all processes in a communicator.
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator.
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency.

MPI - Further Information

- Online examples available at http://www.mcs.anl.gov/mpi/tutorials/perf
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Skjellum, MIT Press, 1994.
  - *MPI: The Complete Reference,* by Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press, 1996.
  - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - *MPI: The Complete Reference Vol 1 and 2,*MIT Press, 1998(Fall).