

# Slides for Chapter 4: Interprocess Communication (IPC)

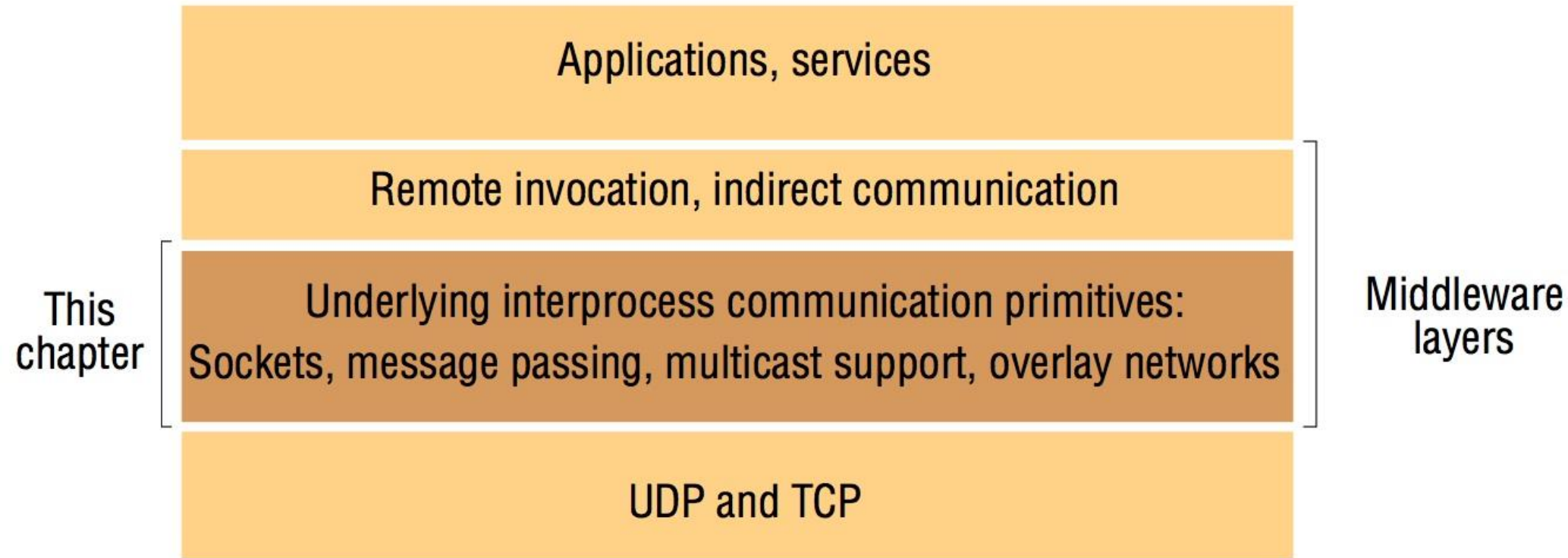
---



*From* **Coulouris, Dollimore, Kindberg and Blair**  
**Distributed Systems:**  
**Concepts and Design**

Edition 5, © Addison-Wesley 2012

Figure 4.1  
Middleware layers



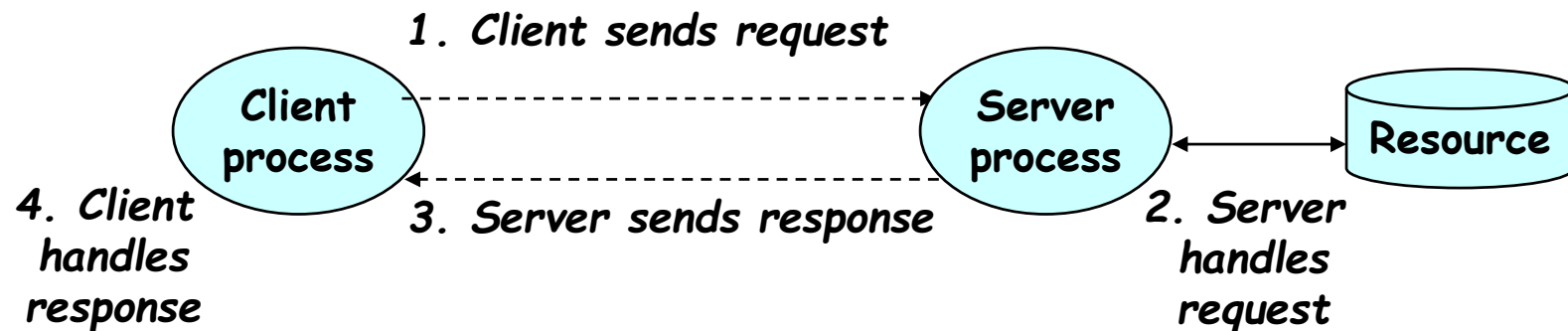
# Introduction

---

- Typical applications today consist of many cooperating processes either on the same host or on different hosts.
- For example, consider a client-server application. How to share (large amounts of ) data?
- Share files? How to avoid contention? What kind of system support is available?
- We want a general mechanism that will work for processes irrespective of their location.

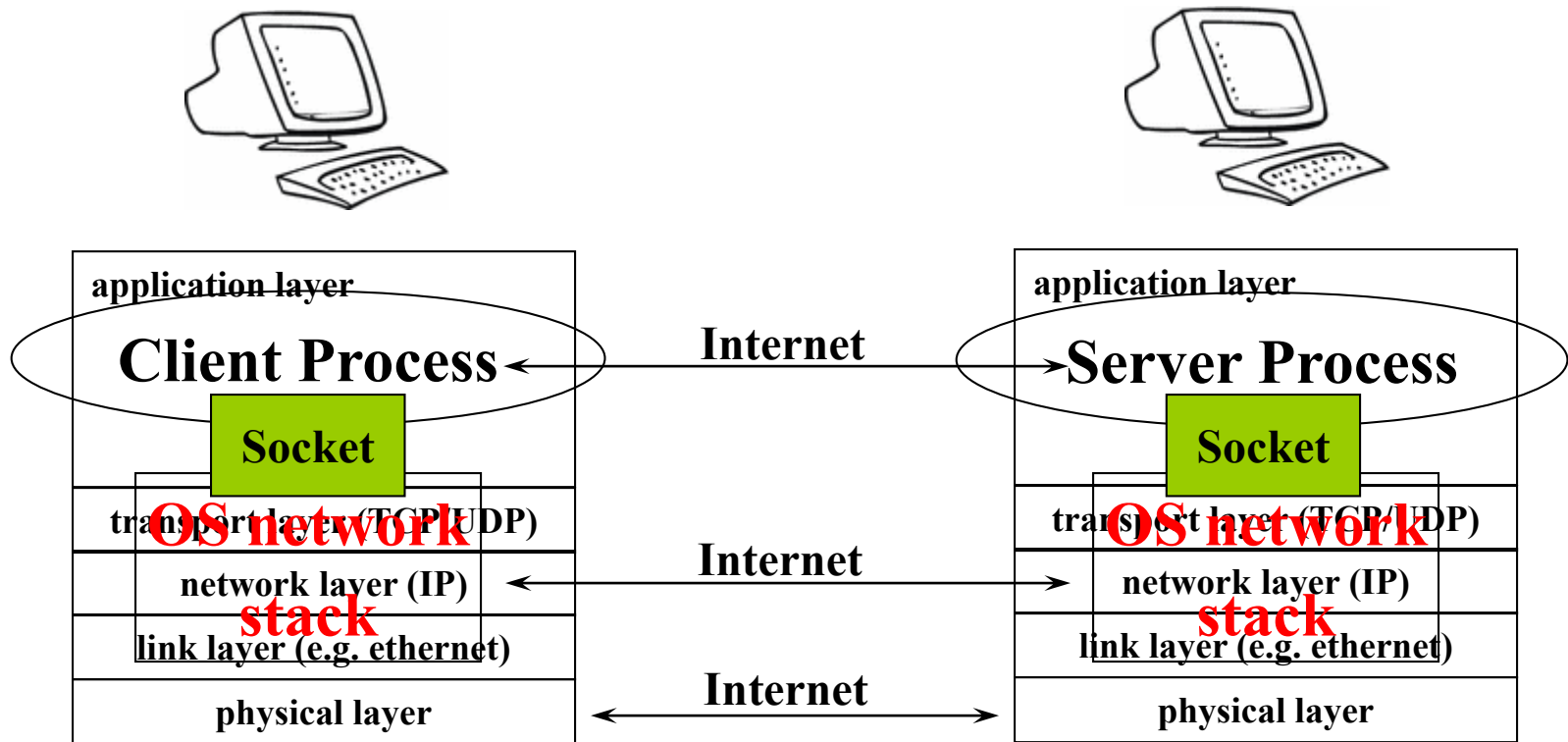
# Example: a client-server application

- Client asks (*request*) – server provides (*response*)
- Typically: single server - multiple clients
- The server does not need to know *anything* about the client even that it exists
- The client should always know *something* about the server at least where it is located



**Note:** *clients and servers are processes running on hosts (can be the same or different hosts).*

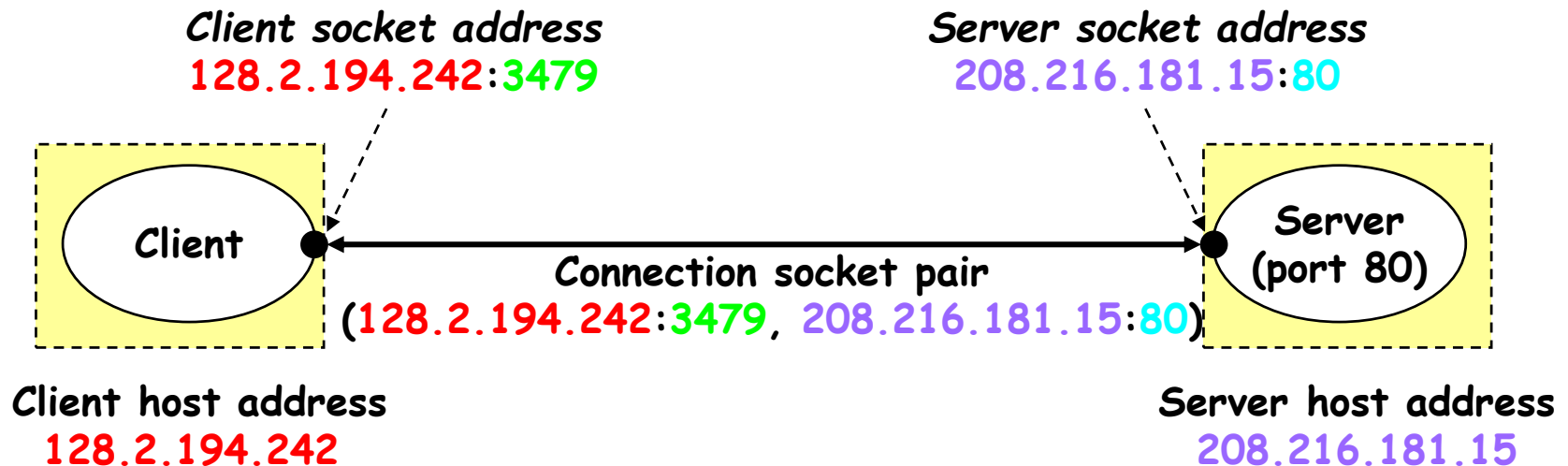
# Sockets as means for IPC



The interface that the OS provides to its networking subsystem

# Internet Connections (TCP/IP)

- Address the machine on the network  
By IP address
- Address the process  
By the “port”-number
- The pair of *IP-address + port* – makes up a “socket-address”



Note: 3479 is an ephemeral port allocated by the kernel

Note: 80 is a well-known port associated with Web servers

# Clients

---

## Examples of client programs

- Web browsers, `ftp`, `telnet`, `ssh`

## How does a client find the server?

- The IP address in the server socket address identifies the host

- The (well-known) port in the server socket address identifies the service, and thus implicitly identifies the server process that performs that service.

- Examples of well known ports

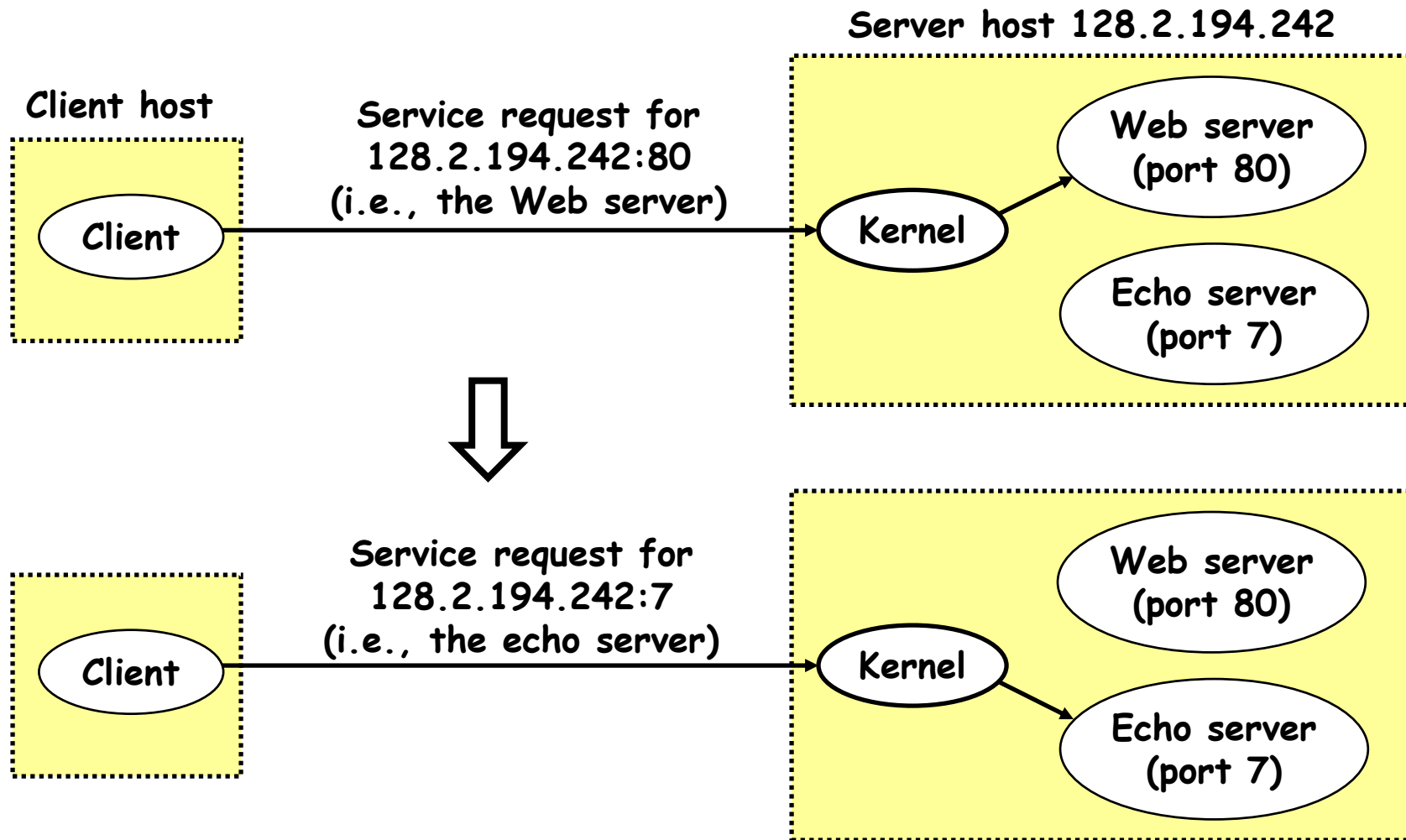
Port 7: Echo server

Port 23: Telnet server

Port 25: Mail server

Port 80: Web server

# Using Ports to Identify Services





# Servers

- Servers are long-running processes (daemons). Created at boot-time (typically) by the init process (process 1) Run continuously until the machine is turned off.
- Each server waits for requests to arrive on a well-known port associated with a particular service.

Port 7: echo server

Port 23: telnet server

Port 25: mail server

Port 80: HTTP server

**See `/etc/services` for a comprehensive list of the services available on a Linux machine.**

- Other applications should choose between 1024 and 65535

# Purposes of IPC

---

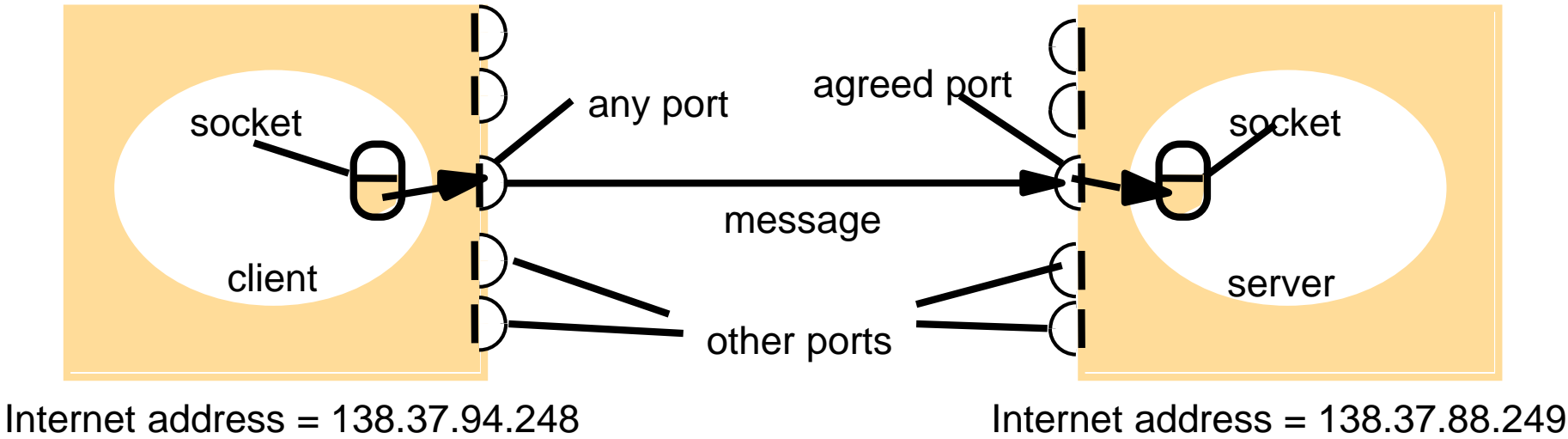
- Data transfer
- Sharing data
- Event notification
- Process control

# What are sockets?

---

- Socket is an abstraction for an end point of communication that can be manipulated with a file descriptor.
- It is an abstract object from which messages are sent and received.
- Sockets are created within a communication domain just as files are created within a file system.
- A communication domain is an abstraction introduced to bundle common properties of processes communicating through sockets. Example: UNIX domain, internet domain.

Figure 4.2  
Sockets and ports



# IPC

---

- IP address and port number. In IPv4 about  $2^{16}$  ports are available for use by user processes.
- UDP and TCP abstraction of the above is a socket.
- Socket is associated with a protocol.
- IPC is transmitting a message between a socket in one process to a socket in another process.
- Messages sent to particular IP and port# can be received by the process whose socket is associated with that IP and port#.
- Processes cannot share ports with other processes within the computer. Can receive messages on diff ports.

# Socket Names

---

- Applications refer to sockets by name.
- But within the communication domain sockets are referred by addresses.
- Name to address translation is usually done outside the operating system.

# Socket types

---

- The format in which an address is specified is according to a domain:
- AF\_UNIX (address format of UNIX) - a path name within the file system,
- AF\_INET (internet format) : network address, port number etc.
- Communication style: stream , datagram, raw or sequenced packets
- Stream : reliable, error-free, connection-oriented comm.
- Datagram: Connectionless, unreliable, message boundaries preserved.

# Functions: creation

---

- Socket creation : socket system call creates sockets on demand.

`sockid = socket (af, type, protocol);`

where sockid is an int,

af - address family , AF\_INET, AF\_UNIX, AF\_APPLETALK etc.

type - communication type:

    SOCK\_STREAM, SOCK\_DGRAM, SOCK\_RAW etc.

protocol - some domains have multiple protocol, use a 0 for your appl.

Example: `door1 = socket(AF_UNIX, SOCK_DGRAM,0);`



# Functions: bind

---

- Socket binding: A socket is created without any association to local or destination address. It is possible to bind the socket to a specific host and in it a specific port number.
- `socerr = bind (sockid, localaddr, addrlen)`
- **localaddr** - a struct of a specific format for each address domain;
- **addrlen** - is the length of this struct; obtained usually by *sizeof* function.

# Functions: bind

---

Example: type *sockaddr\_un* defines localaddr format for unix family.

its definition, (you don't have to define it... it is in *un.h* file - include this file)

```
struct sockaddr_un {  
    short sun_family;  
    char sun_path[108]; };
```

in your program:

```
#define SocName "testsock"  
sockaddr_un mysoc;  
mysoc.sun_family = AF_UNIX;  
strcpy(mysoc.sun_path, SocName);  
binderr = bind(sockid, &mysoc, sizeof(mysoc));
```

# Functions: close

---

- `close (s)`; closes the specified socket. This is done by a process or thread when it no longer needs the socket connection. Premature closing results in “broken pipe” error.
- When a socket is created it is represented by a special file (‘s’ in the place where d appears for directory files when you `ls -l`). The name of the file is the name assigned in the socket bind command.

# Functions: connect

---

- A socket is created in an unconnected state, which means that the socket is not associated with any destination.
- An application program should call connect to establish a connection before it can transfer data thru' reliable stream socket. For datagrams connect is not required but recommended.
- `connect ( sockid, destaddr, addlength);`
- Example: `if (connect(sock, &server, sizeof(server)) < 0) ...`
- “sendto” command does not need “connect”

# Functions: sending

---

- Five different system calls : **send**, **sendto**, **sendmsg**, **write**, **writv**
- **send**, **write** and **writv** work only with connected sockets. No parameter for destination address. Prior connect should be present for communication.
- Example : `write (sock, DATA, sizeof(DATA));`
- **sendto (socket, message, length, flags, destaddr, addrlen);**
- flags allow for special processing of messages. Use 0 in your appln.
- **sendmsg** is same as **sendto** except that it allows for different message structure.

# Functions: receiving

---

- Five different calls are available: read, readv, recv, recvfrom, recvmsg
- read, readv, and recv are for connection-oriented comm.
- **read(socdescriptor, buffer, length); Example: read(sock, buf, 1024);**
- For your application (project) you may use read.
- For connectionless, datagram-kind :
- recvfrom(same set of params as sendto); except that message length and addr length return values.

# Functions: listen

---

- accept and listen are connection-oriented communication.
- These are for AF\_INET, SOCK\_STREAM type of sockets.
- listen: To avoid having protocols reject incoming request, a server may have to specify how many messages need to be queued until it has time to process them. Example:  
`listen(socket,length);`
- accept: wait for the call. Example: `accept(sockid, sockaddr, sizeof sockaddr);`

# Functions: accept

---

- **accept** : blocks until a connect calls the socket associated with this connection. socket -- bind --(listen) -- accept is the sequence. “connect” from calling process will complete the connection and unblock the process or thread that is blocked on “accept”
- Now read, write or writev can be executed to carry out the actual communication over the connection established.



# Functions: getsockname

---


- **getsockname (sockid, sockaddr, sizeof sockaddr);** : given a sockid returns the address of the socket identified by sockid.
- This address may be needed, for instance, by an accept function call.
- There are other functions such as gethostbyname may be needed for internet domain sockets. See man pages for the details.

# Sockets used for datagrams

---

Sending a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ClientAddress)
•
•
sendto(s, "message", ServerAddress)
```

A black arrow points from the `sendto` function in the 'Sending a message' box to the `recvfrom` function in the 'Receiving a message' box, indicating the transfer of data.

Receiving a message

```
s = socket(AF_INET, SOCK_DGRAM, 0)
•
•
bind(s, ServerAddress)
•
•
amount = recvfrom(s, buffer, from)
```

*ServerAddress* and *ClientAddress* are socket addresses

# Sockets used for streams

## Requesting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
•
connect(s, ServerAddress)
•
•
write(s, "message", length)
```

## Listening and accepting a connection

```
s = socket(AF_INET, SOCK_STREAM, 0)
•
bind(s, ServerAddress);
listen(s, 5);
•
sNew = accept(s, ClientAddress);
•
n = read(sNew, buffer, amount)
```

*ServerAddress* and *ClientAddress* are socket addresses

# Sockets on Unix

---

- When a program calls fork, the newly created process inherits access to all open sockets.
- For threads socket identifiers should be defined in the common address space or passed as parameters.
- include files : <socket.h>, <un.h> or <in.h>, and other related header files.
- When linking add a **-lsocket -lnsl** options besides others you may use.

# A review of previous works

---

- Message passing refers to a means of communication between
  - different threads within a process
  - different processes running on the same node
  - different processes running on different nodes
- When messages are passed between two different processes we speak of *inter-process communication*, or *IPC*.

# A review of previous works

---

- Message passing can be used as a more process-oriented approach to synchronization than the "data-oriented" approaches used in providing mutual exclusion for shared resources.
- The two main dimensions
  - Symmetric or asymmetric process/thread naming
  - Synchronous vs. asynchronous

# A review of previous works

## Simple Message Passing

- One process/thread is the sender and another is the receiver

### Symmetric Naming

Process P Sender	Process Q Receiver
. . send(Q, message); . .	. . receive(P, &message); . .

### Asymmetric Naming

Process P Sender	Process Q Receiver
. . send(Q, message); . .	. . receive(&message); . .

(\*) The receiver will accept a message from any sender. The sender can pass its own id inside the message if it wants.

# A review of previous works

---

## Simple Message Passing

- One process/thread is the sender and another is the receiver

## Synchronous Communication

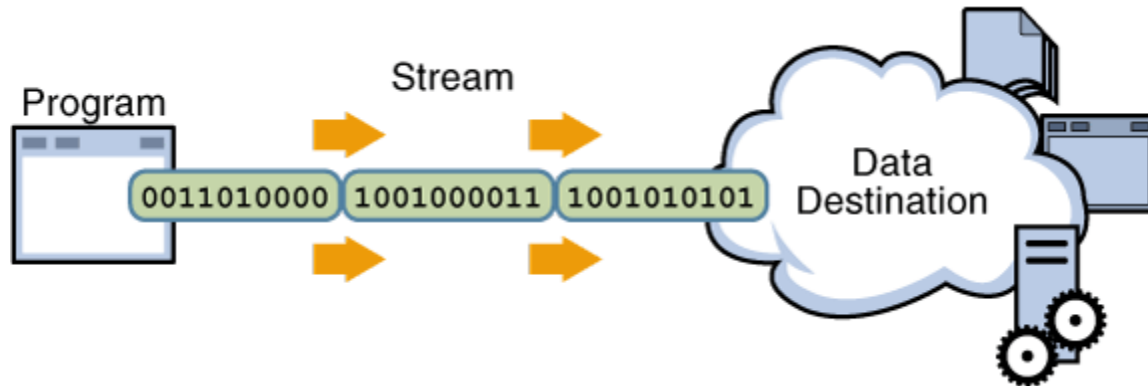
P and Q have to wait for each other (one blocks until the other is ready).

## Asynchronous Communication

Underlying system buffers the messages so P and Q don't have to wait for each other. *This is less efficient due to the overhead of managing the buffer.*



- [illegible]



## 4.3. External data representation and marshalling

---

**Problem?** Different machines have different primitive data reps,

1. Integers: big-endian and little-endian order
2. float-type: representation differs between architectures
3. char codes: ASCII, Unicode

For example, IBMs are little endian, while Motorolas and Suns are big endian. MIPS processors allowed you to select a configuration where it would be big or little endian.

## 4.3. External data representation and marshalling

### 1. big-endian and little-endian order

For example, we have a 32 bit quantity, 90AB12CD16

#### **Big Endian**

you store the most significant byte in the smallest address. Here's how it would look:

Address	Value
1000	90
1001	AB
1002	12
1003	CD

#### **Little Endian**

you store the *least* significant byte in the smallest address. Here's how it would look:

Address	Value
1000	CD
1001	12
1002	AB
1003	90

## 4.3. External data representation and marshalling

### 1. big-endian and little-endian order

For example, we have a 32 bit quantity, 90AB12CD16

#### **Big Endian**

you store the most significant byte in the smallest address. Here's how it would look:

Address	Value
1000	90
1001	AB
1002	12
1003	CD

#### **Little Endian**

you store the *least* significant byte in the smallest address. Here's how it would look:

Address	Value
1000	CD
1001	12
1002	AB
1003	90

## 4.3. External data representation and marshalling

---

So, we must either:

Have both sides **agree on an external representation** or transmit in the sender's format along with an indication of the format used. The receiver converts to its form.

- External data representation: an agreed standard for the representation of data structures and primitive values
- e.g., CORBA Common Data Rep (CDR) for many languages; Java object serialization for Java code only

## 4.3. External data representation and marshalling

---

- Marshalling: process of taking a collection of data items and assembling them into a form suitable for transmission
- Unmarshalling: disassembling (restoring) to original on arrival
- Three alter. approaches to external data representation and marshelling:
  1. CORBA's common data representation (CDR)
  2. Java's object serialization
  3. XML (Extensible Markup Language) : defines a textual format for rep. structured data

## 4.3. External data representation and marshalling

---

First two: marshalling & unmarshalling carried out by middleware layer

- XML: software available

First two: primitive data types are marshalled into a binary form

- XML: represented textually

Whether the marshalled data include info concerning type of its contents?

- CDR: no, just the values of the objects transmitted
- Java: yes, type info in the serialized form
- XML: yes, type info refer to externally defined sets of names (with types), *namespaces*

## 4.3. External data representation and marshalling

---

- Although we are interested in the use of external data representation for the arguments and results of RMI and RPCs, it has a more general use for representing data structures, objects, or structured documents in a form suitable for transmission or storing in files



## 4.3. External data representation and marshalling

### CORBA CDR

- 15 primitive types: short, long, unsigned short, unsigned long, float, double, char, boolean, octet, any
- Constructed types: sequence, string, array, struct, enum and union

note that it does not deal with objects (*only Java does: objects and tree of objects*)

<i>Type</i>	<i>Representation</i>
<i>sequence</i>	length (unsigned long) followed by elements in order
<i>string</i>	length (unsigned long) followed by characters in order (can also can have wide characters)
<i>array</i>	array elements in order (no length specified because it is fixed)
<i>struct</i>	in the order of declaration of the components
<i>enumerated</i>	unsigned long (the values are specified by the order declared)
<i>union</i>	.type tag followed by the selected member

## 4.3. External data representation and marshalling

<i>index in sequence of bytes</i>	<i>4 bytes</i>	<i>notes on representation</i>
0–3	5	<i>length of string</i>
4–7	"Smit"	<i>'Smith'</i>
8–11	"h "	
12–15	6	<i>length of string</i>
16–19	"Lond"	<i>'London'</i>
20–23	"on "	
24–27	1984	<i>unsigned long</i>

The flattened form represents a *Person* struct with value: {'Smith', 'London', 1984}

## 4.3. External data representation and marshalling

- Type of a data item not given: assumed sender and recipient have common knowledge of the order and types of data items
- Types of data structures and types of basic data items are described in CORBA IDL
  - Provides a notation for describing the types of arguments and results of RMI methods

```
Struct Person {  
    string name;  
    string place;  
    unsigned long year;  
};
```

## 4.4. Multicast communication

---

A multicast address is a destination address for a group of hosts that have joined a multicast group. A packet that uses a multicast address as a destination can reach all members of the group unless there are some filtering restriction by the receiver.

## 4.4. Multicast communication

**Table 12.1** *Multicast Address Ranges*

<i>CIDR</i>	<i>Range</i>	<i>Assignment</i>
224.0.0.0/24	224.0.0.0 → 224.0.0.255	Local Network Control Block
224.0.1.0/24	224.0.1.0 → 224.0.1.255	Internetwork Control Block
	224.0.2.0 → 224.0.255.255	AD HOC Block
224.1.0.0/16	224.1.0.0 → 224.1.255.255	ST Multicast Group Block
224.2.0.0/16	224.2.0.0 → 224.2.255.255	SDP/SAP Block
	224.3.0.0 → 231.255.255.255	Reserved
232.0.0.0/8	232.0.0.0 → 224.255.255.255	Source Specific Multicast (SSM)
233.0.0.0/8	233.0.0.0 → 233.255.255.255	GLOP Block
	234.0.0.0 → 238.255.255.255	Reserved
239.0.0.0/8	239.0.0.0 → 239.255.255.255	Administratively Scoped Block

## 4.5 Overlay networks

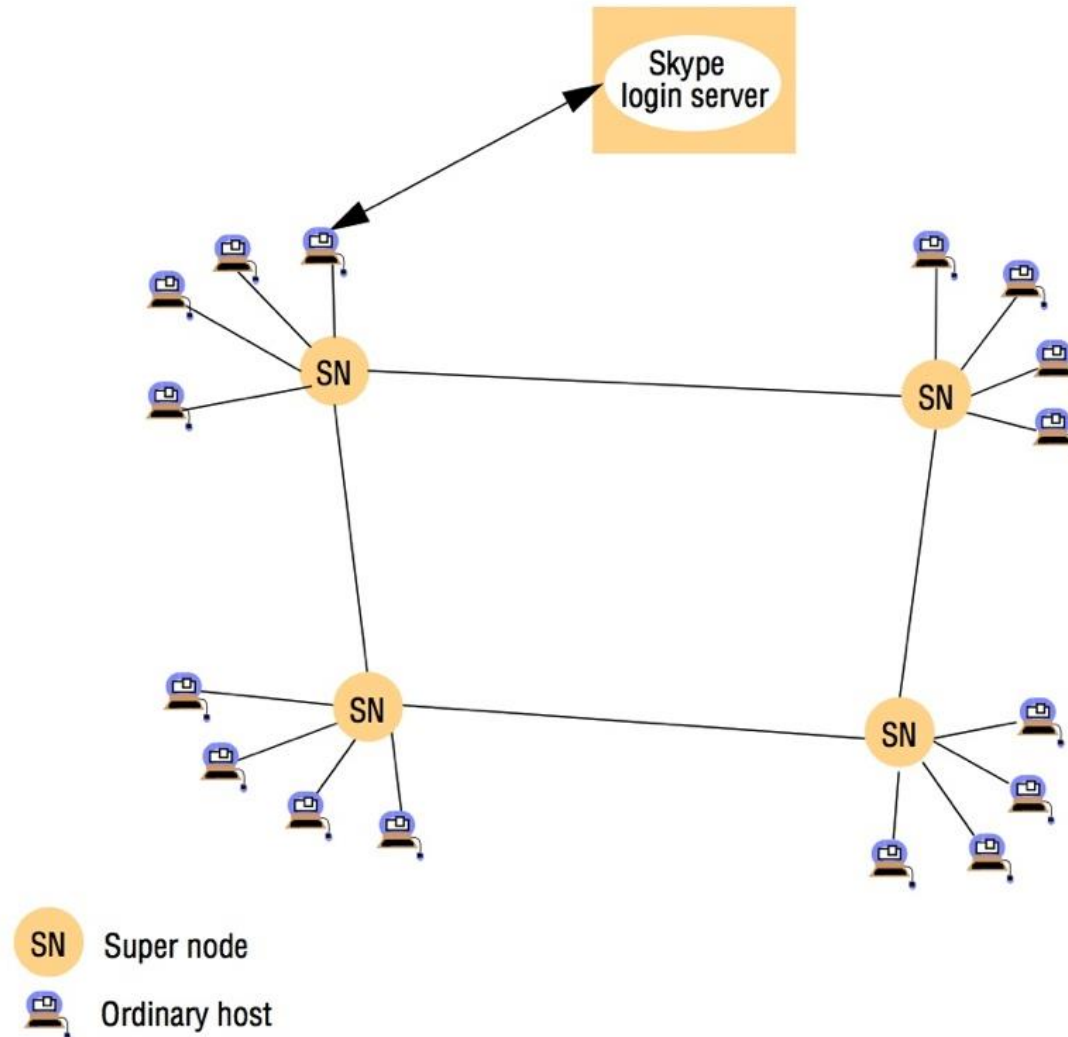
<i>Motivation</i>	<i>Type</i>	<i>Description</i>
<i>Tailored for application needs</i>	Distributed hash tables	One of the most prominent classes of overlay network, offering a service that manages a mapping from keys to values across a potentially large number of nodes in a completely decentralized manner (similar to a standard hash table but in a networked environment).
	Peer-to-peer file sharing	Overlay structures that focus on constructing tailored addressing and routing mechanisms to support the cooperative discovery and use (for example, download) of files.
	Content distribution networks	Overlays that subsume a range of replication, caching and placement strategies to provide improved performance in terms of content delivery to web users; used for web acceleration and to offer the required real-time performance for video streaming [ <a href="http://www.kontiki.com">www.kontiki.com</a> ].

## 4.5 Overlay networks

<i>Tailored for network style</i>	Wireless ad hoc networks	Network overlays that provide customized routing protocols for wireless ad hoc networks, including proactive schemes that effectively construct a routing topology on top of the underlying nodes and reactive schemes that establish routes on demand typically supported by flooding.
	Disruption-tolerant networks	Overlays designed to operate in hostile environments that suffer significant node or link failure and potentially high delays.
<i>Offering additional features</i>	Multicast	One of the earliest uses of overlay networks in the Internet, providing access to multicast services where multicast routers are not available; builds on the work by Van Jacobsen, Deering and Casner with their implementation of the MBone (or Multicast Backbone) [ <a href="#">mbone</a> ].
	Resilience	Overlay networks that seek an order of magnitude improvement in robustness and availability of Internet paths [ <a href="#">nms.csail.mit.edu</a> ].
	Security	Overlay networks that offer enhanced security over the underling IP network, including virtual private networks, for example, as discussed in Section 3.4.8.

---

Figure 4.16  
Skype overlay architecture





## 4.6 Case study: MPI

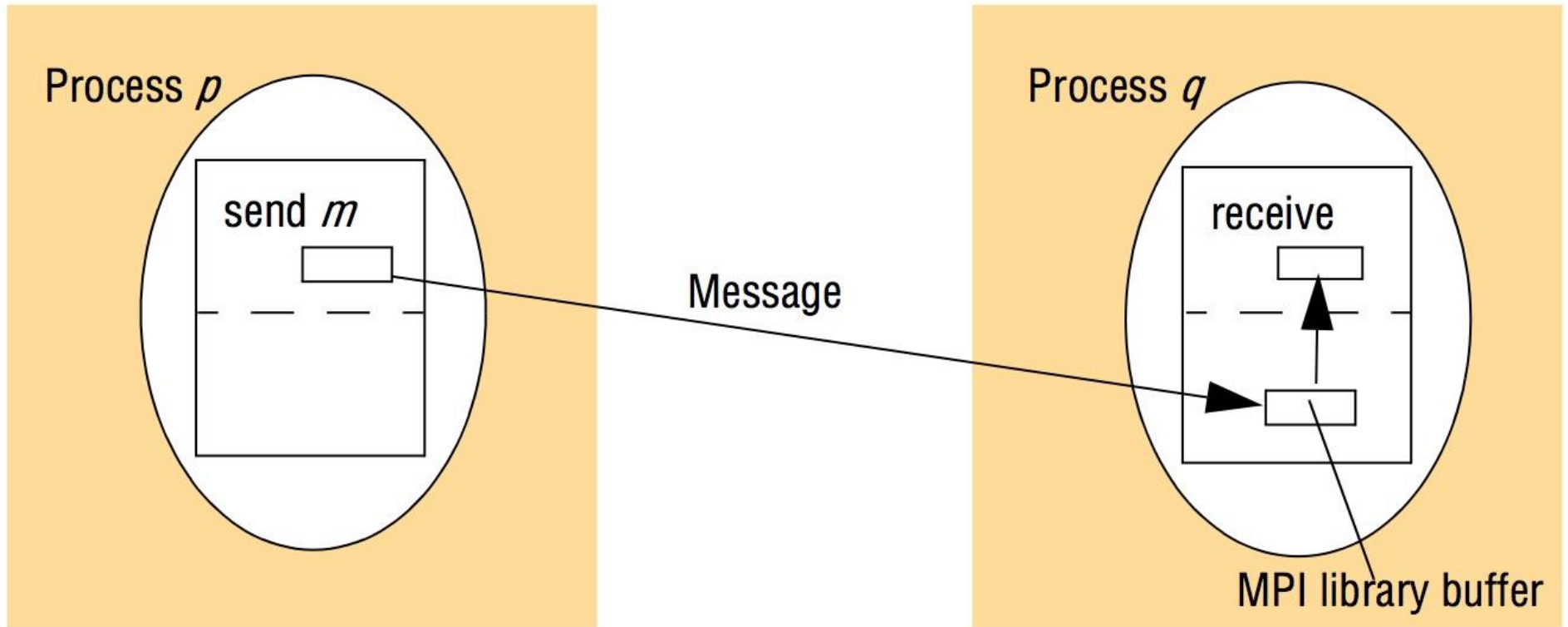


Figure 4.18  
Selected send operations in MPI

<i>Send operations</i>	<i>Blocking</i>	<i>Non-blocking</i>
<i>Generic</i>	<i>MPI_Send</i> : the sender blocks until it is safe to return – that is, until the message is in transit or delivered and the sender's application buffer can therefore be reused.	<i>MPI_Isend</i> : the call returns immediately and the programmer is given a communication request handle, which can then be used to check the progress of the call via <i>MPI_Wait</i> or <i>MPI_Test</i> .
<i>Synchronous</i>	<i>MPI_Ssend</i> : the sender and receiver synchronize and the call only returns when the message has been delivered at the receiving end.	<i>MPI_Issend</i> : as with <i>MPI_Isend</i> , but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been delivered at the receive end.
<i>Buffered</i>	<i>MPI_Bsend</i> : the sender explicitly allocates an MPI buffer library (using a separate <i>MPI_Buffer_attach</i> call) and the call returns when the data is successfully copied into this buffer.	<i>MPI_Ibsend</i> : as with <i>MPI_Isend</i> but with <i>MPI_Wait</i> and <i>MPI_Test</i> indicating whether the message has been copied into the sender's MPI buffer and hence is in transit.
<i>Ready</i>	<i>MPI_Rsend</i> : the call returns when the sender's application buffer can be reused (as with <i>MPI_Send</i> ), but the programmer is also indicating to the library that the receiver is ready to receive the message, resulting in potential optimization of the underlying implementation.	<i>MPI_Irsend</i> : the effect is as with <i>MPI_Isend</i> , but as with <i>MPI_Rsend</i> , the programmer is indicating to the underlying implementation that the receiver is guaranteed to be ready to receive (resulting in the same optimizations),