COMP 242 Class Notes

Section 3: Interprocess Communication

1. Interprocess Communication

Readings: Chapter 7, Comer; Sections 2.6.4, 2.6.5, 4.6, 4.7 Singhal.

An operating system provides **interprocess communication** to allow processes to exchange information. Interprocess communication is useful for creating *cooperating* processes. For instance an 'ls' process and a 'more' process can cooperate to produce a paged listing of a directory.

There are several mechanisms for interprocess communication. We discuss some of these below.

1.1. Shared Memory

Processes that share memory can exchange data by writing and reading shared variables. As an example, consider two processes *p* and *q* that share some variable *mesg*. Then *p* can communicate information to *q* by writing new data in *mesg*, which *q* can then read.

The above discussion raises an important question. How does *q* know when *p* writes new information into *mesg*? In some cases, q does not need to know. For instance, if it is a load balancing program that simply looks at the current load in p's machine stored in mesg. When it does need to know, it could poll, but polling puts undue burden on the cpu. Another possibility is that it could get a *software interrupt*, which we discuss below. A familiar alternative is to use semaphores or conditions. Process *q* could block till *p* changes *mesg* and sends a signal that unblocks *q*. However, these solutions would not allow *q* to (automatically) block if *mesg* cannot hold all the data it wants to write. (The programmer could manually implement a bounded buffer using semaphores.) Moreover, conventional shared memory is accessed by processes on a single machine, so it cannot be used for communicating information among remote processes. Recently, there has been a lot of work in distributed shared memory, which you will study in 203/243.

1.2. Software Interrupt

We can introduce a service call that lets one process cause a software interrupt in another:

```
Interrupt(process id, interrupt number)
```

and another that allows a process to associate a **handler** with an interrupt:

```
Handle(interrupt number, handler)
```

Software interrupts are often used by an operating system to inform a process about the following events:

- The user typed the "attention key".
- An alarm scheduled by the process has expired.
- Some limit, such as file size or virtual time, has been exceeded.

It is important to distinguish among interrupts, traps, and software interrupts. In all cases, the handler is a software program. Interrupt and trap numbers are defined by the hardware which is also responsible for calling the procedure. Software trap numbers are defined by the operating system which is responsible for calling the procedure. An interrupt handler is called in response to a signal from another device while a trap handler is called in response to an instruction executed within the cpu. A software interrupt handler is called in response to the invocation of a system call.

1.3. Message Passing

The most popular form of interprocess communication involves **message passing**. Processes communicate with each other by exchanging messages. A process may **send** information to a **port**, from which another process may **receive** information. The sending and receiving processes can be on the same or different computers connected via a communication medium.

One reason for the popularity of message passing is its ability to support client-server interaction. A **server** is a process that offers a set of services to **client** processes. These services are invoked in response to messages from the clients and results are returned in messages to the client. Thus a process may act as a *sort server* by accepting messages that ask it to sort a list of data items.

In this course we shall be particularly interested in servers that offer operating system services. With such servers, part of the operating system functionality can be transferred from the kernel to utility processes. For instance file management can be handled by a *file server*, which offers services such as *open*, *read*, *write*, and *seek*. Similarly, terminal management can also be handled by a server that offers services such as *getchar* and *putchar*.

There are several issues involved in message passing. We discuss some of these below.

1.3.1. Reliability of Messages

Messages sent between computers can fail to arrive or can be garbled because of noise and contention for the communication line. There are techniques to increase the reliability of data transfer. However, these techniques cost both extra space (longer messages to increase redundancy, more code to check the messages) and time (delays and retransmissions). Message passing techniques can be distinguished by the reliability by which they deliver messages.

1.3.2. Order

Another issue is whether messages sent to a port are received in the order in which they are sent. Differential buffering delays and routings in a network environment can place messages out of order. It takes extra effort (in the form of sequence number, and more generally, time stamps) to ensure order.

1.3.3. Access

An important issue is how many readers and writers can exchange information at a port. Different approaches impose various restrictions on the access to ports. A **bound port** is the most restrictive: There may be only one reader and writer. At the other extreme, the **free port** allows any number of readers and writers. These are suitable for programming client/server interactions based on a *family* of servers providing a common service. A common service is associated with a single port; clients send their service requests to this port and servers providing the requested service receive service requests from the port. Unfortunately, implementing free ports can be quite costly. The queue associated with the port is kept at a site which, in general, is remote to both a sender and a receiver. Thus both sends and receives result in messages being sent to this site. The former put messages in this queue and the latter request messages from it. (Often the implementation used is a little different from the centralized one described above: When a message is sent to a port, it is relayed to all sites where a receive could be performed on the destination port; then, after a message has been received, all these sites are notified that a message is no longer available for receipt.)

Between these two extremes are **input ports** and **output ports**. An input port has only one reader but any number of writers. It models the fairly typical many client, one server situation. Input ports are easy to implement since all receives that designate a port occur in the same process. Thus the message queue associated with a port can be kept with the receiver. Output ports, in contrast, allow any number of readers but only one writer. They are easier to implement than free ports since the message queue can be kept with the sender. However, they are not popular since the one client, many server situation is very unusual. Note that sending to an output port does not broadcast the message to all the receivers - a specific message is received by a single server. Perhaps a ‘‘practical’’ use of an output port is distribution of tickets or other resources by a ‘‘vendor’’ to a set of ‘‘buyers’’.

Several applications can use more than one kind of port. For instance a client can enclose a bound report in a *request* message to the input report of a server, which can then return results to the client by sending messages to the bound port.

1.3.4. Remote Assignment vs Procedure Call

Some systems such as CSP (Section 2.6.4, Singhal) regard the information in a message to be **data** subject to interpretation by the recipient. In these systems, the *send* operation is of the form:

send `(port, outdata)`

while the receive is of the form:

```
receive (port, indata)
```

The completion of the receive causes the *outdata* to be assigned to *indata* (if the ports named are the same). Thus, a matching send and receive essentially perform a **remote assignment**:

```
indata <- outdata
```

Typically, *indata* and *outdata* are untyped byte streams though with language support typed data may be transmitted in messages.

Other systems, such as Ada (section 2.6.4, Singhal), regard the information in a message to be a *request* for **service**. The service is named by the port, and message contains parameters. In these systems, the *send* operation is of the form:

```
<result> <- send  <port or service name> (<parameters>)
```

and the *receive* operation is of the form:

```
receive <port or service name> (<formal parameters>): <result type>
begin
        service request
        reply (answer)
end
```

(Here, we are assuming language support for specifying these operations)

The *send* operation is similar to a procedure call, and the *receive* operation is similar to a procedure declaration. When the receive succeeds, the parameters of the incoming message are assigned to the formal parameters of the **receive** statement. The receiving process then executes code that services the request, and then sends any results in a **reply** message. The following are examples of matching *send* and *receive*:

```
/* client */
foo <- send add (3, 5);

/* server */
receive add (p1, p2: int): int
begin
        reply (p1 + p2)
end
```

The main difference between the *send* and a procedure call is that the former can result in code being executed in a process on a remote machine. Thus the parameters are *marshalled* into a message from which they are *unmarshalled* into the receiver's address space. Hence, such a send is called a **remote procedure call**. The main difference between the *receive* and a procedure declaration is that the code associated with the *receive* is not executed till the process executes the *receive* operation. At this point, in the case of synchronous RPC, the sender and the receiver are said to be in a **rendezvous**. Later, we shall see other differences between local and remote procedure calls.

The second form of message-passing can be simulated by the first. Thus the following communication according to the first form simulates the 'add' example:

```
/*client*/
send (addPort, 3) /* addPort is an input port */
send (addPort, 5)
send (addPort, replyPort); /* reply port is a bound port */
receive (replyPort, result);

/*server*/
receive (addPort, p1);
receive (addPort, p2);
receive (addPort, p3)
send (replyPort, p1 + p2);
```

Thus the second form of communication is higher-level compared to the first and requires fewer kernel calls. However, it is also less flexible, since it requires *exactly* one reply to a message from the receiver of the message, does not allow incremental transmission of the returned information, and also requires that all parameters of a request come from a single source. Sometimes it is useful if the receiver of a message forwards the request to another server. In such a situation the answer could come from a process other than the one to which the request was made. Also, in **dataflow** computations, often each operand of an operation comes from a different process. a client may want to receive When all operands of an operation arrive at a server process, the operation is said to be triggered. Finally, a ''more'' process receiving a file from an ''ls' process may wish to receive and display the file incrementally. RPC does not directly support the above scenarios.

1.3.5. Synchronous vs Asynchronous

The *send*, *receive*, and *reply* operations may be **synchronous** or **asynchronous**. A synchronous operation **blocks** a process till the operation completes. An asynchronous operation is **non-blocking** and only *initiates* the operation. The caller could discover completion by polling, by software interrupt, or by waiting explicitly for completion later. (Does it make sense to have an RPC send not block?) An asynchronous operation needs to return a call/transaction id if the application needs to be later notified about the operation. At notification time, this id would be placed in some global location or passed as an argument.

The notion of synchronous operations requires an understanding of what it means for an operation to complete. In the case of remote assignment, both the send and receive complete when the message has been delivered to the receiver. In the case of remote procedure call, the send, receive, and reply complete when the result has been delivered to the sender, assuming there is a return value. Otherwise, the send and receive complete when the procedure finishes execution. During the time the procedure is executing, the sender and receiver are in a rendezvous, as mentioned before.

Note that synchronous/asynchronous implies blocking/not blocking but not vice versa, that is, not every blocking operation is synchronous and not every non blocking operation is asynchronous. For instance, a send that blocks till the receiver machine has received the message is blocking but not synchronous since the receiver process may not have received it. Similarly, we will see later a Xinu receive that is non-blocking but is not asynchronous. These definitions of synchronous/asynchronous operations are similar but not identical to the ones given in your text books, which tend to equate synchronous with blocking.

Asynchronous message passing allows more parallelism. Since a process does not block, it can do some computation while the message is in transit. In a synchronous system, such parallelism can be achieved by forking a separate process for each concurrent send, but this approach incurs the cost of extra process management. This cost is typically bearable with lwps but not hwps.

Asynchronous message passing introduces several problems. What happens if the message cannot be delivered? The process never waits for delivery of the message, and thus never hears about the error. Software interrupts could be used to report such errors. Another problem related to asynchronous communication is to do with buffering. If messages sent asynchronously are buffered in a space managed by the OS, then a process may fill this space by flooding the system with a large number of messages.

1.3.6. Buffering of Messages

Consider the case when a process sends the value of variable *smesg* to some port. While the message is in transit, it needs to be kept in some memory area. It may be kept in the sender's address space (in the variable *smesg*), or may be **buffered** in an address space managed by the operating system such as the process table.

Keeping the data in the sender's address space is both time and space efficient. It does not require space for buffering and does not require copying to and from a buffer. However, it is not suitable if asynchronous sends are allowed (why?).

Several questions arise when data are buffered by the OS: What is the size of a message buffered by the OS? How many buffers are allocated? What happens when the buffer area gets exhausted? Are buffers allocated per port, per process, or shared by all processes and ports? There are no good general answers to these questions.

1.3.7. Pipes

One solution to some of the buffering problems of asynchronous send is to provide an intermediate degree of synchrony between pure synchronous and asynchronous. We can treat the set of message buffers as a ‘‘traditional bounded buffer’’ that blocks the sending process when there are no more buffers available. That is exactly the kind of message passing supported by Unix pipes. Pipes also allow the output of one process to become the input of another.

A pipe is like a file opened for reading and writing. Pipes are constructed by the service call *pipe*, which opens a new pipe and returns *two* descriptors for it, one for reading and another for writing. Reading a pipe advances the read buffer, and writing it advances the write buffer. The operating system may only wish to buffer a limited amount of data for each pipe, so an attempt to write to a full pipe may block the writer. Likewise, an attempt to read from an empty buffer will block the reader.

Though a pipe may have several readers and writers, it is really intended for one reader and writer. Pipes are used to unify input/output mechanisms and interprocess communication. Processes expect that they will have two descriptors when they start, one called ‘standard input’ and another called ‘standard output’. Typically, the first is a descriptor to the terminal open for input, and the second is a similar descriptor for output. However, the command interpreter, which starts most processes, can arrange for these descriptors to be different. If the standard output descriptor happens to be a file descriptor, the output of the process will go to the file, and not to the terminal. Similarly, the command interpreter can arrange for the standard output of one process to be one end of a pipe and for the other end of the pipe to be standard input for a second process. Thus a listing program can be piped to a sorting program which in turn directs its output to a file.

1.3.8. Selectivity of Receipt

A process may wish to receive information from a subset of the set of ports available for receipt. Systems differ in the amount of selectivity provided to a receiving process. Some allow a process to receive either from a particular port or all ports. Others allow a process to specify any subset of the set of all ports. Still others allow a process to peek at the value of incoming messages and make its decision to accept or reject messages based on the contents of messages.

Selection of a subset of ports is often specified as follows:

```
select
        receive <port> ...
        receive <port> ...
        ...
        receive <port> ...
end
```

(We are assuming RPC here, with synchronous send) If none of the ports has a message, then the process blocks. If several ports have messages, then one port is chosen non-deterministically. If only one port has a message, then the corresponding receive is executed. Typically, a *select* is enclosed within a loop. Thus after servicing a request, a process can service another request.

Often a system allows a **guard** to be attached to an arm of the **select**. The guard is a boolean expression and the associated *receive* occurs only if the condition is true. The following example shows guarded receives in a 'bounded buffer' process:

loop
    select
        when count > 0 **receive** consume (...) ...
        when count < size **receive** produce (...) ...
    end
end loop.

Here count keeps the number of filled buffers. The consuming process is blocked if the buffer is empty and the producing process is blocked if it is full.

Notice a process executing such a loop statement is similar to a monitor. Each receive in an arm of a select corresponds to an entry procedure declaration in a Monitor. A process services one receive at a time, just as a monitor executes one entry procedure at a time. The guards correspond to waits on conditions. The Lauer and Needham paper contains a more detailed discussion on this topic.

1.4. Xinu Low-Level Message Passing

Xinu offers two kinds of message passing primitives: one 'low-level', and the other 'high-level'. We now discuss the first kind. A discussion of the implementation of the second kind requires discussion of some of the memory management issues.

We first discuss the semantics of the low-level message passing and then its implementation.

1.4.1. Semantics

(1) Ports are input ports limited to one per process. Thus the process id of the receiver of the port is used as an identifier to this port.

(2) The information in a message is considered as data to be interpreted by the recipient. Thus RPC is not supported.

(3) The **send** operation takes as arguments the message and process id and delivers the message to the specified process. The operation is asynchronous, thus a sender does not wait till the message is received.

(4) The queue associated with the port holds at most one message. Moreover, a message size is restricted to 1 word. Thus memory allocated to buffer and queue messages is kept under control.

(5) Two kinds of receives are provided. The first called **receive** is a synchronous receive. It blocks the receiver till a message arrives. The second, called **recvclr** is non-blocking. If the process has a message when it calls *recvclr*, the call returns the message. Otherwise it returns the value *OK* to the caller without delaying to wait for a message to arrive. Thus *recvclr* allows a process to *poll* the system for message arrival. It is useful when the process does not know whether a message will arrive or not and does not want to block forever in case a message does not arrive. An example of its use: if software interrupts are not allowed, a process may use *recvclr* to check, every now and then, if the user has hit the "attention" key.

(6) Communicating processes reside within the same computer, so the message passing is trivially reliable and ordered. A *reply* operation is not supported. Since a process is limited to one input port, the issue of selective receipt does not arise.

1.4.2. Implementation

The Receive State

A process waiting for a message is put in the *receive* state. The arrival of a message moves it to the *ready* state.

Send

(1) It checks to see if the specified recipient process has a message outstanding or not. If there is an outstanding message, the *send* does nothing and returns (why?). Otherwise, it buffers the message and sets a field to indicate that now there is an outstanding message. Finally, if the receiving process is waiting for a message, it moves the process to the ready list (and calls *resched*), enabling the receiving process to access the message and continue execution.

(2) A message is buffered in a field in the process table entry of the receiving process. (Could it be stored in the process table entry of the sending process?)

Receive

(3) It checks to see if there is a message waiting. If not, it moves the process to the *receive* state (does the process need to be moved to any list?), and calls *reschedule*.

(4) It then picks up the message, changes a process table entry field indicating there are no outstanding messages, and returns the message.

Recvclr

It checks if a message has arrived. If no, it returns *OK*. If yes, it picks the message, changes appropriate fields, and returns the message.

2. Communication across a Network

Our implementation has assumed intra-machine IPC. Distributed systems require communication between remote machines. For instance a long-haul network that supports remote file copy requires communication between the remote and local process involved in the file copy, a local-area network that supports file servers needs to support communication between a client and a file server, and a multicomputer OS needs to be able to support communication between arbitrary processes running on different machines.

One of the important concepts behind network communication is **layering**. The hardware provides the most primitive layer of network communication. Layers on top embellish this communication. We will look at both the hardware (or physical) layer and embellishments to it.

2.1. The Physical Layer

In this layer we study the low-level hardware primitives for network communication. This communication may be **circuit switched** or **packet-switched**. Circuit-switched networks operate by forming a dedicated connection (circuit) between the two points. While a circuit is in place, no other communication can take place between the channels involved in the communication. The US telephone system uses such communication between two telephones. Packet-switched networks operate by dividing the conversation between two parties into **packets**, and multiplexing the packets of different conversations onto the communication channels. A packet, typically, contains only a few hundred bytes of data that includes header information that identifies the sender and receiver.

Each approach has its advantages. Circuit switching guarantees a constant communication rate: once the circuit is established, no other network activity will decrease the communication rate. This is particularly important for audio/video communication. One disadvantage is throughput: no one else can use the circuit during a 'lull' in the conversation between the two parties. Therefore circuit switching is not a very popular method for computer communication and packet switching is always preferred since it provides better utilization of the channel bandwidth, which is specially important for asynchronous transfer of bulk data. Since a channel bandwidth, typically, is fairly high, sharing of its does not present many problems, specially for traditional data transfer applications. For multimedia communication, packet-switched networks have been designed that try to reserve part of the network bandwith for some communication channels. In the remainder of the discussion we shall assume packet switching.

2.2. Network Topologies and Access Control Protocols

An important issue in the design of a network is the network topology. A popular network toplogy is the **bus topology**. Under this topology, the communicating devices are connected to a common bus, and packets contain addresses of the receiving devices. Thus, while a packet is available to all devices, only the addressed device actually receives it.

Since all devices share a common bus, we need a protocol to ensure that devices to do not corrupt packets by simultaneously transmitting them on the bus. One such protocol is the **CSMA/CD** (Carrier Sense Multiple Access with Collision Detection) protocol. Under this protocol, a sending device listens to the bus to detect another concurrent transmission. In case of collision, it backs off for a random period of time before trying again, and sends a jamming signal to all other devices asking them to ignore the packet. The backoff period increases exponentially with each successive unsuccessful attempt at sending a message.

An alternative access protocol is the **token ring protocol**, which prevents rather than cures collisions. It arranges the devices in a *logical* ring and allows a unique token to be passed along this ring from device to device. Only the token holder is allowed to send messages. Token passing is implemented as a special control packet.

One can also arrange the devices *physically* in a **token ring topology**, which supports point-to-point transmission rather than broadcast of messages. At each point, a device either consumes the packet or forwards it to the next point.

This topology can, of course, use the token ring protocol for controlling access. This protocol is implemented under this topology by continuously keeping a physical token in circulation, which may be free or busy. A sending device marks a free token as busy before sending the message and marks it as free the next time it arrives at that site after the transmission is completed.

It can also use the **slotted ring protocol**, which circulates, instead of a single token, a number of fixed length slots, which may be busy or free. A sending device waits for a slot marked free, sends the message in it, and marks it as busy. To ensure than one device does not hog the network, a device can use only one slot at a time.

2.2.1. The Ethernet

We now look very briefly at the **Ethernet**, a local-area packet-switched network technology invented at Xerox PARC in the early 1970s that uses the bus topology with the CSMA/CD protocol. The channel used for communication is a coaxial cable called the **ether**, whose bandwidth is 10 Mbps and maximum total length 1500 meters. Each connection to the ether has two major components. A **transceiver** connects directly to the ether cable, sensing and sending signals to the ether. A **host interface** connects to the transceiver and communicates (as a device) with the computer (usually through the computer bus).

Each host interface attached to a computer is assigned a 48 bit **Ethernet address** also called a **physical address**. This address is used to direct communication to a particular machine. Vendors purchase blocks of physical addresses and assign them in sequence as they manufacture Ethernet interface hardware.

Each packet transmitted along an ether is called an Ethernet **frame**. A frame is a maximum of 1536 bytes and contains:

(1) a preamble (64 bits or 8 octets), used for synchronization,

(2) source and destination addresses (6 octets),

(3) packet type (2 octets), used by higher layers,

(4) data (46 to 1522 octets), and

(5) Cyclic Redundancy Code (CRC) (4 octets), which is a function of the data in the frame and is computed by both the sender and the receiver.

The first three fields of a frame form its **header**.

A destination address in the packet may be the physical address of a single machine, a **multicast** address of a group of nodes in the network, or the network **broadcast** address (usually all 1's). A host interface picks up a frame if the destination address is:

(1) the physical address of the interface

(2) a multicast address of the group to which the host belongs, or

(3) one of the alternate addresses specified by the operating system.

2.3. Internetworking

So far we have seen how packets travel between machines on one network. Internetworking addresses transmission of packets between machines on different networks.

Communication of data between computers on different networks requires machines that connect (directly or indirectly) to both networks that are willing to shuffle packets from one network to another. Such machines are called **bridges**, **gateways** or **routers**. As an example, the machine `ciscokid.oit.unc.edu` serves as a router between our departmental FDDI network and the rest of the world. Similarly, the machine `mercury.cs.unc.edu` serves as a bridge between an Ethernet subnet and the main departmental FDDI backbone. Look at `http://www.cs.unc.edu/compServ/network/current.html` for our current network configuration. Not every machine connected to two networks is a gateway. For instance, the machine `jeffay.cs.unc.edu` is connected to connected to both an ethernet and the FDDI backbone, but is not a gateway.

2.3.1. Internet Addresses

Internetworking requires a network-independent communication paradigm. One such paradigm is illustrated by the Internet. Each machine connected to the Internet is assigned a 32-bit internet address that serves as its universal identifier. This address is a triple (*adrKind*, *netid*, *hostid*), where *adrKind* identifies the kind of address (*A*, *B*, and *C*), *netid* identifies a network, and *hostid* identifies a host on that network. The number of bits devoted to each element depends on the kind of address:

(1) Class *A* addresses, which are used for the few networks like the Internet backbone that have a large number of hosts, have the 1st bit set to ‘0’, next 7 bits allocated for the *netid*, and the next 24 bits for *hostid*.

(2) Class *B* addresses, which are used for intermediate sized networks, have the first two bits set to ‘10’, the next 14 bits allocated for *netid*, and the next 16 bits for *hostid*.

(3) Class *C* addresses, which are used for small sized networks like Ethernets, have the first two bits set to ‘11’, the next 22 bits allocated for *netid*, and the remaining 8 bits for *hostid*.

Internet addresses are usually written as four decimal integers separated by decimal points, where each integer gives one octet of the internet address. Thus the 32 bit internet address:

```
10000000 00001010 00000010 00011110
```

is written

```
128.10.2.30
```

Note that a host connected to more than one network has more than one internet address. For instance the internet addresses of *ciscokid.oit.unc.edu* look like:

```
152.2.254.254 (for the departmental network)
192.101.24.38 (for a network that connects to Greensboro).
```

The internet addresses are assigned by a central authority. By convention, an internet address with host set to zero is considered to be an address of the network, and not a particular host. Thus the address of the local FDDI backbone is

```
152.2.254.0
```

(Look at the file `/etc/hosts` and `/etc/networks` for internet addresses of different computers and networks. Execute `/usr/local/bin/nslookup` to find the IP address of a machine from its name. Execute `/usr/local/etc/traceroute` to find the path a message takes from the local machine to a remote machine specified as an argument.) Also the hostid consisting of all 1’s indicates a broadcast.

The internet is implemented as a layer above the physical layer. It provides its own unit of transfer called a **datagram**. Like an Ethernet frame, a datagram is divided into a header and data area. Like the frame header, a datagram header contains the source and destination addresses, which are internet addresses instead of physical addresses.

A datagram needs to be embedded in the physical unit of transfer. For instance a datagram transmitted on an Ethernet is embedded in the data area of an Ethernet frame. (Thus the data area of an Ethernet frame is divided into the header and data areas of the Internet datagram)

How are internet addresses mapped to physical addresses? We first consider communication between hosts on the same network. Later we will consider communication that spans networks.

2.3.2. Communication within a Network

One approach is to make each machine on the network maintain a table that maps internet addresses to physical addresses. A machine that needs to send a datagram to another machine on the network consults the table to find the destination physical address, and fills this address in the physical header.

A disadvantage of this approach is that if the physical address of a host changes (due to replacement of a hardware interface, for instance) tables of all hosts need to be changed. Therefore the DARPA Internet uses a different approach called **ARP** (for Address Resolution Protocol). The idea is very simple: When machine *A* wants to send a datagram to machine *B*, it broadcasts a packet that supplies the internet address of *B*, and requests *B's* physical address. Host *B* receives the request and sends a reply that contains its physical address. When *A* receives the reply, it uses the physical address to send the datagram directly to *B*.

Each host maintains a cache of recently acquired internet-physical address bindings. This cache is looked up before an ARP packet is sent.

In the above discussion we have assumed that node *B* knows its internet address. How does a node know its internet address? Machines connected to secondary storage can keep the address on a local file, which the operating system reads at startup. Diskless machines however support only remote files, to access which they need to communicate with some file server. However, they do not know the internet address of the file server either.

One approach, illustrated by the **Reverse Address Translation Protocol (RARP)**, is for a diskless machine to broadcast a packet requesting its internet address. Some server, that serves such requests, locates the internet address of the machine and sends back a message containing the internet address of the requester.

2.3.3. Indirect Routing

Now assume that *A* and *B* are on different networks *N1* and *N2*. In this situation *A* needs to identify some gateway on *N1* that can deliver the packet on *N2*, and sends the datagram (using ARP) to that gateway. The gateway, when it receives the message either delivers it directly to the host or forwards it to some other gateway, depending on the internet address of the destination.

How does a node find a gateway for a network? It maintains an **internet routing table**, that consists of pairs (N, G), where *N* is an internet network address and *G* is an internet gateway address. This table is updated dynamically to optimize the routing.

2.4. Process to Process Communication: UDP and TCP/IP

So far we have seen how arbitrary hosts communicate with each other. How do arbitrary processes on different machines communicate with each other?

One approach to such communication is illustrated by the User Datagram Protocol (UDP) which is a layer above the Internet. The unit of transfer in this layer is the **UDP datagram**, and the destination is an input within a host. Thus, the destination of a message is specified as the pair (host, port). A UDP datagram is embedded in the data field of the Internet datagram, and contains its own header and data areas. The UDP header identifies the destination port and a reply port. Appropriate software distributes the datagrams reaching a host onto the queues of appropriate ports.

UDP provides unreliable delivery: datagrams may be lost due to electrical interference, congestion, or physical disconnection. Often processes require a communication protocol that provides reliable delivery. One such protocol built on top of IP is TCP (for Transmission Control Protocol). TCP/IP supports end-to-end stream communication: a stream is established by connecting to it and terminated by closing it. To support reliable delivery, each packet is acknowledged. Should the acknowledgement also be acknowledged? If so, what about the ack of the ack of the ack, an so on...? The answer is that the ack is not acked. Instead, if the sender does not send the ack within some transmission period, T, it retransmits the packet, and repeats the process till it gets the ack. This process is expected but not guaranteed to terminate as long as the remote machine/network is not down. After a certain number of tries, the sender gives up and closes the connection.

Retransmission can result in duplicate packets being received at the sending site. To distinguish between normal packets and duplicates, packets are associated with sequence numbers. A packet with a sequence number less than or equal to the last one received successfully is discarded as a duplicate. However, its ack is resent since the ack for the original packet may have been lost.

The system allows a sequence of packets within a *sliding window* to have outstanding acknowledgements. This approach increases the concurrency in the system since packets and acks can be travelling simultaneously on the system. It also reduces the number of messages in the system since an ack for a packet simply indicates the next expected sequence number, thereby implicitly acking for all messages with a lower sequence number. With each ack, the transmission window slides forward past the message acked, hence the term sliding window. TCP/IP allows the size of the sliding window to vary depending on how much buffer space the receiver has. TCP/IP connections are two-way and an ack for a packet received from a machine can be piggybacked on a message sent to that machine.

2.5. OS Interface: Sockets

We have seen earlier abstract ideas behind OS-supported message passing, which can be implemented on the network layers mentioned above. To show how an OS layer sits on top of these network layers, let us consider the concrete example of Unix sockets. Unix sockets provide an interface to UDP and TCP/IP and other protocols at this level. Unlike IP ports, sockets support a combination of free, input, and bound ports. Socket declarations are given in the file `<sys/socket.h>.` A server providing a service first creates a socket that serves as an input port:

```
service_sock = socket (af, type, protocol)
```

where *af* indicates address family or name space (AF_INET - (host, port), AF_Unix - Unix file system, AF_APPLETALK), *type* indicates type of connection (SOCK_DGRAM, SOCK_STREAM) and *protocol* indicates the kind of protocol to be used (IPPROTO_IP - System Picks, IPPROTO_TCP, ...) After this the server does the following:

```
bind (service_socket, local_addr, addr_len)
listen (service_socket, qlength) - stream
server_end = accept (service_socket, remote_addr, remote_addr_len)  - stream only
write (server_end, msg, len)
or
send (server_end, msg, length, flags)
or
sendto (server_end, msg, length, flags, dest_addr, addr_len) - datagram only

read (server_end, &msg, len)
or
recv (server_end, &msg, len, flags)
or
recvfrom (server_end, msg, length, flags, &from_addr, &from_addr_len) - datagram only
```

and the receiver can similarly execute

```
client_connection_sock = socket (af, type, protocol)
bind (client_connection_socket, local_addr, addr_len) - stream optional
connect (client_connection_sock, dest_addr, dest_addr_len)
read, write, send, receive, sendto, recvfrom
```

The `bind` call binds the socket to a local address. The structure of local addresses is given in the file `<netinet/in.h>.` The address indicates an internet address and port number. The port number is a 16 bit number the server chooses. Port numbers 0..1023 are reserved by system servers such as `ftp` and `telnet.` Look at the file `/etc/services` for port number of system servers. A user-defined server, must choose an unbound port number greater than 1023. You can explicitly pick one of these values and hope no other process is using the port. A port number of 0 in the port field indicates that the system should choose the next unbound port number, which can then be read back by the server using `getsockname.` When communicating internet addresses, port numbers, and other data among machines with different byte orders, you should use routines (such as `htons, htonl,` and `ntohs`) that convert between network and host byte orders for shorts/longs. A bind call can use a special constant (htonl (INADDR_ANY)) for an internet address to indicate that it will listen on any of the internet addresses of the local host.

To talk to a server, a client needs to create a connection endpoint through its own socket and bind calls. Then it can use `connect` to link its socket with the server socket. If it does not have the internet address of the host, it can determine this address from the host name using `gethostbyname.`

A socket bound by the server serves as an ‘‘input port’’ to which multiple clients can connect. The connect call creates a new ‘‘bound port’’ for the server-client connection. The client end of the bound port is the socket connected by the client. The other end is returned to the server by the `accept` call, when a successful connection is established by the client. The server determines the number of connections to a bound socket using the `listen` call. For UDP, no connections need to be established through `accept` and `connect`- the `connect` call can be invoked but it is a local operation simply storing the intended remote address with the socket. In the stream case, the client usually does not bind its end of the socket to a local port, the system automatically binds the socket to an anonymous port. Sockets are not strictly input or bound ports since they can be inherited and accessed by children of the process that created them (through the mechanism of file descriptor inheritance we shall see later).

Data can be send/received using either the regular `read` and `write` calls, or the special `send` and `recv` calls, which take additional message-specific arguments such as send "out of band" data. The `sendto` and `recvfrom` calls are applicable only for UDP datagram communication and are necessary if the client has not "connected" the socket to a remote address.

2.6. Sun RPC

Both UDP and TCP require a client to encode its request and parameters in the data area of the message, and then wait for a reply on the reply port of the message. The server in turn needs to decode the request, perform the service, and then encode the reply. It would be more useful if the client could directly call a remote procedure which the server executes to service the request.

Sun RPC demonstrates how a remote procedure call paradigm can be supported on top of UDP or TCP via library routines. Each server registers a procedure pair via the `registerrpc` call,

```
registerrpc (prognum, versnum, procnum, procaddr, inproc, outproc)
```

A client can then invoke `callrpc` to call the remote procedure:

```
callrpc (host, prognum, versum, procnum, arg, inproc, outproc)
```

The procedure `callrpc` takes as arguments the machine number of the server, the procedure identifier, and the input and output parameters. It sends an appropriate UDP or TCP message to the destination machine, waits for a reply, extracts the results, and returns them in the output parameters.

The XDR (eXternal Data Routines) are responsible for marshalling/unmarshalling procedure parameters onto/from XDR streams. Example of XDR (input & output) procedure :

```
typedef struct {
    int f1;
    float f2 } S;

xdr_s (xdrsp, arg)
    XDRS *xdrsp;
      S *arg;
{
      xdr_int (xdrsp, &arg->f1);
      xdr_float (xdrsp, &arg->f2);
}
```

It is called in the marshalling/unmarshalling modes by the client and server RPC system. In the marshalling mode, it writes the argument onto the stream and in the unmarshalling mode it extracts data from the stream into the argument.

2.7. Transparent RPC

Sun remote "procedure call" is different from a local procedure call. Some systems, such as the Xerox Cedar language, try to support **transparent RPC** which looks and behaves like an ordinary procedure call. Such RPC differs from Sun RPC in two main ways:

(1) Since a remote procedure call looks exactly like a local procedure call, it does not explicitly indicate the location of the remote machine. The remote machine is determined by a special binding phase, which occurs before the call is made.

(2) Programmers do not explicitly marshall parameters or unmarshall results. The marshalling and unmarshalling (XDR) routines are generated from the declarations of procedures invoked remotely. For this reason, systems

that support such RPC are sometimes also called *RPC generators*.

Let us fill some of the details behind the concept of transparent RPC system. Such a system expects procedure headers and the types of the procedure parameters to be encapsulated in **interfaces**. An interface is implemented by a server and used by a client to invoke the procedures defined in the interface. An interface can be compiled for remote invocation. Before we look at the implementation of transparent RPC, let us resolve some semantic issues.

2.7.1. Semantics

Reference Parameters

An important difference between a local and remote procedure call has to do with address parameters. Since addresses in one process have no meaning in another (unless we have shared memory), some systems disallow address parameters while others create an isomorphic copy of the data structure pointed to by the actual parameter at the receiving process and point the formal parameter at this copy. The latter systems support transparency only for languages that do not allow address arithmetic.

Binding

How does a client machine identify the server? We can associate each interface definition with a global type name, T. Also we can associate each implementation of T with a global instance name, I. An instance is implemented by some process P on some host H. There is a spectrum of binding strategies possible, based on the **binding awareness** in the client program:

(1) One extreme approach is for the client program to simply identify which interfaces types it is using. The server RPC system publishes the interfaces implemented by each server, giving the location (host, process, and interface id) of the implementation. The client RPC system chooses one of these published implementations for the interfaces used by the client. This is the minimum amount of binding awareness possible since the interface type is necessary to link the client to appropriate generated routines.

(2) The other extreme is for the client to indicate not only the type of the interface but also the complete location of the implementation in which it is interested.

(3) Intermediate choices are to specify some but not all the details of the location, letting the system figure out the unspecified details. In particular, a client program can specify I or (I, H).

Less binding awareness makes the program more portable and easier to write. On the other hand, it gives the client programmer less control and also makes the binding less efficient since the system must maintain and search (centralised or replicated) information about published interfaces.

No of Invocations

How many times has the remote procedure call executed when it returns to the invoker? Ideally, we would want to maintain the semantics of local procedure call, which is guaranteed to have executed execute exactly once when it returns. However, these semantics are difficult to maintain in a distributed environment since messages may be lost and remote machines may crash. Different semantics have been proposed for number of remote invocations based on how much work the RPC system is willing to do:

(1) *At-least-once:* The call executes at least once as long as the server machine does not fail. These semantics require very little overhead and are easy to implement. The client machine continues to send call requests to the server machine until it gets an acknowledgement. If one or more acknowledgements are lost, the server may execute the call multiple times. This approach works only if the requested operation is **idempotent**, that is, multiple invocations of it return the same result. Servers that implement only idempotent operations must be **stateless**, that is, must not change global state in response to client requests. Thus, RPC systems that support these semantics rely on the design of stateless servers.

(2) *At-most-once:* The call executes at most once - either it does not execute at all or it executes exactly once depending on whether the server machine goes down. Unlike the previous semantics, these semantics require the detection of duplicate packets, but work for non-idempotent operations.

(3) *Exactly once*: The system guarantees the local semantics assuming that a server machine that crashes will eventually restart. It keeps track of **orphan calls**, that is, calls on server machines that have crashed, and allows them to be later adopted by a new server. These semantics and their implementation were proposed in

Bruce Nelson's thesis, but because of the complexity of the implementation, were never implemented as far as I know. Nelson joined Xerox where he implemented the weaker at-most-once semantics in the Cedar environment.

How should the caller be told of RPC failures in the case of at-least once or at-most-once semantics? We cannot return a special status in case of transparent RPC since local procedure calls do not return such values. One approach, used in Cedar, is to raise a host failure exception, which makes the client program network aware even though the call syntax is transparent.

2.7.2. Implementation

There are several basic components that work together to allow a client to make a distributed invocation in a server that implements a remote interface:

Client Code

Client code, C, written by the client programmer, that makes the remote invocation. For instance, a call of the form:

```
i : = P (s)
```

where P is a procedure defined in some remote interface.

Server Code

The *server code*, S, written by the server programmer, that implements the remote procedure. For instance a procedure of the form:

```
procedure P (s: S): int {
   /* implement the functionality of P */
    ...
 return (result)
}
```

Client Stub

A *client stub*, for each remote procedure P, generated by the interface compiler, that is linked to the client code and is called by C when it makes the remote invocation. C cannot call the remote P directly since they are in separately linked address spaces. Therefore, what it actually does is call the client stub, which marshalls the parameters, talks to the RPC runtime to send a remote message, receives a return message from the runtime, unmarshalls the results, and returns it to C. Thus, the form of the stub is:

```
procedure P (s: S): int {
  /*  marshall s */
   xdr_s (xdrsp, s)
  /* send  message via RPC runtime */
   ...
  /* receive message from RPC runtime */
   ...
  /* unmarshall result */
   xdr_int (xdrsp, result);
 return (result)
}
```

Server Stub

A *server stub*, for each each remote procedure P, generated by the interface compiler, that is linked to the server code, S, and invokes the implementation of P. It is the dual of the client stub - It unmarshalls the parameters and marshalls the results. Thus, its form is:

```
procedure PServerStub {
    /* unmarshall s */
    xdr_s (xdrsp, s);
    /* call P */
    result <- P (s);
    /* marshall result */
    xdr_int (xdrsp, result);
    /* send result via RPC runtime */
    ...
}
```

Server Dispatcher

A *dispatcher*, generated by the RPC system, and linked to the server, which receives an incoming call request and invokes the corresponding server stub. The call request identifies the procedure to be invoked and the dispatcher is responsible for mapping it to the server stub. This mapping is generated at interface compilation time.

XDR Routines

The *XDR routines* for predefined (simple) types are written by the implementer of the RPC system, while the routines for user-defined types are generated by the compiler based on the definitions of the types of the arguments of the remote procedures.

RPC Runtime

The *RPC runtime*, which exists at both the client and server sites. It is responsible for sending and receiving messages from remote sites. It is also responsible for binding remote calls. It can, like Sun RPC, simply use a general, lower-level networking layer such as UDP or TCP/IP to send and receive messages. However, it is possible to define efficient, specialised communication protocols for implementing transparent RPC, as illustrated by Cedar.

Specialised Protocols

The Cedar system uses a special network protocol to support at-most-once RPC semantics. Such semantics can be supported on top of a connection-based reliable protocol such as TCP/IP. However, they are not optimal for RPC mechanisms, since they have been designed to support asynchronous reliable transfer of bulk data such as files. There are two possible optimisations possible in a specialised protocol:

(1) *Implicit Sessions*: A bulk data protocol requires explicit opening and closing of sessions. An RPC implementation on top of such a protocol can use two main approaches to opening/closing sessions: First, a client machine can open a session with each possible server machine with which the client may communicate. Second, the client machine can open/close the connection before/after each call. The first approach amortizes the cost of opening/closing connections over all calls to a server machine, but uses more connections (which are scarce resources) at any one time and requires probe messages inquiring the status of a server machine to be sent even when no RPC is active. The second approach, on the other hand, requires connections to be active and pinged only during the procedure call but requires an explicit opening/closing per call. A specialised protocol can offer the advantages of both approaches by implicitly opening/closing a connection at the start/termination of a call.

(2) *Implicit Acknowledgement:* A bulk-data protocol can result in each packet being acknowledged. In a specialised protocol, the RPC reply can acknowledge the last packet. This can result in no acknowledgments being sent for RPC calls with small arguments.

The Cedar implementation shows how a specialised protocol may be implemented. Each call is assigned an id from an increasing sequence of ids generated by the system. The RPC runtime at the client machine sends to the receiver the following client information: *call id, dispatcher id, procedure id,* and *arguments.* The RPC runtime at the server machine sends back to the client: *call id* and *results.* The client runtime breaks the client information into one or more packets, encloses the packet number, and asks for an acknowledgement for all packets except the last one, which is acknowledged by the reply. After each packet, it waits for an ack/reply. If it does not receive the ack/reply within a certain time, T, it resends the packet, asking for an explicit ack (even for the last packet). After receiving all acks, the client may need to wait for the RPC to finish. It periodically sends probe packets to check the status of the server machine. If it does not receive acks to normal/probe to packets after a certain number, R, of retransmissions, it determines that the server machine has crashed and reports an exception to the client program. Once it receives a reply, it waits for a certain period of time, D, for another call to the server. If a call is made within that period, then that call serves as an ack for the reply of the previous call. Otherwise, the client explicitly sends an ack. (In class I assumed a different algorithm wherein the ack to a probe message may send the status (in case the call is taking a long time) or the return value (in case the reply was lost) - in fact Cedar uses the simpler scheme described above.)

Under this protocol, short calls (duration $<$ T) with small arguments (fitting in one packet) and occurring frequently (inter call delay $<$ D) resulting in no lost messages require only two messages to be communicated. A long call (not requiring probe packets) requires an additional retransmission of the argument and its explicit call. A really long call requires transmissions and acks of probe packets. A call that is not followed quickly by another call requires an ack for the reply.

Thus, this procotol is optimised for the first case: it is not possible to do better in this case. A bulk transfer protocol would require additional open and close messages and an additional argument ack unless it is piggybacked on the reply message. On the other hand, this protocol may result in more messages to be communicated for other cases since it makes a client wait for an ack for a previous packet before sending the next one. As a result, the server machine must ack all packets except the last one. A bulk data transfer protocol allows multiple packets to have outstanding acks and allows one server message to serve as an ack for a sequence of client messages. Thus, this protocol follows the principle of making the usual case efficient while making the unusual cases possible but not necessarily efficient.