Lectures on distributed systems:

Client-server communication

Paul Krzyzanowski

Everything lives, moves, everything corresponds; the magnetic rays, emanating either from myself or from others, cross the limitless chain of created things unimpeded; it is a transparent network that covers the world, and its slender threads communicate themselves by degrees to the planets and stars. Captive now upon earth, I commune with the chorus of the stars who share in my joys and sorrows.

—Gérard de Nerval, Aurélia,
Part 2, chapter 6

Networking

IF WE ARE TO BUILD DISTRIBUTED SYSTEMS, we will have an environment where independent machines will be working cooperatively with each other *without* shared memory. To work together, they will have to communicate with each other. This is where the interconnect – the network – comes in.

Modes of connection

Communication over a network can be classified into two types:

circuit-switched
In this network, a dedicated channel exists to the remote machine. An example is that of a telephone network: when you place a call, a dedicated circuit is established for you to the destination. In a circuit-switched network, you are guaranteed to have access to the full bandwidth of the circuit.

packet-switched
Connections are shared in this type of network. Data that is transported across the network is broken up into chunks called *packets*. Because packets from different sources (and to different destinations) are now intermixed, each packet must contain the address of the destination (in a circuit-switched network, this isn't needed since the circuit is a dedicated connection). An ethernet network is an example of this type of network. In a packet-switched network, the bandwidth that you see will usually be less than the capacity of the network since you're sharing the channel with others.

Parlez-vous français? ¡Sí, muy bien!

For computers (or people, for that matter) to be able to communicate, they must speak the same language and follow the same conventions. For humans, this means speaking the same language and knowing how low to bow, which hand gestures not to use, and whether it is acceptable to excrete gas. For computers, this requires knowing how to find out how long a packet is that is coming over a network (so we can get it), knowing where the destination address is stored, and being able to properly interpret all the data within it. The issue is non-trivial because different computers have different concepts of which order to store bytes of a word, how long an integer is, and what character set is being used. The instructions and conventions needed for successful communication is known as a **protocol**.

To ease the task of communicating and provide a degree of flexibility, network protocols are generally organized in layers. This allows you to replace a layer of the protocol without having to replace the surrounding layers. It saves higher-level software from having to bother with formatting an ethernet packet. The most popular model of guiding (not specifying) protocol layering is the *OSI Reference Model*, designed in 1977 and refined somewhat thereafter. It contains seven layers of protocols:

Layer	*Name*	*Function*
7	application	the protocol of applications using networking, such as file transfer, directory services, distributed processing applications, and many others.
6	presentation	responsible for the selection of an agreed-upon syntax (data representation). This layer may have to convert data between the agreed-upon representation and the machine's native types.
5	session	responsible for connection establishment, data transfer, and for connection release. It tracks who initiated a conversation and may manage the re-establishing of a logical communication channel.
4	transport	provides reliable end-to-end communications by providing service-level (transport) addressing, flow control, datagram segmentation, and end-to-end error checking. It can ensure that packets appear to arrive in the correct order and issue retransmission requests to ensure the reliable message delivery.
3	network	relay and route information to the destination. This layer is responsible for managing the journey of packets between local area networks and figuring out intermediate hops (if needed).
2	data link	provides the first level of organization of data –

		the datalink frame, which includes source address, destination address, content, and some form of checksum for error detection. includes Media Access Control (MAC) and Logical Link Control (LLC). MAC covers rules for accessing the media and dealing with contention. The LLC portion covers frame synchronization, flow control, and error checking.
1	physical	deals with the specification of the data signals, voltage levels, transmission speed, and connectors.

Some networking terminology

This section will provide a whirlwind tour of some of the more commonly encountered terms encountered in networking.

A **local area network** (**LAN**) is a communication network that covers a small area (a few rooms, a building, or a set of buildings), incorporates a shared transmission medium, and offers a relatively high data rate (typically 1 Mbps – 1 Gbps) with relatively low latency. The devices on a LAN are peers, so that any device can initiate a data transfer with any other device. Most elements on a LAN are **workstations**. This covers most any computing device, including PCs, Macs, Suns, etc. Workstations and other endpoints (devices) on a LAN are called **nodes**.

For a node to be connected to the LAN, interface hardware is needed. This is known as an **adapter** and is a circuit that usually sits on an expansion slot on a PCI or PCMCIA bus. Networking adapters are referred to as **Network Interface Cards**, or **NIC**s.

Media refers to the wires (or wireless RF) connecting together the devices that make up a LAN. The following types of media are generally encountered:

- **Twisted Pair** is the most common. It typically has eight wires and comes in two flavors: shielded twisted pair (STP) or the more common unshielded twisted pair (UTP). Telephone cable is an example of UTP.
- **Coaxial cable** (**coax**) comes in two flavors as well. Thin coax (similar to TV cable) is by far the more popular of the two. Thick coax is now largely obsolete. Thin coax is rarely seen in LANs as well.
- **Fiber**
- **Wireless**

A **hub** is a device that acts as a central point for LAN cable. A **switch** moves data from any input port to a specific destination port. **Concentrators** or **repeaters** regenerate data when passing through, allowing data to pass through longer distances. **Bridges** connect different LAN segments together (layer 2). R**outers** determine the next network point to which packets should be forwarded – they connect different types of local and wide area networks. Another frequently-encountered device is a **CSU/DSU** (Channel Service

Unit/Data Service Unit). This is a device that interfaces between a local area network and a wide area network (a leased data line). It converts a serial stream of data from a LAN to TDM (time-division multiplexed) frames on the data line (e.g., a T-1 line). This brings us to the **T-1**: the T-1 is the most commonly used digital line in the U.S., providing 24 channels with an aggregate data rate of 1.544 Mbps. A T-3 line is also frequently encountered and provides a transmission rate of 44.736 Mbps.

Ethernet is the most common networking technology. It was developed in the mid 1970's at Xerox PARC and standardized by the IEEE 802.3 committee. It is a **baseband** transmission network. This means that all nodes share access to the network media on an equal basis. Data uses the *entire* bandwidth of the media. This is opposed to **broadband** transmission, where a given data transmission uses only a segment of the media by dividing the media into channels. The typical speed of transmission on an Ethernet network is 100 Mbps, with speeds going up to 1 Gbps. Older Ethernet networks generally transmitted at 10 Mbps. Network access on an Ethernet is a mechanism called **Carrier Sense Multiple Access with Collision Detection** (**CSMA/CD**). To send data, a node first listens to the network to see if it is busy (i.e., someone else is sending data). When the network is not busy, the node will send data and then sense to see whether a collision occurred because some other node decided to transmit data concurrently. If a collision was detected, the data is retransmitted. The analogy of CSMA/CD is that of a party line on a telephone.

The original Ethernet media was thick coax, which was called 10Base5 because the maximum length of a cable run was 500 meters. Thin coax replaced thick coax, and is called 10Base2 (with a maximum run of 200 meters). Both 10base5 and 10Base2 cables were organized in a bus topology, with any number of nodes plugging into the same cable run. The most common Ethernet media now is 10BaseT, which is a twisted pair cable organized into a star topology (with a central hub). Each node has a dedicated cable that connects to the central hub (or switch).

Clients and servers

The most common networking relationship is the client-server model. The model contains three components: a client, a server, and a service. A **service** is that task that a machine can perform (such as offering files over a network or the ability to execute a command). A **server** is the machine that performs the task (the machine that offers the service). A **client** is the machine that is requesting the service. These titles are generally used in the context of a particular service rather than in labeling a machine: one machine's client may be another machine's server.

To offer a service, a server must get a *transport address* for a particular service. This is a well-defined location (similar to a telephone number) that will serve to identify the service. The server associates the service with this address before clients can communicate with it.

The client, wishing to obtain a service from the server, must obtain the transport address. There are several ways to do this: it may be hard-coded in an application or it may be found by consulting a database (similar to finding a number in a phone book). The database may be as simple as a single file on a machine (e.g. `/etc/services` on Unix systems) or as complex as accessing a distributed directory server.

We depend on transport providers to transmit data between machines. A **transport provider** is a piece of software that accepts a network message and sends it to a remote machine. There are two categories of transport protocols:

connection-oriented protocols - these are analogous to placing a phone call:

- first, you establish a connection (dial a phone number)
- possibly negotiate a protocol (decide which language to use)
- communicate
- terminate the connection (hang up)

This form of transport is known as **virtual circuit service** (it provides the illusion of having a dedicated circuit). Messages are guaranteed to arrive in order. When a true dedicated circuit is set up (as in a real telephone connection), then the service is not virtual, but true circuit switched service.

connectionless protocols - these are analogous to sending mail:

- there is no connection setup
- data is transmitted when ready (drop a letter in the mailbox)
- there's no termination because there was no call setup

This transport is known as **datagram** service. With this service, the client is not positive whether the message arrived at the destination. It's a cheaper but less reliable service than virtual circuit service.

Internet Protocol

By far the most popular network protocol these days is the family of Internet Protocols. The Internet was born in 1969 as a research network of four machines that was funded by the Department of Defense's Advanced Research Projects Agency (ARPA). The goal was to build an efficient, fault-tolerant network that could connect heterogeneous machines and link together separately connected networks. The network protocol is called the **Internet Protocol**, or **IP**. It is a connectionless protocol that is designed to handle the interconnection of a large number of local and wide area networks that comprise the Internet.

IP may route a packet from one physical network to another. Every machine on an IP network is assigned a unique 32-bit IP address. When an application sends data to a machine, it must address it with the IP address of that machine. The IP address is *not* the same as the machine address (e.g. the ethernet address) but strictly a logical address.

A 32-bit address can potentially support 2^{32}, or 4,294,967,296 addresses. If every machine on an IP network would receive an arbitrary IP address, then routers would need to keep a table of over four billion entries to know how to direct traffic throughout the Internet! To deal with this more sensibly, routing tables were designed so that one entry can match multiple addresses. To do this, a hierarchy of addressing was created so that machines that are physically close together (say, in the same organization) would share a common prefix of bits in the address. For instance, consider the two machines:

name	**address**	**address (in hex)**
cs.rutgers.edu	128.6.4.2	80 06 04 02
remus.rutgers.edu	128.6.13.3	80 06 0d 03

The first sixteen bits identify the entire set of machines within Rutgers University. Systems outside of Rutgers that encounter any destination IP address that begins with 0x8006 have only to know how to route those packets to some machine (router) within Rutgers that can take care of routing the exact address to the proper machine. This saves the outside world from keeping track of up to 65,536 (2^{16})machines within Rutgers.

An IP address consists of two parts:

- network number — identifies the network that the machine belongs to
- host number — identifies a machine on that network.

The network number is used to route the IP packet to the correct local area network. The host number is used to identify a specific machine once in that local area network. If we use a fixed 16-bit partition between network numbers and host numbers, we will be allowed to have a maximum of 65,536 (2^{16}) separate networks on the Internet, each with a maximum of 65, 536 hosts. The expectation, however, was that there would be a few big networks and many small ones. To support this, networks are divided into several classes. These classes allow the address space to be partitioned into a few big networks that can support many machines and many smaller networks that can support few machines. The first bits of an IP address identify the class of the network.

class	*leading bits*	*bits for network number*	*bits for host number*
A	0	7	24
B	10	14	16
C	110	21	8

An IP address is generally written as a sequence of four bytes in decimal separated by periods. For example, an IP address written as 135.250.68.43 translates into the hexadecimal address 87FA442B (135=0x87, 250=0xfa, etc.). In binary, this address is 1000 0111 1111 1010 0100 0100 0010 1011. The leading bits of this address are 10, which identifies the address as belonging to a class B network. The next 14 bits (00 0111 1111 1010) contain the network number (7FA) and the last 16 bits contain the host number (442B).

To allow organizations to create additional networks without requesting additional (and increasingly scarce) network numbers, some high bits of the host number may be allocated for a network number within a higher-level IP network. These local networks are known as *subnets*. Routers within an organization can be configured to extract this additional network ID and use it for routing. For example, a standard class B network allows 16 bits for a host number. This host address may be locally broken into 8 bits for a subnet ID followed by 8 bits for a host ID.

A machine that is connected to several physical networks will have several IP addresses, one for each network.

IP also supports several special addresses. They are the following:

all bits 0	(valid only as a source address) refers to "*all addresses for this machine*". This is *not* a valid address to send over the network but is used when offering a service to state that it is available to all networks that are connected to this machine (this will make sense when we look at sockets and binding)
all host bits 1	(valid only as a destination address) broadcast address: send to all machines on the network. An address 192.10.21.255 means "send the packet to all machines on network 192.10.21."
all bits 1	(valid only as a destination address) broadcast to all machines on every directly connected network.
all bits 1 on a class A network	refers to the local host. Sending data to this address causes it to loop back to the same machine. The typical address used is 127.0.0.1
Leading bits 1110	identifies a class D network. The remaining 28 bits identify a multicast group. A multicast packet is received by all members that are members of that multicast group. A machine may join or leave a multicast group at any time. This address is useful for teleconferencing and sending network video and audio.

Machines in an IP network are named in a hierarchical manner, with each level separated by a dot. Names to the left are lower in the hierarchy. For example, the name **`bescot.cl.cam.ac.uk`** identifies a machine named **`bescot`** in England (**`uk`**), under the Academia hierarchy (**`ac`**), within Cambridge University (**`cam`**), within the Computer Laboratory (**`cl`**). An IP address can only be found by looking it up in some database. In the past, the database was a single file (/etc/hosts) that contained the address of every machine. As the Internet grew, that file became difficult to manage. Now, in most cases, you would contact a name service offered by some machine that may in turn contact other

name servers until it finds a machine that knows the address for a given name. These name servers are known as *Domain Name Servers*, or DNS.

Running out of addresses

As the Internet expanded in the late 1980's and especially the early 1990's to include more networks and more hosts, the three-layer class hierarchy of IP addresses was getting stressed. More and more organizations wanted to be on the Internet. To be on the Internet, each machine would need an IP address. An organization would request an IP network address for a class that was sufficiently large to accommodate all of its machines.

The problem we were having with class-based Internet addressing was not in running out of IP addresses (exhausting all possible IP addresses) but rather running out of IP network addresses (of which there are only two million available). Many organizations also wanted more than a class C network and there are only a bit over 16,000 class A and B networks available.

While the class-based IP addressing scheme can accommodate close to four million distinct addresses, the problem was that the class-based network granularity was too coarse. For example, a class C network can support up to 254[1] host numbers. If an organization needed up to 1,000 hosts, it would need to request a far more precious class B network (of which there are only 16,382 such networks to allocate). The class B network will allow it to have up to 65,534 host numbers, meaning that over 64,000 IP addresses, or over 98% of the address space, will go unused.

To combat this problem, a routing structure called **Classless Inter-Domain Routing** (**CIDR**) was created. This is a structure that attempts to provide a better match between the range of addresses assigned to an organization and the number of addresses the organization really needs.

The practice of identifying a class of network (A, B, or C) by looking at the leading bits of the IP address was abandoned. Instead, an IP network number is defined to be a n arbitrary number[2] of leading bits of an address. Using the earlier example, if an organization needed to support 1,000 hosts, it would request a class B address, wasting over 64,000 addresses. Now it can request a 22-bit network number, which provides it with 10 bits of addressing for hosts, enough for 1,022 machines.

Since we can no longer look at the leading bits of an IP address and know how many of the following bits constitute the network number, each entry in the routing table now has to contain this number explicitly. A CIDR IP address includes the standard 32-bit IP addresses *and* information on the number of bits for the network prefix (for example, 128.6.13.3/16, where the 16 refers to a sixteen bit network number). The pitfall of CIDR

[1] We calculate this via $2^{(\text{host bits})}$-2, subtracting two to disallow addresses of all 1 bits and all 0 bits.
[2] Address registries may choose to limit the ranges of addresses they assign – a lot of tiny networks will yield overly large routing tables. The American Registry for Internet Numbers (ARIN) does not allocate prefixes longer than 20 or shorter than 13 bits. The European registry (RIPE) has 19 bits as its smallest allocation.

is that one now has to manage the prefixes as well as addresses. When routing tables are distributed, they must include the prefixes for the addresses to make sense.

CIDR requires router tables contain both an IP address and the number of bits for the network prefix. This structure helps alleviate another problem in IP networking: large global routing tables. With class-based IP addressing, every network had to have a routing entry in global routers on the Internet. This was both an administrative pain and a performance bottleneck. If network addresses can be assigned "sensibly", routing tables can be simplified. If adjacent network addresses are generally routed in the same way (for example, they belong to the same ISP and the routes split up only when they get to that ISP's network), then the global routing tables do not need to contain all those networks; they can simply specify that less bits are significant for the route (i.e., as far as the router is concerned it is a single route to the network with less bits being used for the network number).

Another innovation in IP addressing also helped alleviate the problem of assigning network addresses to organizations. The idea is that if every machine in an organization does not need to be addressed from the Internet, it need not have a unique IP address. Machines within an organization now can have internal IP addresses that are not unique across the Internet but only unique within the organization. Whenever a machine sends a packet outside the organization, it is routed through a gateway that will translate the address from an internal address to an external address. This will be the address of the gateway system (which must have a true external IP address). In translating the address, the gateway will keep a table of the original address and port number and the outgoing, translated, port number. When a return packet comes for that port number, the gateway identifies it as a response to the original packet and rewrites the destination in the IP header as the internal address and port number of the original sender. This scheme is known as **network address translation**, or **NAT**. The biggest benefit of NAT is that large organizations no longer need to request a network that can address thousands or tens of thousands of hosts. They only need to support the number of hosts that need to be visible from the Internet (e.g., those running services) as well as gateways.

How do you really *get to the machine?*

IP is a *logical* network that sits on top of multiple physical networks. Operating systems that support communication over IP have software that is called an IP driver. The IP driver is responsible for:

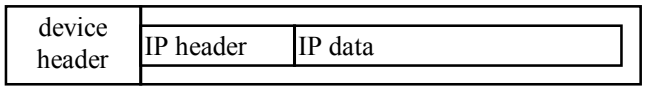
- getting operating parameters from the device driver (which controls the network card) such as maximum packet size, functions to initialize the hardware headers, and the length of the hardware header
- routing packets from one physical network to another
- fragmenting packets – it might have to send a packet that's too big for the network hardware to handle in which case it has to be split into several packets, each with its own IP header containing destination information

- performing send operations from higher level software
- receiving data from the device driver
- dropping data with bad checksums in the header
- dropping expired packets

The device driver for the network interface is responsible for controlling the network interface card. Its behavior is similar to that of character device drivers. It must:

- process interrupts from the network interface, receive packets, and send them up to the IP driver [bottom half]
- get packets from the IP driver and send them to the hardware, ensuring that the packet goes out without a collision (in a shared network such as an Ethernet, a packet may collide with another packet and not be sent) [top half]

The network device typically understands a much simpler interface. For example, an Ethernet device is addressed by a unique Ethernet address or a broadcast address. This address has no relation to the IP address of a machine. The hardware looks at all packets traveling down the wire and picks up those in which the destination address on the Ethernet header matches the address of the device. Before an IP packet can be sent, it has to be enveloped with the necessary information for the network device (such as the device address and length of packet).



The device (Ethernet) address can be found with a facility called **ARP**, **the Address Resolution Protocol**. ARP converts an IP address to a local device (for example, Ethernet) address by taking the following steps:

- check the local ARP cache
- send a broadcast message requesting the Ethernet address of a machine with a certain IP address
- wait for a response (with a time out period)

Routing

How does a packet find its way from here to there? A switching element is used to connect two or more transmission lines (e.g., Ethernet networks). This switching element is known as a **router**. It can be a dedicated piece of hardware or a general-purpose computer with multiple network interfaces. When it gets packet data, it has to decide to which line the data has to be sent. This job is called **routing**. When a router receives an IP packet, it checks the destination address. If the destination address matches that of the receiving system, then the packet is delivered locally. Otherwise, router uses the destination address to search a **routing table**. Each entry in the table has an address, the number of significant bits (usually represented by a bit mast known as a *netmask*), and an outgoing interface. When an entry is located that matches the IP destination address (not counting the bits outside the netmask), the packet can be sent out on the interface defined on that line.

This technique is known as **static routing**. An alternative to static routing is **dynamic routing**, which is a class of protocols by which machines can adjust routing tables to benefit from load changes and failures.

Protocols over IP

IP supports two transport layer protocols and one special protocol. The special protocol is called ICMP — the *Internet Control Message Protocol*. It is responsible for generating control messages and is datagram based. It sends a message to the originator whenever an IP packet is dropped and also generates advisory messages (such as "slow down" or "here's a better route"). The two transport layer protocols over IP are:

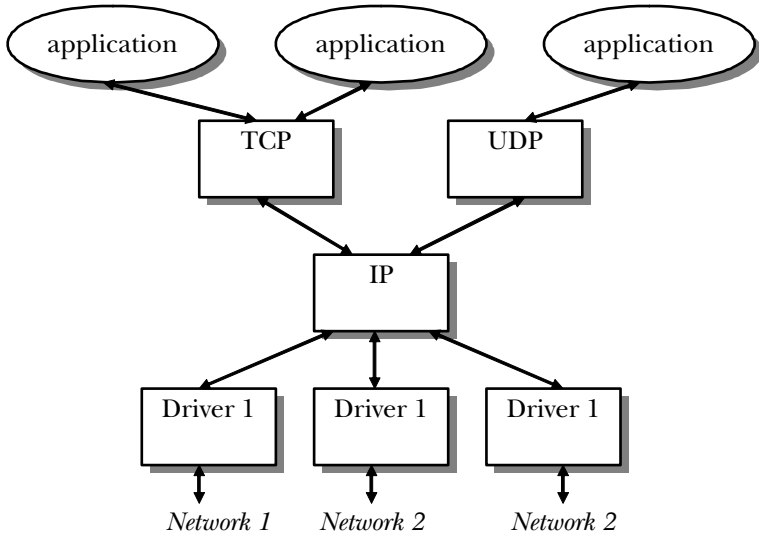
TCP — Transport Control Protocol

- virtual circuit service (connection-oriented)
- sends acknowledgment for each packet received
- checksum to validate data contents
- data may be transmitted simultaneously in both directions over a circuit
- no record markers (one write may have to be read with multiple reads) but data arrives in sequence

UDP — User Datagram Protocol

- datagram service (connectionless)
- data sent may be lost
- data may arrive out of sequence
- recipient's address must be specified in each request

Applications may use either of these protocols to send data over the network.



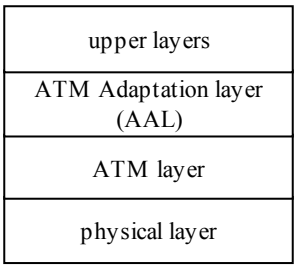
Asynchronous Transfer Mode (ATM) networks

One of the serious pitfalls of IP networking is that it has no mechanisms for an application to specify how the traffic that it generates is to be scheduled over the network. This causes problems in continuous media applications such as video and voice. In these applications, excessive delays in packet delivery can produce unacceptable results. IP version 6, the emerging IP standard, attempts to alleviate this somewhat by allowing applications to tag packets with a *priority* level, but this does not translate directly to bits per second. Another form of networking emerged in the late 1980's and was adopted as an international standard. This form is known as ATM, or Asynchronous Transfer Mode, networking. Its goal is to merge voice and data networking. The former is characterized by a low, but *constant* bandwidth. The later tends to be bursty in bandwidth requirements (0 one minute, 100 Mbps the next). Circuit switching is too costly for data networking since it is a waste of resources to allocate a fixed-bandwidth circuit to bursty traffic. IP-style packet switching, on the other hand, is not suitable for the constant bandwidth requirements for voice telephony.

ATM attacks the problem by using fixed-size packets over virtual circuits. A sender establishes a connection, specifying the bandwidth requirements and traffic type. Traffic type may be constant bandwidth rate (CBR), variable bandwidth with bounded delay (VBR), or available bandwidth (ABR)[3]. If the connection is accepted, a route is determined and routing information is stored in switches within the network. All traffic is carried in fixed-size cells. Fixed size cells provide for predictable scheduling (a large packet is not going to hold up smaller ones behind it) and rapid switching.

[3] Uncompressed voice and video traffic would require CBR scheduling. Compressed video or audio would provide parameters for minimum, average, and peak bandwidth, and request VBR service. Generic data services (e.g. telnet, web access) would request ABR service.

The current standard for an ATM cell is an inconvenient 53-byte size: 48 bytes for data and 5 bytes for the header. To avoid congesting a computer with millions of interrupts per second (one interrupt for each incoming packet), the ATM hardware often supports the splitting of larger chunks of data into multiple ATM cells and assembling incoming ATM cells into larger packets. This is called an *ATM adaptation layer* (AAL). A few adaptation layers exist for handling different traffic types (e.g. AAL 1 for constant bit rate traffic, AAL 2 for variable bit rate traffic, &c.). Perhaps the most popular for data networking is AAL 5. Outbound data is fragmented into multiple ATM cells with a bit set in a field in the last cell to indicate an end of packet. The destination, accepting the packet, simply assembles the cells coming in on that circuit until it gets a cell with an end of packet bit set. At that point, it can deliver the full data packet up to the upper layers of the system software. Compatibility with IP (useful for legacy applications) can be achieved by running the IP protocol over the ATM layer, segmenting and reassembling each IP packet into ATM cells.



ATM protocol stack

While ATM solves a number of problems present in IP/Ethernet networks, its switches and interface boards remain more expensive and Ethernet keeps getting faster, delaying the need for precise cell-level scheduling.

Accessing applications

We now know how data is sent and received between machines. How do we associate a connection (TCP) or data packets (UDP) with an application? Earlier we mentioned that a server must be able to get a ***transport address*** for a service and associate that address with the service. The client must be able to figure out this address and access the service through it.

One popular implementation is the concept of sockets, which were developed in Berkeley[4]. Sockets are an attempt at creating a generalized IPC model with the following set of goals:

- communication between processes should not depend on whether they are on the same machine
- efficiency: this should be an efficient layer on top of network communication facilities
- compatibility: processes that just read from a standard input file and write to a standard output file should continue to work in distributed environments
- must support different protocols and naming conventions (different "communication domains" or "address families")

[4] Another implementation that may be worth learning if you plan to do network programming is UNIX System V's Transport Layer Interface (TLI) and streams modules. In UNIX System V release 4, sockets are implemented as a library on top of streams.

The **socket** is an abstract object from which messages are sent and received. It is created in a communications domain roughly similar to a file being created in a file system. Sockets exist only as long as they are referenced. A socket allows an application to request a particular style of communication (virtual circuit, datagram, message-based, in-order delivery,…). Unrelated processes should be able to locate communication endpoints, so sockets should be *named*. The name is something that is meaningful within the communications domain.

Programming with sockets

There are several steps involved in creating a socket connection. As with the entire area of networking, this section cannot cover the entire topic fully. Several of the references listed provide more complete information. On-line manual pages will provide you with the latest information on acceptable parameters and functions. The interface described here is the system call interface provided by the Solaris operating system and is generally similar amongst all Unix systems (and many other operating systems).

1. Create a socket

A socket is created with the *socket* system call:

```
int s = socket(domain, type, protocol)
```

All the parameters as well as the return value are integers.

- `domain`, or *address family*—communication domain in which the socket should be created. Some of address families are AF_INET (IP family), AF_UNIX (local channel, similar to pipes), AF_NS (Xerox Network Systems protocols).
- `type`—type of service. This is selected according to the properties required by the application: `SOCK_STREAM` (virtual circuit service), `SOCK_DGRAM` (datagram service), `SOCK_RAW` (direct IP service). Check with your address family to see whether a particular service is available.
- `protocol`—indicate a specific protocol to use in supporting the sockets operation. This is useful in cases where some families may have more than one protocol to support a given type of service.

The return value is a file descriptor (a small integer). The analogy of creating a socket is that of requesting a telephone line from the phone company.

2. Name a socket

When we mention naming a socket, we are talking about assigning a transport address to the socket. This operation is called *binding an address*. The analogy is that of assigning a phone number to the line that you requested from the phone company in step 1 or that of assigning an address to a mailbox.

You can explicitly assign an address or allow the system to assign one. The address is defined in a socket address structure. Applications find addresses of well-known services

by looking up their names in a database (e.g., the file `/etc/services`). The system call for binding is:

```
int error = bind(s, addr, addrlen)
```

where *s* is the socket descriptor obtained in step 1, *addr* is the address structure (`struct sockaddr *`) and *addrlen* is an integer containing the address length. One may wonder why don't we name the socket when we create it. The reason is that in some domains it may be useful to have a socket without a name. Not forcing a name on a socket will make the operation more efficient. Also, some communication domains may require additional information before binding (such as selecting a grade of service).

3a. Connect to a socket (client)

For connection-based communication, the client initiates a connection with the *connect* system call:

```
int error = connect(s, serveraddr, serveraddrlen)
```

where *s* is the socket (type `int`) and *serveraddr* is a pointer to a structure containing the address of the server (`struct sockaddr *`). Since the structure may vary with different transports, connect also requires a parameter containing the size of this structure (*serveraddrlen*)

For connectionless service, the operating system will send datagrams and maintain an association between the socket and the remote address.

3b. Accept a connection (server)

For connection-based communication, the server has to first state its willingness to accept connections. This is done with the *listen* system call:

```
int error = listen(s, backlog)
```

The *backlog* is an integer specifying the upper bound on the number of pending connections that should be queued for acceptance. After a *listen*, the system is listening for connections to that socket. The connections can now be accepted with the *accept* system call, which extracts the first connection request on the queue of pending connections. It creates a new socket with the same properties as the listening socket and allocates a new file descriptor for it. By default, socket operations are synchronous, or blocking, and *accept* will block until a connection is present on the queue. The syntax of *accept* is:

```
int s;
struct sockaddr *clientaddr;
int clientaddrlen = sizeof(struct sockaddr);
int snew = accept(s, clientaddr, &clientaddrlen);
```

The *clientaddr* structure allows a server to obtain the client address. *accept* returns a *new* file descriptor that is associated with a *new* socket. The address length field initially contains the size of the address structure and, on return, contains the actual size

of the address. Communication takes place on this new socket. The original socket is used for managing a queue of connection requests (you can still listen for other requests on the original socket).

4. Exchange data

Data can now be exchanged with the regular file system *read* and *write* system calls (referring to the socket descriptor). Additional system calls were added. The *send/recv* calls are similar to *read/write* but support an extra *flags* parameter that lets one peek at incoming data and to send out-of-band data. The *sendto/recvfrom* system calls are like *send/recv* but also allow callers to specify or receive addresses of the peer with whom they are communicating (most useful for connectionless sockets). Finally, *sendmsg/recvmsg* support a full IPC interface and allow access rights to be sent and received. Could this have been designed cleaner and simpler? Yes. The point to remember is that the *read*/write or *send/ recv* calls must be used for connection-oriented communication and *sendto/ recvfrom* or *sendmsg/ recvmsg* must be used for connectionless communication. Remember that with stream virtual circuit service (SOCK_STREAM) and with datagram service (SOCK_DGRAM) the other side may have to perform multiple reads to get results from a single write (because of fragmentation of packets) or vice versa (a client may perform two *write*s and the server may read the data via a single *read*).

5. Close the connection

The shutdown system call may be used to stop all further read and write operations on a socket:

```
shutdown(s);
```

Synchronous or Asynchronous

Network communication (or file system access in general) system calls may operate in two modes: *synchronous* or *asynchronous*. In the synchronous mode, socket routines return only when the operation is complete. For example, *accept* returns only when a connection arrives. In the asynchronous mode, socket routines return immediately: system calls become non-blocking calls (e.g., *read* does not block). You can change the mode with the *fcntl* system call. For example,

```
fcntl(s, F_SETFF, FNDELAY);
```

sets the socket s to operate in asynchronous mode.

Sockets under Java

Java provides several classes to enable programs to use sockets. They are found in the java.net package. The various classes provide for client sockets, server sockets (which can accept connections), and low-level datagram objects.

Client sockets

A client that wants to establish a socket connection to a server, it creates a Socket with a constructor such as:

```
Socket s = new Socket(host, port);
```

Several forms of this constructor exist, allowing you to specify an Internet address instead of a host name, allowing you to specify a local address (useful when you want to limit the socket to a specific network), and allowing you to specify whether you want virtual circuit (stream) or datagram connectivity. For example, to open a datagram (UDP/IP) connection to the *who* service on port 513 on cs.rutgers.edu:

```
Socket s = new Socket("cs.rutgers.edu", 513, false);
```

The final parameter is true for virrtual circuit service and false for datagram service. Now the program can obtain an InputStream and an OutputStream:

```
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
```

InputStream and OutputStream are part of the java.io package and support *read* and *write* methods respoectively, allowing data to be read from and written to the socket. After the work is done, the socket can be closed with the *close* method:

```
s.close();
```

Server sockets

A server socket is a socket that is set up for listening for connection requests for clients. At the system call level, it is a socket that is listening for connections (*listen* system call). In Java, this is encapsulated into a ServerSocket class. There are several forms of the constructor; a useful one is:

```
ServerSocket svc = new ServerSocket(int port, int backlog);
```

Here, the port is the port number (transport address) that clients will connect to and backlog is a queue length to enable the operating system to buffer up connections until they are serviced (so that simultaneous connections do not get dropped). Once you have a ServerSocket, you can use the *accept* method to wait for (block on) connections:

```
Socket req = svc.accept();
```

When a connection from a client is received, *accept* returns a new socket. This socket can be used for communication with the application by obtaining an InputStream and OutputStream as in client sockets. For example,

```
DataInputStream in = new DataInputStream(req.getInputStream());
PrintStream out = new PrintStream(req.getOutputStream());
```

To close the connections, whatever streams, readers, writers, et al. were created should be closed. Finally, the socket should be closed. In this example, on the client:

```
in.close();
out.close();
s.close()
```

On the server:

```
in.close();
out.close();
req.close();
svc.close();
```

Directing sockets: what goes on inside the operating system

We've seen how we can address an application with sockets by specifying a <machine, port number> tuple. If we look at this more carefully, we notice that the server, upon performing an *accept*, gets a *new* socket on which to accept communications for *that* socket and yet the clients need to only address the machine and port number. The operating system is responsible for routing a packet to its destination socket.

Here's how it works: The server does a *socket* system call and gets a file descriptor (a socket) from the operating system. Then it *bind*s that socket to a port number and tells the operating system that it's willing to accept connections with the *listen* system call. It then sleeps in *accept* waiting for a connection. When a new connection comes in, the *accept* returns with a new socket. In this case, another socket does not mean a binding to yet another port number. When a socket is created with the *socket* system call, a structure called the protocol control block (PCB) is allocated and initialized to 0. The protocol control block for that socket contains the following:

server:

local address	*local port*	*foreign address*	*foreign port*
	0	0	0

When we *bind* to a socket, the local port and address are assigned to it. Let's say we bind to address 0 (any address for this host) and port 1234:

server:

local address	*local port*	*foreign address*	*foreign port*
0,0,0,0	1234	0	0

Now we can listen on this socket and accept a connection.

At this time, let's see what the client does. It also creates a socket and a protocol control block entry that's initialized to 0:

client:

local address	*local port*	*foreign address*	*foreign port*
0	0	0	0

Let's say that we don't bind to an explicit port number. The OS decides on a local port number that the socket will own and fills in the protocol control block entry appropriately (let's say it picks port 7801):

client:

local address	*local port*	*foreign address*	*foreign port*
0	7801	0	0

When the client does a *connect* to the server (connect to the server on port 1234), it sends along its address (its IP address on that particular network) and its port number. If we assume that the client is 125.250.68.43 and the server is 192.11.35.15:

connect to server: *send 135.250.68.43:7801 to 192.11.35.15:1234.*

Now, back to the server. The server, sleeping in *accept*, accepts the new connection and returns a new socket descriptor back to the application. The operating system allocates a *new* protocol control block that is a copy of the one on which we were listening but has the foreign address/port fields filled with what the client sent:

sever:

local address	*local port*	*foreign address*	*foreign port*
0.0.0.0	1234	0	0
0.0.0.0	1234	135.250.68.43	7801

The server then sends an acknowledgment back to the client containing its real address and port number. The client now fills that into its PCB entry:

client:

local address	*local port*	*foreign address*	*foreign port*
0	7801	192.11.35.15	1234

Every message from the client is tagged as being data or control (such as a *connect*). If it is data, the operating system searches through the list of PCB's that have the foreign address and port matching those in the incoming message. If it is a control message, the operating system searches for a PCB with 0's for the foreign address and port.

So, even though we have one port number and one application listening (for address,port=0,0), we can have multiple outstanding data connections, each uniquely identified by the set (local address, local port, foreign address, foreign port).

References

Computer Networks, Andrew Tanenbaum, Second edition ©1989 Prentice Hall. [covers networking protocols and theory]

Connected: An Internet Encyclopedia, Third Edition. IP Addressing Review, http://www.freesoft.org/CIE/Course/Subnet/1.htm

Distributed Operating Systems, Andrew Tanenbaum, ©1995 Prentice Hall. [provides a brief description of ATM and coverage of sockets, NFS] *The Design and Implementation of the 4.3 BSD UNIX® Operating System*, S.J. Leffer, M.K. McKusick, M.J. Karels, J.S. Quarterman, ©1989 Addison-Wesley Publishing Co. [describes the design and implementation of sockets]

JAVA in a Nutshell: A Desktop Quick Reference, David Flanagan, © 1997 O'Reilly & Associaes, Inc. [a concise reference to Java and the JDK]

Modern Operating Systems, Andrew Tanenbaum, ©1992 Prentice Hall. [provides a high level description of NFS, sockets, and networking in general]

Networking Applications on UNIX System V Release 4, Michael Padovano, ©1993 Prentice Hall. [good coverage of socket programming and TLI programming; nicely written with good explanations of why you may want to choose one system over another and what to watch out for; also discusses NFS and RFS file systems]

*Classless Inter-Domain Routing (CIDR) Overvie*w, Pacific Bell Internet, http://public.pacbell.net/dedicated/cidr.html

TCP/IP Illustrated, Volume 1: The Protocols, W. Richard Stevens, ©1994 Prentice Hall. [if you want to learn a lot about the TCP protocol, this is the book to get]

UNIX Network Programming, W. Richard Stevens, ©1990 Prentice Hall. [thorough coverage of sockets programming, authentication, TLI programming, and IPC for UNIX]