Transport Level Protocols

The transport level provides *end-to-end communication between processes executing on different machines.* Although the services provided by a transport protocol are similar to those provided by a data link layer protocol, there are several important differences between the transport and lower layers:

1. *User Oriented.* Application programmers interact directly with the transport layer, and from the programmers perspective, the transport layer is the "network". Thus, the transport layer should be oriented more towards user services than simply reflect what the underlying layers happen to provide. (Similar to the beautification principle in operating systems.)

2. *Negotiation of Quality and Type of Services.* The user and transport protocol may need to negotiate as to the quality or type of service to be provided. Examples? A user may want to negotiate such options as: throughput, delay, protection, priority, reliability, etc.

3. *Guarantee Service.* The transport layer may have to overcome service deficiencies of the lower layers (e.g. providing reliable service over an unreliable network layer).

4. *Addressing* becomes a significant issue. That is, now the user must deal with it; before it was buried in lower levels. How does a user open a connection to "the mail server process on *wpi*"?

   Two solutions:

   (a) Use *well known addresses* that rarely if ever change, allowing programs to "wire in" addresses. For what types of service does this work? While this works for services that are well established (e.g., mail, or telnet), it doesn't allow a user to easily experiment with new services.

   (b) Use a *name server.* Servers register services with the name server, which clients contact to find the transport address of a given service.

   In both cases, we need a mechanism for mapping high-level service names into low-level encodings that can be used within packet headers of the network protocols. In its general form, the problem is quite complex.

   One simplification is to break the problem into two parts: have transport addresses be a combination of machine address and local process on that machine.

5. *Storage capacity of the subnet.* Assumptions valid at the data link layer do not necessarily hold at the transport Layer. Specifically, the subnet may buffer messages for a potentially long time, and an "old" packet may arrive at a destination at unexpected times.

6. *We need a dynamic flow control mechanism.* The data link layer solution of preallocating buffers is inappropriate because a machine may have hundreds of connections sharing a single physical link. In addition, appropriate settings for the flow control parameters depend on the communicating end points (e.g., Cray supercomputers vs. PCs), not on the protocol used.

   Don't send data unless there is room. Also, the network layer/data link layer solution of simply not acknowledging frames for which the receiver has no space is unacceptable. Why? In the data link case, the line is not being used for anything else; thus retransmissions are inexpensive. At the transport level, end-to-end retransmissions are needed, which wastes resources by sending the same packet over the same links multiple times. If the receiver has no buffer space, the sender should be prevented from sending data.

7. *Deal with congestion control.* In connectionless internets, transport protocols must exercise *congestion control.* When the network becomes congested, they must reduce rate at which they insert packets into the subnet, because the subnet has no way to prevent itself from becoming overloaded.

8. *Connection establishment.* Transport level protocols go through three phases: establishing, using, and terminating a connection.

   For datagram-oriented protocols, opening a connection simply allocates and initializes data structures in the operating system kernel.

   Connection oriented protocols often exchanges messages that negotiate options with the remote peer at the time a connection is opened. Establishing a connection may be tricky because of the possibility of old or duplicate packets.

   Finally, although not as difficult as establishing a connection, terminating a connection presents subtleties too. For instance, both ends of the connection must be sure that all the data in their queues have been delivered to the remote application.

We'll look at these issues in detail as we examine TCP and UDP. Not too much of OSI terminology as discussed by Tanenbaum.

User Datagram Protocol (UDP)

UDP provides *unreliable* datagram service. It uses the raw datagram service of IP and does not add acknowledgements or retransmissions.

Need delivery to a process. The first difference between UDP and IP is that IP includes only enough information to deliver a datagram to the specified machine. Transport protocols deal with process-to-process communication. How can we specify a particular process?

Although it is convenient to think of transport service between processes, this leads to some problems:

- Processes are identified differently on different machines; we don't want to have machine or operating system dependencies in our protocol.
- Processes may terminate and restart. If a machine reboots, we don't want to have to tell other machines about it.
- Associating a single process with a connection makes it difficult to have multiple processes servicing client requests (e.g. file server processes on a file server).

The solution is to add a level of indirection. Transport level address refer to *services* without regard to who actually provides that service. In most cases, a transport service maps to a single process.

TCP and UDP use *ports* to identify services on a machine. Conceptually, ports behave like mailboxes. Datagrams destined for a port are queued at the port until some process reads them, and each service has its own mailbox.

Like all packets we've seen, UDP datagrams consist of a UDP header and some data. The UDP header contains the following fields:

Source port (16 bits) : Port number of the sender.

Destination port (16 bits) : Port number of the intended recipient. UDP software uses this number to demultiplex a datagram to the appropriate higher-layer software (e.g. a specific connection).

Length (16 bits) : Length of the entire UDP datagram, including header and data.

Checksum (16 bits) : Checksum of entire datagram (including data).

The checksum field is unusual because it includes a 12-byte pseudo header that is not actually part of the UDP datagram itself. The information in the pseudo header comes from the IP datagram header:

IP source address (4 bytes) : Sending machine.

IP destination address (4 bytes) : Destination machine.

UDP Length (2 bytes) : Length of UDP datagram, as given by the lengths in the IP header.

protocol (1 byte) : protocol field of the IP header; should be 17 (for UDP)!

zero (1 byte) : one byte pad containing zero.

The purpose of the pseudo header is to provide extra verification that a datagram has been delivered properly. To see why this is appropriate, recall that because UDP is a transport protocol it really deals with transport addresses. Transport addresses should uniquely specify a service regardless of what machine actually provides that service.

Note: the use of a pseudo header is strong violation of our goal of layering. However, the decision is a compromise based on pragmatics. Using the IP address as part of the transport address greatly simplifies the problem of mapping between transport level addresses and machine addresses.

Port Addresses

How are port addresses be assigned?

- Port numbers 0-255 for reserved for well-known ports. They are reserved for such universal services as mail, telnet, and ftp. Well-known ports are administrated by a central authority. Also ports for services specific to UNIX machines (*/etc/services*).
- Sites are free to assign the remaining ports any way they wish.

Note: UDP does not address the issue of flow control or congestion control. Thus, it is unsuitable for use as a general transport protocol.

UDP datagrams are reliable on the same machine and highly reliable on a LAN such as Ethernet.

Transmission Control Protocol (TCP)

TCP provides reliable, full-duplex, byte stream-oriented service. It resides directly above IP (and adjacent to UDP), and uses acknowledgments with retransmissions to achieve reliability. TCP differs from the sliding window protocols we have studied so far in the following ways:

1. When using TCP, applications treat the data sent and received as an arbitrary byte stream.

   The sending TCP module divides the byte stream into a set of packets called *segments*, and sends individual segments within an IP datagram.

   TCP decides where segment boundaries start and end (the application does not!). In contrast, individual packets are handed to the data link protocols.

2. The TCP sliding window operates at the byte level rather than the packet (or segment) level.

   The left and right window edges are byte pointers.

3. Segment boundaries may change at any time. TCP is free to retransmit two adjacent segments each containing 200 bytes of data as a single segment of 400 bytes.

4. The size of the send and receive window change dynamically. In particular, acknowledgments contain two pieces of information:

   1) A conventional ACK indicating what has been received, and

   2) The current receiver's window size; that is, the number of bytes of data the receiver is willing to accept.

   The presence of *flow control* at the transport level is important because it allows a slow receiver to shut down a fast sender. For example, a PC can direct a supercomputer to stop sending additional data until it has processed the data it already has.

TCP Header

TCP segments contain a TCP header followed by user data. TCP segments contain the following fields:

Source & destination port (2×16 **bits**) : TCP port number of the sender and receiver. TCP ports are essentially the same as UDP ports, but are assigned separately. Thus, TCP port 54 may refer to a different service than UDP port 54.

Sequence number (32 bits) : The sequence number of the first byte of data in the data portion of the segment.

Acknowledgment number (32 bits) : The next byte expected. That is, the receiver has received up to and including every byte prior to the acknowledgment.

Why use 32-bit sequence numbers? Transport protocols must always consider the possibility of delayed datagrams arriving unexpectedly.

Consider the following:

1. Suppose we use a sequence number space of 16 bits (0–65535).
2. Application A sends B several megabytes of data.
3. Each segment contains 1K bytes of data. How long before we start reusing sequence numbers? After only 65 segments.
4. Our internet is really misbehaving:

   1) A retransmits every segment 3 times. (Perhaps our retransmit timers are wrong.)

   2) Packets aren't being lost, just delayed.
5. At some point, TCP B expects sequence numbers 0—4096, but an old duplicate having a sequence number of 1024 arrives. B will incorrectly accept the duplicate as new data.
6. The described scenario is not entirely unrealistic. For example, a cross-country file transfer can easiliy have a throughput of greater than 20 kbps. (e.g., 20 1k segments per second).

Insuring that our sequence number space is large enough to detect old (invalid) datagrams depends on two factors:

1. The amount of wall-clock time that a datagram can remain in the network.
2. The amount of time that elapses before a given sequence number becomes reused. Thus, we use 32 bit sequence numbers. In addition, now we see why IP has a TTL field — we need to insure that datagrams don't stay in the network for too long.

Flow control window (16 bits) : The size of the receive window, relative to the acknowledgment field. The sender is not allowed to send any data that extends beyond the right edge of the receiver's receive window. If the receiver cannot accept any more data, it advertises a flow-control window of zero.

Checksum (16 bits) : Checksum of TCP header and data.

Options (variable length) : Similar to IP options, but for TCP-specific options.

One interesting option is the *maximum segment size* option, which allows the sender and receiver to agree on how large segments can be. This allows a small machine with few resources to prevent a large machine from sending segments that are too large for the small machine to handle. On the other hand, larger segments are more efficient, so they should be used when appropriate.

Padding (variable) : Padding to insure that the size of the TCP header is a multiple of 32 bits.

Data offset (4 bits) : The number of 32-bit words in the TCP header. Used to locate the start of the data section.

Note: A TCP segment does not have to contain any data.

Urgent pointer (16 bits) : When urgent data is present, this field indicates the byte position (relative to the sequence number) just past *urgent data*.

The *urgent pointer* is something we have not encountered before. It allows the sending application to indicate the presence of high-priority data that should be processed ASAP (e.g., drop what you are doing and read all the input).

One example use is when a *telnet* user types CTRL-C to abort the current process. Most likely, the user would like to get a prompt right away and does not want to see any more output from the aborted job. Unfortunately, there may be thousands of bytes of data already queued in the connection between the remote process and the local terminal.

The local shell places the CTRL-C in the input stream and tells the remote telnet that urgent data is present. The remote telnet sees the urgent data, then quickly reads the data, and when it sees the CTRL-C, throws away all the input and kills the running job. The same type of action then takes place so that the remote telnet can signal the local telnet to throw away any data that is in the pipeline.

Flags (6 bits) : The flags field consists of 6 1-bit flags:

- **Urgent pointer (URG)** : If set, the urgent pointer field contains a valid pointer. If the urgent pointer flag is 0, the value of the urgent field is ignored.
- **Acknowledgment valid (ACK bit)** : Set when the acknowledgment field is valid. In practice, the only time that the ACK bit is not set is during the 3-way handshake at the start of the connection.
- **Reset (RST)** : The reset flag is used to abort connections quickly. It is used to signal errors rather than the normal termination of a connection when both sides have no more data to send.

  Upon receipt of a RST segment, TCP aborts the connection and informs the application of the error.
- **Push (PSH)** : Flush any data buffered in the sender or receiver's queues and hand it to the remote application.

  The PSH bit is requested by the sending application; it is not generated by TCP itself. Why is it needed?

  1. TCP decides where segment boundaries start and end.
  2. The sending TCP is free to delay sending a segment in the hope that the application will generate more data shortly. This performance optimization allows an application to (inefficiently) write one byte at a time, but have TCP package many bytes into a single segment.
  3. A client may send data, then wait for the server's response. What happens next? If TCP (either the sender or receiver) is buffering the request, the server *application* won't have received the request, and the client will wait forever (e.g., deadlock).
  4. To prevent deadlock, the client sets the PSH flag when it sends the last byte of a *complete* request. The PSH directs TCP to flush the data to the remote application.
- **Synchronization (SYN)** : Used to initiate a new connection. (Described below.)
- **Finish (FIN)** : Used to close a connection. (Described below.)

3-Way Handshake

pg 395 Tanenbaum

TCP uses a 3-way handshake to initiate a connection. The handshake serves two functions:

1. It ensures that both sides are ready to transmit data, and that *both* ends know that the other end is ready *before* transmission actually starts.
2. It allows both sides to pick the initial sequence number to use.

When opening a new connection, why not simply use an initial sequence number of 0? Because if connections are of short duration, exchanging only a small number of segments, we may reuse low sequence numbers too quickly. Thus, each side that wants to send data must be able to choose its initial sequence number. The 3-way handshake proceeds as follows:

1. TCP *A* picks an initial sequence number (A_SEQ) and sends a segment to *B* containing: SYN_FLAG=1, ACK_FLAG=0, and SEQ=A_SEQ.
2. When TCP *B* receives the SYN, it chooses its initial sequence number (B_SEQ) and sends a TCP segment to *A* containing: ACK=(A_SEQ+1), ACK_BIT=1, SEQ=B_SEQ, SYN_FLAG=1.
3. When *A* receives *B*'s response, it acknowledges *B*'s choice of an initial sequence number by sending a dataless third segment containing: SYN_FLAG=0, ACK=(B_SEQ+1), ACK_BIT=1, SEQ=A_SEQ+1 (data length = 0).
4. Data transfer may now begin.

Note: The sequence number used in SYN segments are actually part of the sequence number space. That is why the third segment that *A* sends contains SEQ=(A_SEQ+1). This is required so that we don't get confused by old SYNs that we have already seen.

To insure that old segments are ignored, TCP *ignores* any segments that refer to a sequence number outside of its receive window. This includes segments with the SYN bit set.

Terminating Connections

An application sets the FIN bit when it has no more data to send. On receipt of a FIN segment, TCP refuses to accept any more *new* data (data whose sequence number is greater than that indicated by the FIN segment).

Closing a connection is further complicated because receipt of a FIN doesn't mean that we are done. In particular, we may not have received all the data leading up to the FIN (e.g., some segments may have been lost), and we must make sure that we have received all the data in the window.

Also, FINs refer to only 1/2 of the connection. If we send a FIN, we cannot send any more new data, but we must continue accepting data sent by the peer. The connection closes only after both sides have sent FIN segments.

Finally, even after we have sent and received a FIN, we are not completely done! We must wait around long enough to be sure that our peer has received an ACK for its FIN. If it has not, and we terminate the connection (deleting a record of its existence), we will return a RST segment when the peer retransmits the FIN, and the peer will abort the connection.

Two-army Problem

White army to attack the blue. There is no protocol for correctly coordinating the attack. Can only communicate through unreliable means. Is the last messenger necessary? Yes.

There is no satisfactory solution to the problem. Analagous to closing a connection. Best we can do is get it right most of the time.

TCP Congestion Control

Transport protocols operating across connectionless networks must implement *congestion control*. Otherwise, *congestion collapse* may occur. Congestion collapse occurs when the network is so overloaded that it is only forwarding retransmissions, and most of them are delivered only part way before being discarded. Congestion control refers to reducing the offered load on the network when it becomes congested.

What factors govern the rate at which TCP sends segments?

1. The current sending window size specifies the amount of data that can be in transmission at any one time. Small windows imply little data, large windows imply a large amount of data.
2. If our retransmit timer is too short, TCP retransmits segments that have been delayed, but not lost, increasing congestion at a time when the network is probably already congested!

Both of these factors are discussed in the following subsections

TCP Retransmission Timers

What value should TCP use for a retransmission timer?

1. If our value is too short, we will retransmit prematurely, even though the original segment has not been lost.
2. If our value is too long, the connection will remain idle for a long period of time after a lost segment, while we wait for the timer to go off.
3. Ideally, we want our timer to be close to the true round trip (delay) time (RTT). Because the actual round trip time varies dynamically (unlike in the data link layer), using a fixed timer is inadequate.

To cope with widely varying delays, TCP maintains a dynamic estimate of the current RTT:

1. When sending a segment, the sender starts a timer.
2. Upon receipt of an acknowledgment, stop the timer and record the actual elapsed delay between sending the segment and receiving its ACK.
3. Whenever a new value for the current RTT is measured, it is averaged into a *smoothed* RTT (SRTT) as follows:

   $$SRTT = (\alpha \times SRTT) + ((1 - \alpha) \times RTT)$$

   α is known as a *smoothing* factor, and it determines how much weight the new measurement carries. When α is 0, we simply use the new value; when α is 1, we ignore the new value.

   Typical values for α lie between .8 and .9.

   Because the actual RTT naturally varies between successive transmissions due to normal queuing delays, it would be a mistake to throw out the old one and use the new one. Use of the above formula causes us to change our SRTT estimate slowly, so that we don't overreact to wild fluctuations in the RTT.

Because the SRTT is only an estimate of the actual delay, and actual delays vary from packet to packet, set the actual retransmission timeout (RTO) for a segment to be somewhat longer than SRTT. How much longer?

TCP also maintains an estimate of the *mean deviation* (MDEV) of the RTT. MDEV is the difference between the measured and expected RTT and provides a close approximation to the *standard deviation*. Its computation is as follows:

$$SMDEV = (\alpha \times SMDEV) + ((1 - \alpha) \times MDEV)$$

Finally, when transmitting a segment, set its retransmission timer to RTO:

$$RTO = SRTT + 4 \times SMDEV$$

Was originally proposed as 2, but further experience has shown 4 to be better.

Early versions of TCP (4.2 and 4.3 BSD) used a much simpler retransmission algorithm that resulted in excessive retransmissions under some circumstances. Indeed, improper retransmission timers led to excessive retransmissions which contributed to congestion collapse.

Slow-Start TCP

The second congestion control mechanism in TCP adjusts the size of the sending window to match the current ability of the network to deliver segments:

1. If the send window is small, and the network is idle, TCP will make inefficient use of the available links.
2. If the send window is large, and the network is congested, most segments will be using gateway buffer space waiting for links to become available.
3. Even in an unloaded network, the optimal window size depends on network topology: To keep all links busy simultaneously, exactly one segment should be in transmission on each link along the path. Thus, the optimal window size depends on the actual path and varies dynamically.

TCP uses a *congestion window* to keep track of the appropriate send window relative to network load. The congestion window is not related to the flow-control window, as the two windows address orthogonal issues. Of course, the actual send window in use at any one time will be the smaller of the two windows. There are two parts to TCP's congestion control mechanism:

1. Increase the sender's window to take advantage of any additional bandwidth that becomes available.

   This case is also referred to as *congestion avoidance* and is handled by *slowly*, but continually, increasing the size of the send window. We want to slowly take advantage of available resources, but not so fast that we overload the network. In particular we want to increase so slowly that we will get feedback from the network or remote end of the connection before we've increased the level of congestion significantly.

2. Decrease the sender's window suddenly and significantly in response to congestion.

   This case is known as *congestion control* and is handled by decreasing the window *suddenly* and *significantly*, reacting after the network becomes overloaded.

An Example

To see how things work, let us assume that TCP is transmitting at just the right level for current conditions. During the *congestion avoidance* phase, TCP is sending data at the proper rate for current conditions.

To make use of any additional capacity that becomes available, the sender slowly increases the size of its send window. When can the sender *safely* increase its send window size?

As long as it receives a positive indication that the data it is transmitting is reaching the remote end, none of the data is getting lost, so there must not be much (if any) congestion. Specifically, TCP maintains a variable *cwnd* that specifies the current size of the congestion window (in segments). When TCP receives an acknowledgment that advances the send window, increase *cwnd* by 1/*cwnd*.

This *linear* increase enlarges the size of the congestion window by one segment every round trip time. (The increase is linear in real time because it window increases by a constant amount every round trip time.)

Because the send window continually increases, the network will eventually become congested. How can TCP detect congestion? When it fails to receive an ACK for a segment it just sent. When the sender detects congestion, it

halves the current size of the congestion window,

saves it in a temporary variable *ssthresh*, and

sets *cwnd* to 1.

At this point, *slow start* takes over.

During slow start, the sender increases *cwnd* by one on every new ACK.

In effect, the sender increases the size of the window *exponentially*, doubling the window size every round trip time.

Once *cwnd* reaches *ssthresh*, congestion avoidance takes over and the window resumes its linear increase.

Slow Start Properties

Slow start has several important properties:

1. If the network is congested, the transmission rate drops precipitously (e.g., exponentially). If the network is congested, it is better to be conservative and stop transmitting quickly.
2. Increasing the size of the send window linearly is necessary. Analysis and simulation shows that an exponential increase is unstable, while a linear increase is not.
3. If a segment has been lost, do we just retransmit the lost segment, or everything in the send window? Sending just the segment known to be lost would make better use of network resources (in general). Moreover, when the ACK for the retransmitted segment finally arrives, the ACK will uncover an entire window of new data (assuming that only the one segment was lost). If the sender transmits an entire window's worth of data at once, the data will be sent as a burst. If the network is operating near its congestion level, a packet burst is likely to result in dropped packets.

   Slow start guarantees that a sender will never transmit more than two back-to-back packets.

Finally, how does TCP detect the presence of congestion? Because source quench messages are unreliable, TCP assumes that all lost packets result from congestion. Thus, a retransmission event triggers the slow start phase of the algorithm.

Nagle's Algorithm

To avoid sending too many small packets Nagle proposed that no "less than full" packets should be sent if unACKed packets exist.

The idea is to avoid *silly window syndrome* where many one-byte data packets are sent one after another.

Works well in reducing network congestion, but some interactive applications such as X-Windows or HTTP/1.1 pipelining must disable it to obtain better performance. Otherwise X-Windows exhibits jerky mouse movement and HTTP/1.1 with pipelining exhibits delays in sending requests.

Wireless TCP

On wireless links, dropped packets are not necessarily because of congestion, but because of bit errors. Don't want TCP to react to errors (timeouts) by reducing the congestion window.

Domain Name System (DNS)

Lower-level protocol layers use compact 32-bit Internet addresses. In contrast, users prefer meaningful names to denote objects (e.g., *eve*). Using high-level names requires an efficient mechanism for mapping between high-level names and low-level addresses.

Originally, the Internet was small and mapping between names and addresses was accomplished using a centrally-maintained file called *hosts.txt*. To add a name or change an address required contacting the central administrator, updating the table, and distributing it to all the other sites. This solution worked at first because most sites had only a few machines, and the table didn't require frequent changes. The centrally-maintained table suffered from several drawbacks:

1. The name space was *flat*, and no two machines could use the same machine name.
2. As the Internet grew, changes to the database took days to weeks to take effect.
3. The central site (*nic.ddn.mil*, previously known as *sri-nic.arpa*) became congested with the increase in the number of sites retrieving copies of the current table.
4. The Internet grew at an astonishing rate.

The Domain Name System (DNS) is a hierarchical, distributed naming system designed to cope with the problem of explosive growth:

1. It is *hierarchical* because the name space is partitioned into *subdomains.*
2. It is distributed because management of the name space is delegated to local sites. Local sites have complete control (and responsibility) for their part of the name space.

   DNS queries are handled by servers called *name servers.*
3. It does more than just map machine names to internet addresses. For example, it allows a site to associate multiple machines with a single, site-wide mailbox name.

In the DNS, the name space is structured as a tree, with *domain names* referring to nodes in the tree. The tree has a *root*, and a *fully-qualified* domain name is identified by the *components* of the path from the domain name to the root.

In figure *cs.purdue.edu*, *garden.wpi.edu*, and *decwrl.dec.com* are fully-qualified domain names.

The top level includes several subdomains, including (among others):

edu: Educational organizations (500,000+ registered.)

com: Companies (e.g., *ibm.com*). (460,000+ .)

net: Organizations offering network service (e.g., *nyser.net).* (18,000+.)

gov: Government organizations (e.g., *nsf.gov*). (100,000+.)

Resource Records

The DNS links data objects called *resource records* (RRs) to domain names. RRs contain information such as internet addresses or pointers to name servers.

Resource records consist of five parts:

Owner (variable length): The domain name associated with the RR.

Type (16 bits): The type of data the RR contains:

- **A:** An address.
- **MX (mail exchanger):** A list of domain names that are willing to accept mail addressed to this RR's *owner*. Each MX record consists of a preference number (lower numbers are preferred) and a domain name.
- **HINFO:** Information about a host such as vendor, brand, model, architecture and operating system.
- **PTR:** Pointer to another domain name.
- **NS:** Name server.

Class: The protocol type for *type* (e.g, Internet, ISO, etc.) Even though the DNS was designed to solve an Internet problem, it can be used by other protocol families.

Time to live (32 bits): The length of time that a client may cache the RR (in seconds). Note: the TTL here serves a completely different purpose than the one found in the IP header.

Data (variable length): The actual data stored in the RR; actual format depends on *type* and *class*.

The following table gives a set of sample RRs in use at WPI. Nameservers: *ns.wpi.edu*, *ns1.barrnet.net*, *ns3.cw.net* and *ece.wpi.edu*. Information can be obtained from the Unix program *nslookup*.

OWNER	TYPE	DATA
garden.wpi.edu	HINFO	DS5k/240 FDDI ULTRIX
wpi.wpi.edu	HINFO	DS5000/260 ULTRIX
garden.wpi.edu	A	130.215.8.200
walnut.wpi.edu	A	130.215.8.90
wpi.wpi.edu	MX	0 bigboote.wpi.edu
garden.wpi.edu	MX	0 bigboote.wpi.edu
cs.wpi.edu	MX	0 owl.wpi.edu

Note: The mail address *cew@cs.wpi.edu* is valid, even though there is no machine called *cs.wpi.edu*. Also note that mail to *garden* and *wpi* go to *bigboote*.

How does the SMTP mailer decide which machine to send mail addressed to XXX@cs.wpi.edu?

1. It queries the DNS for a RR of type MX for name *cs.wpi.edu*. In Unix, the utility *sendmail* handles the task of mail delivery.
2. The returned RR contains a list of machines that vare willing to accept mail for domain *cs.wpi.edu*, and the mailer attempts to contact each one in succession until it is able to establish a connection with a machine willing to accept the mail.
3. Lower preferences are tried first. We have only one machine. Could have backups.

Name servers are the programs that actually manage the name space. The name space is divided into *zones of authority*, and a name server is said to be *authoritative* for all domain names within its zone.

Name servers can *delegate* responsibility for a *subdomain* to another name server, allowing a large name space to be divided into several smaller ones.

At Purdue, for instance, the name space *purdue.edu* is divided into three subdomains: *cs*, *cc*, and *ecn*.

Name servers are linked by pointers. When a name server delegates authority for a subdomain, it maintains pointers to the name servers that manage the subdomain. Thus, the DNS can resolve fully-qualified names by starting at the root and following pointers until reaching an authoritative name server for the name being looked up. (See the DNS record of type PTR.

Note: The shape of the name space and the delegation of subdomains does not depend on the underlying topology of the Internet.

DNS Queries

When a client (application) has a name to translate, it sends a DNS *query* to a name server. DNS queries (and responses) are carried within UDP datagrams. There are two types of queries:

Recursive: The server resolves the name completely, even if it has to send additional queries to other servers in order to obtain the desired answer.

Iterative: If the name server can't answer the query, have it return a pointer to another name server that has more information. The client then sends the query to the other name server. Note:

1. If the name belongs to a subdomain, the server returns a pointer to the name server responsible for that part of the name space.
2. If the server has no information about the name, it returns a pointer to the root name servers.

DNS Queries (and responses) consist of four parts:

Question: A section containing a list of questions (domain name, type, class triples). A single query can contain several questions.

Answer: A section containing answers to the queries in the question section. The answer section is filled in by the server.

Authority: If the query cannot be resolved, a section containing pointers to authoritative name servers that can. That is, if the query has been directed to a non-authoritative server, it may not be able to answer the query. This happens frequently, because a client almost always asks its local server to translate everything, even non-local names.

Additional: RRs that are likely to be of help to the clients, such as the Internet address of servers listed in the authority section (e.g., "hints").

Conceptually, any application that accesses information managed by the DNS must query the DNS. In practice, DNS queries are hidden in library routines that a user simply calls without having to worry about how they are implemented. In Unix, for example, the routine *gethostbyname(3)* finds the IP address of a host. Although *gethostbyname* interacts with name servers, it behaves like a regular procedure call.

The DNS also addresses two important issues:

Caching: Clients cache responses to reduce the number of (repetitious) queries sent to other name servers. This greatly reduces the number of queries, because most applications refer to the same names repeatedly.

Note: The owner of a RR manages the caching behavior for its names. Each RR includes a TTL field that specifies for how long a name may be cached. For names that don't change often, long time outs (e.g. several days) are used.

Replication: The DNS allows multiple authoritative name servers for a given zone. Thus, if a server is down, another might still be able answer a query.

Typically, one name server is designated the *master*, with the remaining servers designated slaves. The master/slave machines run a special protocol so that slave servers obtain new copies of the database whenever it changes. However, clients may query either masters or slaves.

Is the address 130.215.24.1 hierarchical? Yes. Thus, the DNS also maps machine addresses back into host names. How? By reversing them!

The top level of the name space includes the domain *in-addr.arpa*, and machine addresses are reversed and converted into the form: *1.24.215.130.in-addr.arpa*, which can be translated by the DNS.