4 Operating System Support

Aim: What do we need from an operating system to support distributed systems?

Main Points:

Layering An organisational principle for modular communications

Protocol techniques Acknowedgements, timers, windows etc

Operating Systems Abstractions Processes, threads, sockets, daemons

Basic service provided by network - variable loss, bandwidth, latency.
Need to layer other services on top.

4.1 Protocols

Protocol: Agreement between two (or more) parties as to how information is to be transmitted.

At minimum, will include the interpretation of the bits in the packet.
May include finite state machines movements between senders and receivers.
Protocol information in headers (or trailers) at front of packets.

4.2 Layering

Networks are used to transmit messages between processes.

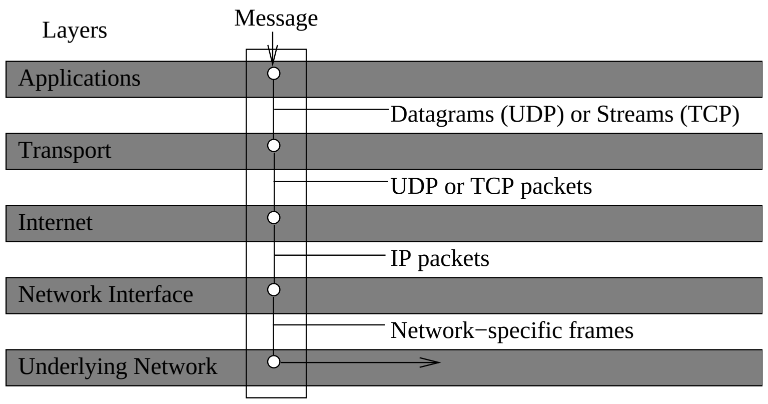
Protocols used to give messaging functionality

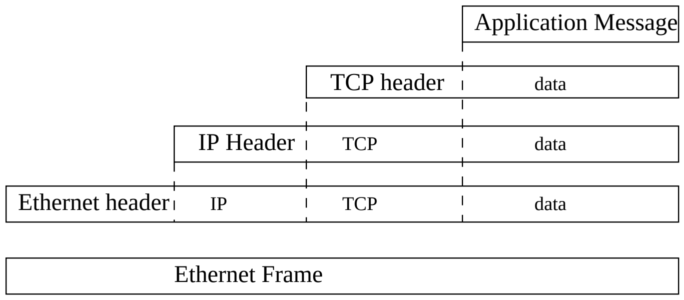
Packets in reality	Messaging abstraction
Limited Size	Arbitrary Size
Unordered (sometimes)	ordered
unreliable	reliable
machine to machine	process to process
Local Area Net	routed anywhere
Asynchronous	Synchronous
Insecure	Secure

Table 1: Packets and messages

4.2.1 Reason for Layering

- Easier to build functions with higher abstractions.
- Define ordering of abstractions to simplify necessary functions
- Provides modularity





4.2.2 Layering in the Internet

4.2.3 The Headers in an Ethernet Frame

4.3 Reliable Transmission: The Basic Techniques

These are some of the basic protocol techniques that are used in lower layer protocols.

They are often reused in higher level protocols.

4.3.1 Labelling

-
- Split up message into smaller chunks.
- Place label in header indicating which part of message it is.
  eg "abcdefg" → 1 of 3 "abc", 2 of 3 "def", 3 of 3 "g"
- Labels typically sequential fixed field integers

4.3.2 Acknowledging

To tell the sender data has been received correctly (after checking checksums etc), use *acknowledgements.*

- When the data packet is received, send an *acknowledgement* message back to the sender.
- Acknowledgement message contains the label of the message being acknowledged.
- Acknowledgement of a packet can sometimes implicitly acknowledge reception of all previously sent packets (Go back N)
- Multiple labels can be sent in the acknowledgement (Selective acknowledgement)
- If data is being returned to the sender, acknowledgement information can be *piggybacked* on return data packet. Acknowledgement information is part of the header eg TCP

4.3.3 Timeouts and retransmission

- The sender measures the expected time between sending a message and receiving an acknowledgement
- Sender starts a timer after sending a packet.
- If the timer expires before an acknowledgement is received, the packet is resent
- Receiver must be able to deal with duplicate packets

Questions

1. How can the expected round trip time be measured?
2. What value should the time be set to?

4.3.4 Negative Acknowledgement

Sometimes, the pattern of data exchange makes it easier to use *Negative* acknowledgements

- If data-flow is constant, receiver knows when packet is expected.
- Can send an negative acknowledgement (NAK) indicating expected messages hasn't been received

4.3.5 Windowing

- To increase utilization, and decrease acknowledgement overhead, multiple packets can be sent before the sender waits for an acknowledgement
- The number of packets that can be sent is the *window*.
- Window needs to be fixed to avoid overloading network or receiver.

- Careful adjustment of the window size is key to avoiding and controlling congestion and dyanmic performance (slow start in TCP).

4.3.6 State Synchronization

- Both ends of the protocol exchange typically need to *agree* on some starting *state*
- In labelling systems, both ends need to agree on initial label value
- Use a *handshake* of messages containing suggestions for state, and then return messages agreeing the value of the state.

4.4 Group Communication

- Often, sender wants the same packet replicated to multiple receivers eg game updates, mirroring etc
- Network offers *multicast* to provide this functionality.
- Receivers join a particularly addressed group - class D addresses in IP, $1110XXXX.XXXXXXXX.XXXXXXXX.XXXXXXXX$
- Network conspires to deliver packets sent to this address efficiently to all receivers.
- Available in Local Area, not always available in wide area (unfortunately).

4.5 Sockets

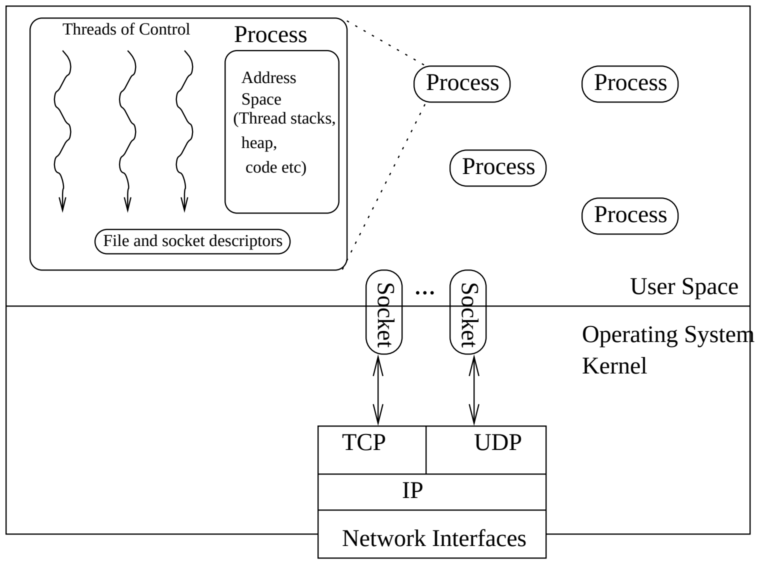
- Sockets are the near-universal abstraction for using TCP and UDP
- Provides data structures for holding addresses and other context information, and methods for sending and receiving data.
- Clients use sockets to talk to servers, often known as *daemons*[1]

4.5.1 Operating Systems Background

4.5.2 Socket Usage: Single threaded TCP Server

```
try {
   Create a socket
```

[1]from the Hackers Dictionary
daemon /day'mn/ or /dee'mn/ n.
[from the mythological meaning, later rationalized as the acronym 'Disk And Execution MONitor'] A program that is not invoked explicitly, but lies dormant waiting for some condition(s) to occur. The idea is that the perpetrator of the condition need not be aware that a daemon is lurking (though often a program will commit an action only because it knows that it will implicitly invoke a daemon). [...] Daemons are usually spawned automatically by the system, and may either live forever or be regenerated at intervals.



```
   Bind the address to the socket
   loop {
     Accept connections on socket
     Receive data on connection
     do some work
     Send Response
     Close connection on socket
   }
} catch and deal with any exceptions
```

4.5.3 Socket Usage: TCP client

```
try {
   Create a socket
   Bind the address of the server to the socket
   // Allows the operating system to choose port
   // and address for the receiver
   Open connection on socket // Connection setup
   Send data on connection
   Wait for Response
   Close connection on socket
} catch and deal with any exceptions
```

4.5.4 Gotchas

- The chunk of data written to a socket is not necessarily read as a chunk for the remote socket - packetisation may fragment the chunks.

- Both ends have to agree on how to interpret the data written and read from the socket - the concrete syntax of the data stream.
- Data interpretation is through application standards defining bit fields eg the RFCs defining HTTP, SMTP, FTP etc
- Interpreting data can require very messy bit manipulation.

4.5.5 Socket Usage: Concurrent TCP Server

- A single threaded server can only deal with one client at a time
- This is often unacceptable in terms of performance eg web servers

```
try {
   Create a socket
   Bind the address to the socket
   loop {
     Accept connections on socket
     spawn a worker thread to deal with connection
   }
} catch and deal with any exceptions
```

and the worker thread goes

```
try {
   Receive data on Connection
   Do some work
   Send response
   Close Connection
} catch and deal with any exceptions
```

4.5.6 Thread Pools

- Spawning a thread for every connection can consume too many resources if connections come too quickly
- An alternative approach is to have a fixed size pool of threads
- When connection is received pass the connection to the next available thread from pool
- When all the threads are busy, server blocks and does't accept more connections

4.6 Request and Response

- Design patterns for client server communications are very stereotyped.
- Can we automatically generate client server code?
- Yes!
- We can model a server as an object waiting for methods to be called
- Client then obtains reference to object and calls methods
- Distributed Object Systems and Remote Method Invocation (next lecture)

4.7 Conclusion

- Layering is a modularisation approach allowing services to improve upon the services offered by lower services.
- Protocol techniques qfrom lower layers are often re-used again and again (see the "End to End Argument" paper.
- Sockets encapsulate TCP/UDP endpoints and can be used to construct clients and servers.