FIT3142 Tutorial #1

Protocols and Inter-Process Communications

2.1 Question 1 (25%)

Explain the limitations of Transmission Control Protocol in a high throughput networking environment.

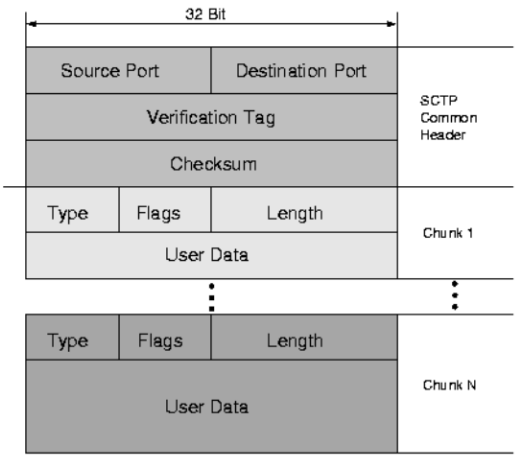
- High performance in carrying files and other bulk data was not a consideration when TCP and UDP were developed;
- TCP was developed to provide for “reliable” data transfers, ensuring that data (in packets) always arrived, and arrived in order, regardless of performance losses;
- UDP was developed to provide for “unreliable” data transfers, using a simpler protocol with slightly better performance than TCP, due to lesser overheads – applications had to manage reliability and in-order transfers;
- In a distributed computing environment, especially where high volumes of traffic must be carried, TCP and UDP are often too slow, as they were not optimised for high performance;
- GridFTP, SCTP, and Fast TCP are alternatives;

2.2 Question 2 (25%)

Compare the Stream Control Transmission Protocol and FastTCP.

- Stands for ‘Stream Control Transmission Protocol’
- Originally developed for Public Switched Telephone Networks (PSTNs) – to carry telephony signaling messages over IP networks;
- Is a reliable transport-layer protocol as an overlay on a connectionless protocol such as IP; Attempts to bridge the gap between TCP and UDP
- Is message-oriented
- Uses associations rather than connections
- Associations may have multiple parallel streams and are full-duplex
    - Reliable transfer of segments of data is separated from data delivery
        - Reliable transfer, unreliable connection.
- Supports multi-streaming
- Multi-streaming:
    - Provides for independent data delivery among streams
    - Thus reduces the risk of head-of-lineblocking among application objects
- Multi-homing:
    - Provides redundancy at path-level
- 4-way handshake rather than ‘3’

- Makes it resistant to SYN-based attacks



SCTP properties

- Slow Start and Congestion Avoidance phases like TCP:
  - Selective Acknowledgement
  - Fast Retransmit
- Provides partial data ordering
- Guarantees reliability in data transfer
- Improves on latency and throughput

SCTP Congestion Control

- Uses 3 parameters: rwnd, cwnd and ssthresh
- cwnd: depicts amount of data to be sent rather than which data to be sent
- SCTP performs congestion control when cwnd > ssthresh
- The cwnd size during congestion increases by 1 Message Transmission Unit/Round Trip Time

FastTCP

- Purpose: to overcome congestion control shortcomings of TCP
- Quicker response to network congestion
- Uses queuing delays and packet loss data to control congestion
- FastTCP congestion control mechanism is broken into 4 components:
  - Data Control
  - Window Control
  - Burstiness Control
  - Estimation

The above 4 components are an overlay on the TCP protocol processing layer

- Data control component determines which data packets need to be transmitted
- Window control determines how many packets to transmit – works on a Round Trip Time scale
- Burstiness control determines when to transmit packets – works on smaller time scales i.e. more frequently

- Estimation component provides information to assist the above components in decision making

Estimation Component:

  - Provides estimations of various input parameters to the other three decision-making components
  - When a positive ack is received for a packet, it calculates the RTT, and updates the average queuing delay and the minimum RTT
  - When a negative ack (NACK) is received, signaled by timeout or three repeated acks, it generates a loss value for this packet
- Window Control:
  - Determines the congestion window size based on feedback from Estimation component
  - Same algorithm for incrementing congestion window size is used each time, regardless of the state of the sender (unlike TCP)

2.3 Question 3 (25%)

Explain the differences between the three basic forms of Inter-Process Communications (IPC).

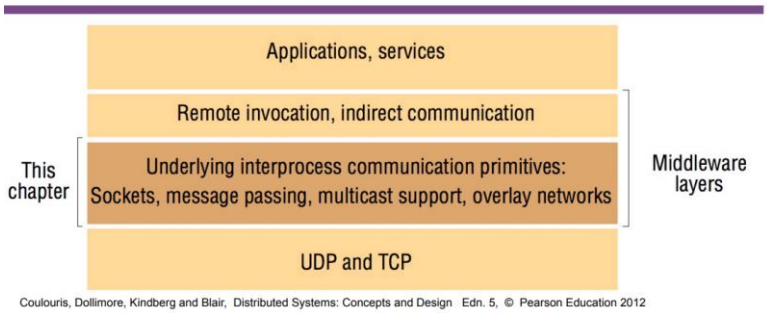
Which cannot be operated over a network, and why?

• IPC exists in many forms, but can be broadly divided into three categories:

1. Shared Memory IPC that relies on the Virtual Memory system to map pages or segments between processes;
2. Message Passing IPC in which the kernel carries discrete messages (or signals) between processes;
3. Stream Oriented IPC in which the kernel provides a stream channel between two processes with FIFO properties;

- Stream Oriented IPC is the model and the programming abstraction mostly employed in distributed systems;
- The two most commonly used stream oriented APIs are BSD Sockets and SVR4 STREAMs;

Stream Oriented IPC – Integration



Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5, © Pearson Education 2012

- **The "middleware" provides the API interface for the user program running in a process;**
- **The middleware interfaces to the protocol stack software, which is usually embedded in the operating system kernel;**

2.4 Question 4 (25%)

Explain the internal and functional differences between the Berkeley socket scheme, and the AT&T System V STREAMs scheme.

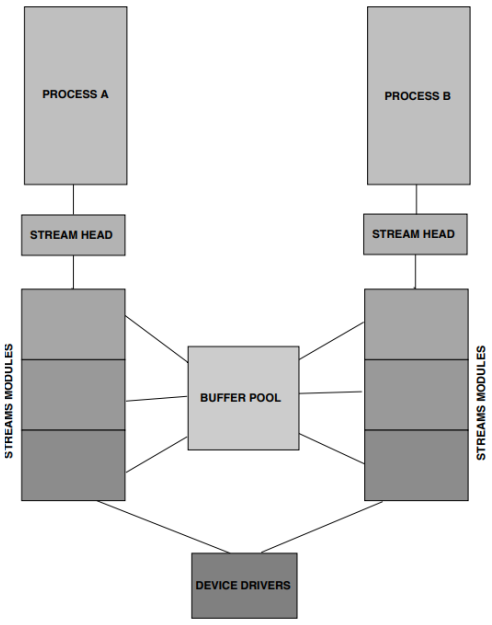


Fig.2.2 STREAMS Transport - Conceptual Model

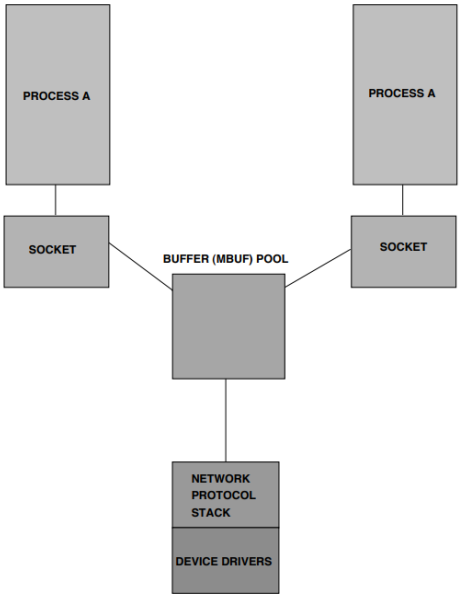


Fig 2.4 BSD Socket Transport - Conceptual Model

Berkeley Sockets

The BSD socket scheme evolved during the early eighties, in response to DARPA demand for efficient networked applications. The 4BSD release was largely funded by DARPA with the intention of providing a standard operating system for contractor research sites connected to ARPANET.

The socket interface is a far simpler scheme than System V Streams, and was designed with a focus on throughput performance, rather than architectural elegance and modularity. In practice this reflects in the fact that BSD based kernels have traditionally been slightly faster than System V kernels, running the same hardware.

Central design objectives of the BSD IPC scheme were:

- transparency - it should not matter whether two communicating processes are on the same platform, or different platforms.
- efficiency - IPC was layered on top of the network communications system, this was done to minimise the overhead associated with carrying IPC traffic between the IPC scheme and the networking scheme. While this cost modularity (as provided by Streams) it gained performance.
- compatibility - existing Unix applications used as character stream filters (grep, awk, sed, sort, uniq etc) should not require any changes to function in the new scheme.

A number of implementation issues arose during the development of the socket communications scheme, these were dealt with as follows:

Diverse communications protocols needed to be supported, so a communications domain scheme was devised. This allows the hiding of protocol specific idiosyncrasies in addressing and transport, in that the user needs to only specify the domain, eg UNIX or INTERNET.

An abstract object was needed as an endpoint to the connection. This object became the socket construct, which is dynamic ie it is created, used and destroyed on a connection by connection basis. A uniform scheme was required to describe the attributes associated with a connection. This was achieved by typing the sockets.

Sockets have a number of properties:

1. Delivering data in order
2. Delivery of data without duplication
3. Reliable delivery
4. Preservation of message boundaries
5. Support for out of band (OOB) messaging
6. Connection oriented communications

A pipe connection has for instance properties 1, 2 and 3. A datagram socket is unreliable, whereas a stream socket is reliable and may also carry OOB messages.

Sockets must have a naming scheme, so that processes can connect without having to know anything about each other.

The simplicity of the socket interface is a model of technical elegance, and is therefore very easy to use. The following example is for a client process.

Sockets are created with a socket() call, which returns a file descriptor number. This is analogous to the creation of a file, in that a memory datastructure has been allocated, via which read and write operations to the object may be carried out.

int socket(int domain, int type, int protocol);

where the domain may be one of the following:

- AF_UNIX (operating-system internal protocols),
- AF_INET (ARPA Internet protocols),
- AF_ISO (ISO protocols),
- AF_NS (Xerox Network Systems protocols), and
- AF_CCITT (ITU-T X.25 protocols).

the type one of the following:

- SOCK_STREAM (stream socket)
- SOCK_DGRAM (datagram socket)
- SOCK_RAW (raw socket)
- SOCK_SEQPACKET (sequenced stream socket)

The protocols argument allows choice, in some instances, of which protocol to use within the domain.

Once the socket has been created, it must be bound to a name, and a connection must be opened to allow the transmission of data.

*int bind(int socket, struct sockaddr *name, int namelen);*

The socket created by the socket() call exists in its protocol name space, but doesn't have a name assigned to it. The bind call assigns a unique name to the socket.

*int connect(int socket, struct sockaddr *name, int namelen);*

The connect call is then used by the client to open the transmission path to the server process.

Server process operation is slightly more complex, and involves listening and accepting connections. This is accomplished with the following calls:

int listen(int socket, int backlog);

The backlog argument specifies the number of pending connections which may be queued up to be accepted.

int accept(int socket, struct sockaddr *addr, int *addrlen);

The accept() call is somewhat more powerful than its peers, in that it will actually create a new file descriptor (socket) for each accepted connection. It returns the value of the new descriptor, but leaves the original socket open to accept further requests.

Once the connection is open, data may be read and written with conventional read(), write() calls as well as socket specific send(), recv() calls, the latter providing support for OOB messaging. Once the socket is no longer needed, it is destroyed with a close call.

As is clearly evident, the socket scheme is very simple to use, and is intentionally blended into the file descriptor scheme in use. Interestingly, the CSRG designers chose to produce a separate mechanism for opening the socket connection, to that used for file opening. The alternative is to overload the open system call to provide a common interface.

System V Streams

The Stream mechanism used by System V Unixes is one of the architecturally most elegant ideas within the original AT&T product. That elegance is however costly both in terms of complexity of use and CPU performance, in comparison with the simpler BSD socket scheme, and can become a significant performance factor where a system is running a large number of users over network connections.

Streams first appeared in System V Release 3.0, and are the brainchild of Dennis Ritchie. The central idea behind Streams is that of a modular interface for an I/O stream, which allows the stacking of multiple modules on a stream. These modules may implement communication protocols, serial I/O cooking (ie line disciplines) or multiplexing and demultiplexing of multiple I/O streams into a single channel. One nice way of looking at Streams is that every module is in effect a transform engine, which does a specific mapping of the input I/O stream into an output I/O stream.

The possibilities available to designer using Streams are very broad, and the design philosophy of Streams, if religiously adhered to, should allow considerable portability of streams module code. Space limitations preclude a more detailed discussion of the inner workings of Streams, and interfacing to Streams, and this will the subject of a future item. Interested readers are referred to Berny's Magic Garden Book.

Summary

Nearly all Unix implementations today support System V Shared Memory and both the BSD socket and System V Streams interfaces. In the latter instances, this however in itself says very little about the achievable throughput performance on either interface, as implementations can be quite different from one another.

A typical style of implementation on a contemporary generic SVR4 system will see sockets implemented by a socket library, which essentially piggybacks a socket connection on top of the OS native Streams IPC mechanism. From a performance perspective this is most unfortunate, as the computational overheads of Streams operations are further increased by the overheads of the socket interface layer above it. This is why networked applications written around sockets typically perform worse on generic SVR4 systems. One major Unix vendor is adding native socket support in its SVR4

kernel, simply to work around this limitation, but ultimately this problem will persist on most platforms until the established base of socket applications is ported across to Streams.

As is evident, IPC is not an aspect of application and OS integration to be trifled with, if performance is an issue. The naive view that the vendors of 4GL products, and the vendors of platforms have addressed all of the issues, is exactly that, naive. If system IPC performance is a serious issue, which it usually is in a client-server environment, this area must be on the checklist of items to be evaluated. Those who ignore it do so at their peril.