Network Games Infrastructure

Network Games

- Introduction
- Infrastructure
- Communication
- Threads

- What makes network games attractive?
  - Social
  - Interaction
  - Anonymous
    - can be someone else
  - Play forever
    - there is always somebody

Definition

- A network game is a that is played by two or more players at the same time where each player has a personal view of the game and interaction communication is handled by a third party.

- Examples are:
  - Chess by mail
  - War games

Characteristics

- Two or more players
- Usually asynchronous
- Simultaneous
- Uses 3rd party to communicate
- Each player has a personal view of the game
- Game mimics real-life
  - The game does not stop
  - Waiting for others

Issues

- Delays
  - Better communication
- Synchronization
  - Decide on game style
- Congestion
  - Scalability
  - Changing behaviour patterns
- Accuracy
  - deploy approximation/prediction techniques
- Boot strap
  - Determine entry point(s)

Network Games Types

- Token Based
  - Board games – Chess, Rummy
- Synchronous
  - Card games – poker, bridge
- Asynchronous
  - Racing game
  - War games
  - Most MMORPG

Network Games – technological Requirements

- Communication
  - Network
- Accurate board game
  - Visualization
  - Game status
- Interaction between players
  - Voice, text

Synchronous games

Pros
- Simple
- All players have same view
- Game state is preserved

Cons
- Wait for all players
- Slow
- Must communicate with all players all the time
- Harder to implement
  - Keep tab on players
- Poor fault tolerant

Asynchronous games

Pros
- Simple
- Fast
  - no waiting for input
  - Little overhead
- Communication is based on game events
- Easier to implement
- Better fault tolerant
  - Not impacted by missing players

Cons
- Game state is not preserved
- Player boards may not be correct
- Must know who is affected by recent changes

Communication needs

- Speed
- Reliability
- Ease of use
- Flexibility
- Scalability
- Concurrency
- …

HW – What Has Changed / Is Changing?

- Computation ability increases
  - Clock speed
  - # of transistors
  - Instruction based (concurrent processing)
  - Multi processors
- Affordability
  - Cheap
- Resources
  - More resources
  - More options for add on

HW – What Has Changed / Is Changing?

- Implications
  - Software cannot keep up
  - Backwards compatibility
  - Expensive to build special purpose HW
  - Life expectancy of HW is short – 2-7 years
- Gaming
  - Must work with what we have
  - Affects how we play
  - Creates hardship (e.g., delays)
  - Limits the imagination
  - Limits the available options

For example: Concurrency

- Is concurrency available?
  - Yes an no?
- Concurrency on the surface
  - Multicore (parallelization)
  - Multiprocessor
  - Serial communication
  - Can process one thing at a time

Network games and CS

- Benefits from other areas of CS
  - Parallel computing
  - Distributed computing
  - Protocol development
- There is a difference – the objectives
  - A program has a goal in mind – find protein structure, data mining, etc.
  - Games do not have a goal – the players do

Today's technologies

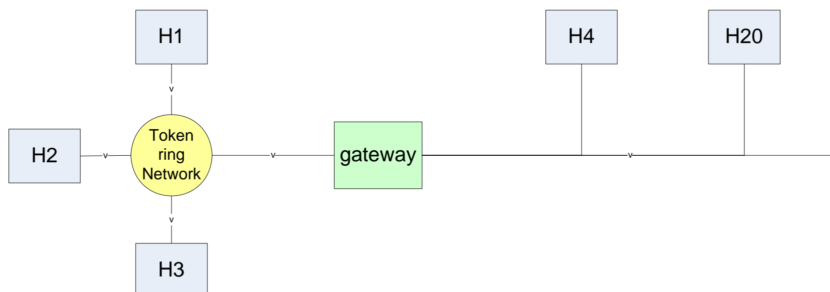
- SMP
  - Symmetric multi processing
  - Shared memory processing
- Cluster of computers
  - Workstations
  - Fast intercommunications
- Past technologies – can benefit from the logic
  - All to all
  - Ring (with/without chords)
  - Mesh/torus
  - hypercube

Computer Networks

- Relatively young technology
  - Started in the 70 with ARPA net
  - Still evolving
  - Dynamic
    - Wired – telephone, cable, fiber optics…
    - Wireless

- Standard technologies
  - Telephone lines - Modems
  - Local Area Networks (LAN) - Ethernet
  - Internet – Wide Area Network (WAN)

Network overview (last)



How to view a network?

- Model a network as a graph G=(V,E)
  – Nodes
  – Links
- Define the connectivity → interaction
- Define the topology → interaction

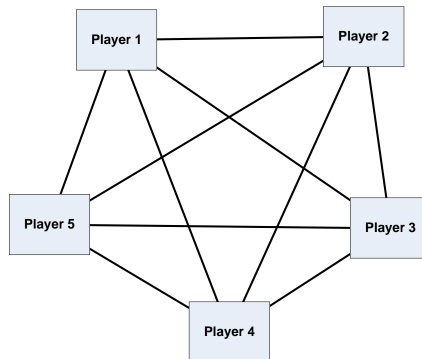
Game Architectures

Unstructured
  – Peer-to-peer

Structured
- Client/Server
  – One server per game
- Self server
  – A player is also a server
- Distributed servers
  – Multiple servers

Peer-to-Peer

- All to all topology
- All are equal
- Minor overhead
  - Implementation dependent



Peer-to-Peer

- Simple version (synchronous)
  - Each client transmits its state/changes to other
  - Wait until everyone receives the data
  - Proceed to next step

Peer-to-Peer – synchronous version

Pros

- Simple
- inexpensive
- Server not required
- Good for token-based games
- Good for low bandwidth NW
- Good for wireless?
  - within range

Cons

- Frame rate depends on
  - Slowest machine
  - Worst connection
- Hackable
- Not good for real-time games
- Bootstrap
- Many messages
- Bad scalability

Peer-to-Peer – asynchronous version

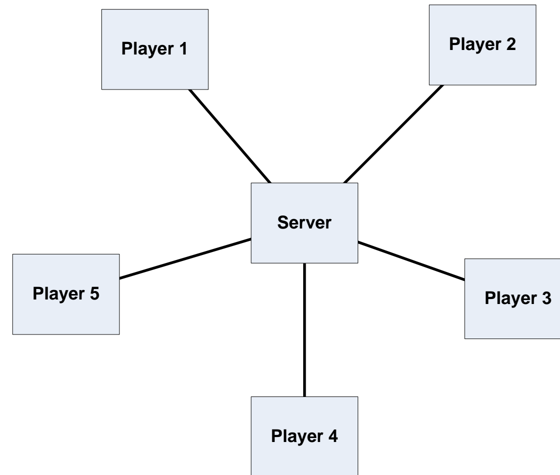
Pros

- Simple
- inexpensive
- Server not required
  - Including the Game's authors!
- Good for real-time games
  - Missing information not crucial

Cons

- Game can go out of synch
  - Slowest machine
  - Bad connection
- Hackable
- Not good for real-time games
- Bootstrap
- Many messages
- Poor scalability

Client Server



Client/Server

Pros
- Scalable
- Bootstrap
- Easy to control the game
  - Server has a global view of game
- Independent of players' HW
- Less hackable
- Better scalablity

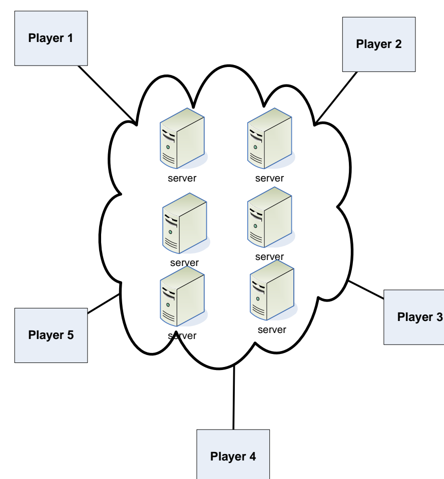
Cons
- Server must be provided
- Expensive
- Single point of failure
- Server must be
  - powerful
  - Excellent connectivity

Client as a Server Provider

- Peer-to-peer
- Server resolves the action
- One peer is the server

Multiple Server – Servers Farm

- Many machines coordinate service
- Used for large virtual worlds
- Player is assigned to a server

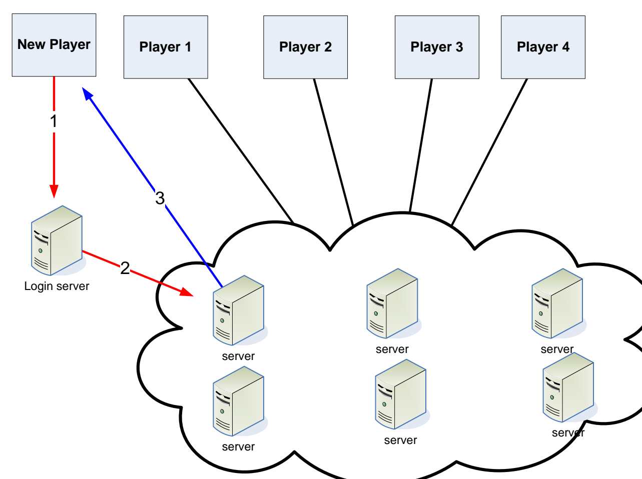


Multiple Server

Pros
- Many machines coordinate the game
- No single point of failure
- Load balancing
- Easy maintenance
- Used for large virtual worlds
- Easy maintenance
- Harder to hack
- Permits control of players

Cons
- Expensive
- Some synchronization may be required
- Run time load balancing may suffer
- bootstrap

Multiple Server

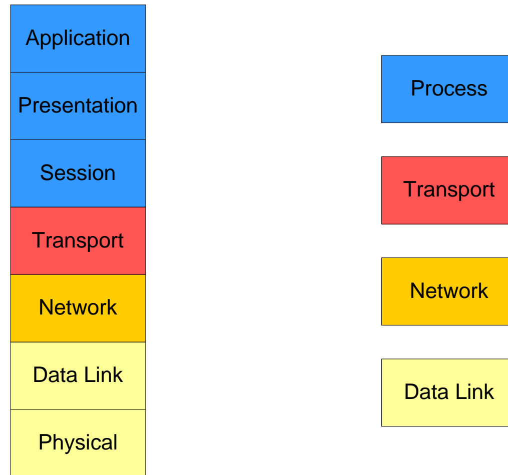


Networking – under the hood

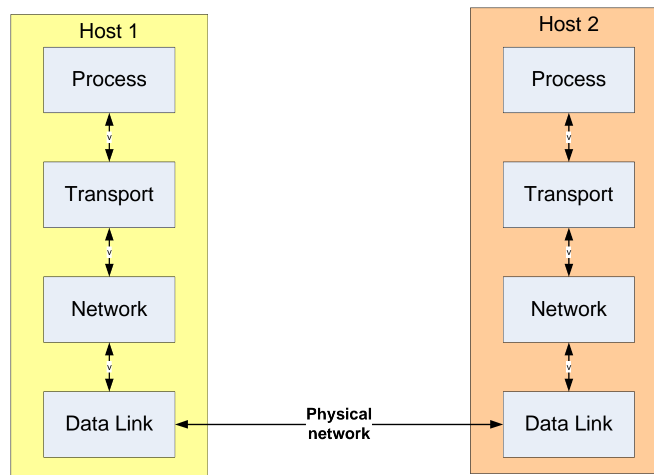
The OSI Model

- Developed in the late 70's
  - Framework for the development of standards.
  - Guidelines only (no specifications)

The OSI Model – 4 layer model

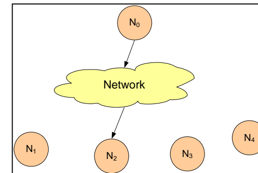


OSI Hierarchy

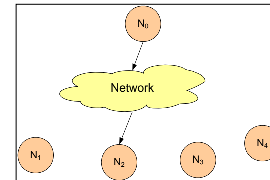


Messages

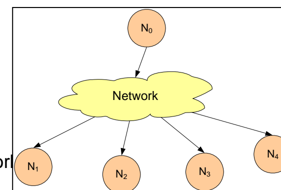
- Unicast
  - 1-1 – communication



- Multicast
  - 1-K – one to selected few



- Broadcast
  - 1- N - one to all

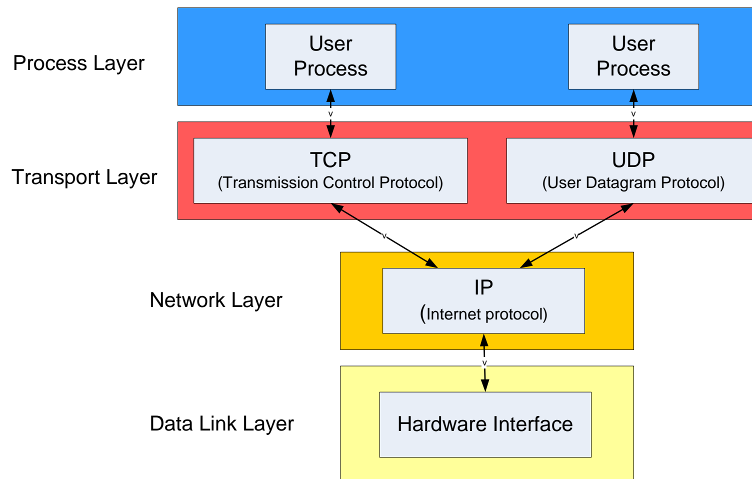


Protocol

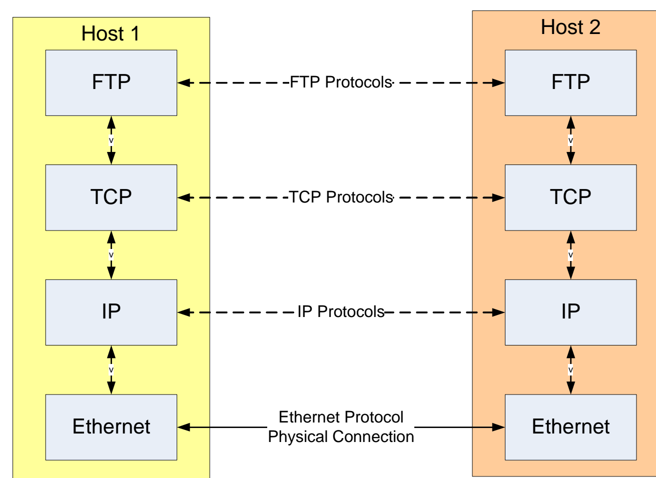
- Agreed Rules of communication between
  - Two applications or
  - Two instances of an application

- Consists of
  - Message format
  - Message Semantics
  - Error behaviour

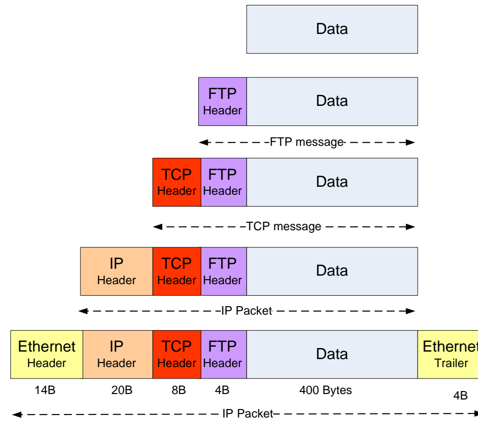
The TCP/IP protocol



The TCP/IP protocol



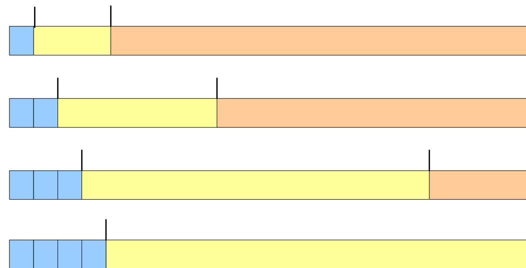
Data Encapsulation



- Network router
  - 100Mb/s
- Message
  - 450 bytes = 3,600 bits
- Transmission time
  - 100m/3600 = 0.036ms
  - Send to 100 users = 0.36ms
- Message
  - 1550 bytes = 12,400 bits
- Transmission time
  - 100m/12400 = 0.12ms
  - Send to 100 users = 12ms

IP Address

- IP address is a 32 bit address
  - Unique to each host within its network
- Four classes of addresses
  - Specify the network
  - Specify the host



Ports

- Issues
  - How can a server handle many clients?
  - How can a client connect/talk to different services on the same host?
- Solution
  - Provide an application (client or server) with a special "mail box" – a **port**

Ports

- A port is a a 16 bit unique identifier
  - No two applications should use the same port on the same host
- Reserved ports
  - Some ports are reserved
    - Port 80 for http://
    - Port 21 for FTP
  - Well known applications – 1-1023
  - Registered ports – 1024 – 49151
  - Dynamic ports – 49152-65535
- Other systems may use different ranges

Putting it all together

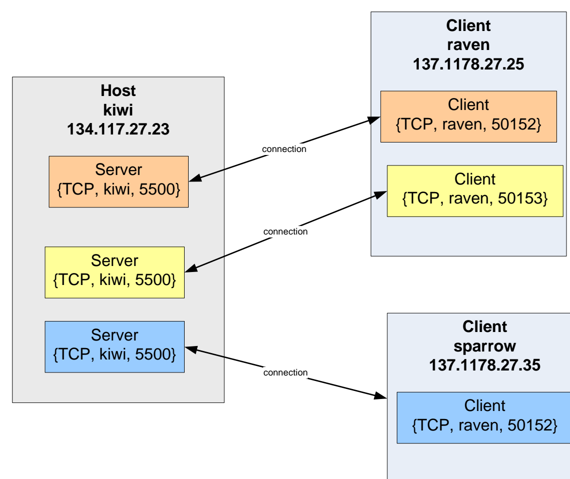
- Each interaction can be viewed as a 5 touple

{Protocol, source add., source process, dest. add., dest. process}

{Protocol, source add., source port, dest. add., dest. port}

{TCP, 134.117.27.23, 5500, 134.117.27.25, 1622}

Putting it all together



Internet User Protocols

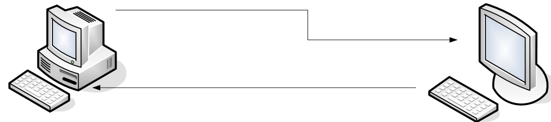
- TCP
  - Connection based
  - Reliable
  - Bytes arrive in order they were sent
  - Collects small packets and transmits them together
  - Stream of bytes
- UDP
  - Connectionless
  - Unreliable
  - Arbitrary arrival order

TCP

- Reliable stream of bytes
  - Implies the need for a “connection”
- Connection sets up data structures
  - Hold incoming packets
  - Hold outgoing packets
  - Handle retransmits

TCP Reliability

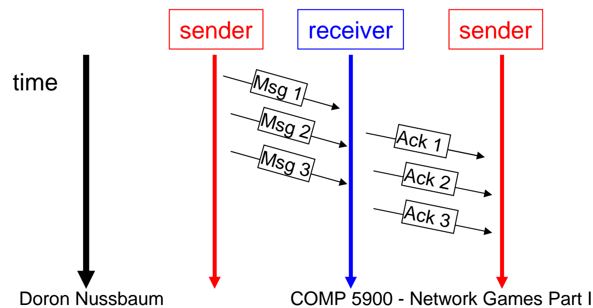
- Each data requires –
  - send-Receive-Acknowledge



- Sender retains data until an **ACK** is received
- If an ACK was not received → Sender retransmits data

TCP

- Send-Receive-Ack round trip is expensive (time)
- Solution: overlaying the send and ack messages
- Side effect – more work
  - Ensure that messages are received
  - Maintain a queue in sender and receiver



Send

TCP Sender Queue

- Sends data and insert it into a send queue
- Sets a timer on this queue item
- If timer expires, and no ack then
  - Re-send data
  - Set a new timer (longer)

- If ack is received then
  - If the corresponding queue item is the oldest,
    - Free the slot for new data
  - Otherwise mark as received
- If no queue space avail
  - sender **waits!**

TCP Receiver Queue

- Send an ACK for each received packet
- If the packet is the next in the sequence
  - Forward packet to application
- Otherwise keep packet in queue

TCP – communication behaviour

- May halt the transmission temporarily
  - (e.g., when no ack messages are received)
- Attempt to re-establish the communication
  - Sends test message(s)
  - Slowly increases the transmission rate
- Transmission rate changes
  - Cut transmission rate by half each time
  - Increase transmission rate by a constant

TCP Wrap-up

- Connection based
  - Reliable arrival: Retransmit
  - Reliable order: Sequence numbers
- TCP can minimize the overhead of small data
  - buffers up data on 200ms intervals
- TCP Has an "emergency" channel
  - OOB Out Of Band

UDP

- Connectionless!
  - Does not worry about the data
- Unreliable transmission
  - Lost packets are lost forever
  - Must be handled outside of protocol
- Race condition
  - Arrival order may not match transmission order
  - Must be handled outside of protocol
- Fast
  - When data is received the application received it

UDP

- Losing Packets!
  - No knowledge if and what was lost - critical data!!
  - Must be self managed
- Arrival order may be time sensitive
  - Past location may be meaningless
  - Mixing up order can be critical – car racing

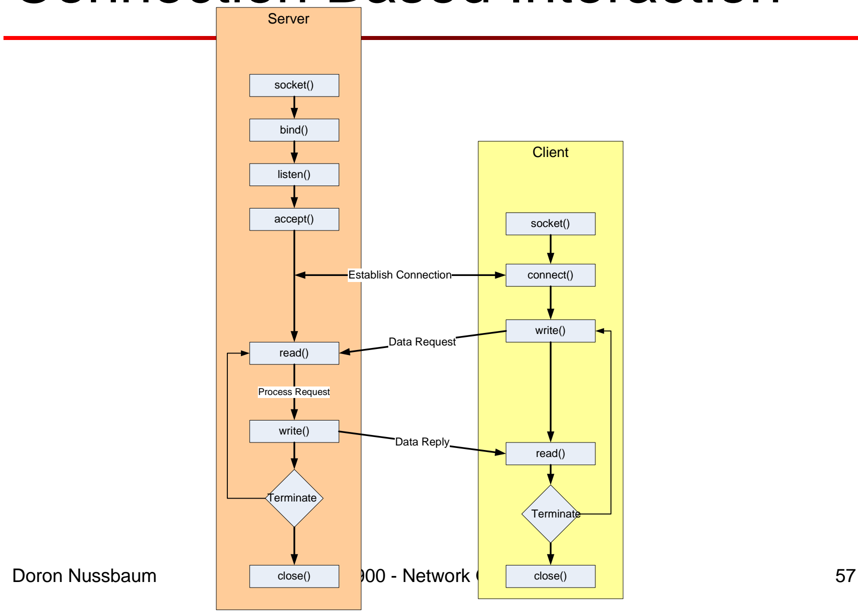
Sockets

- An interface/library that implements the TCP/IP protocol
- Consists of a set of functions that enable communication between two or more hosts over a network.
- Resemble file I/O

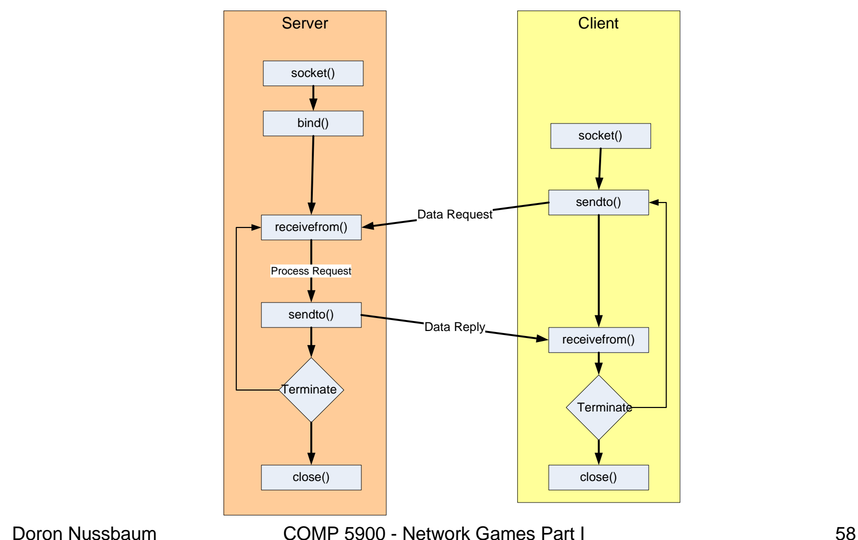
Server
- Initialize Winsock.
- Create a socket.
- Bind the socket.
- Listen on the socket for a client.
- Accept a connection from a client.
- Receive and send data.
- Disconnect.

Client
- Initialize Winsock.
- Create a socket.
- Connect to the server.
- Send and receive data.
- Disconnect.

Connection Based Interaction



Connectionless Based Interaction



IP Strucutres

- IPV4 and IPV6 – will use IPV4

```
struct sockaddr {
   short sa_family;
   char sa_data[14];
}
```

The family of communication protocol (Use AF_xxxx) AF_INET

Size of data depends on the protocol

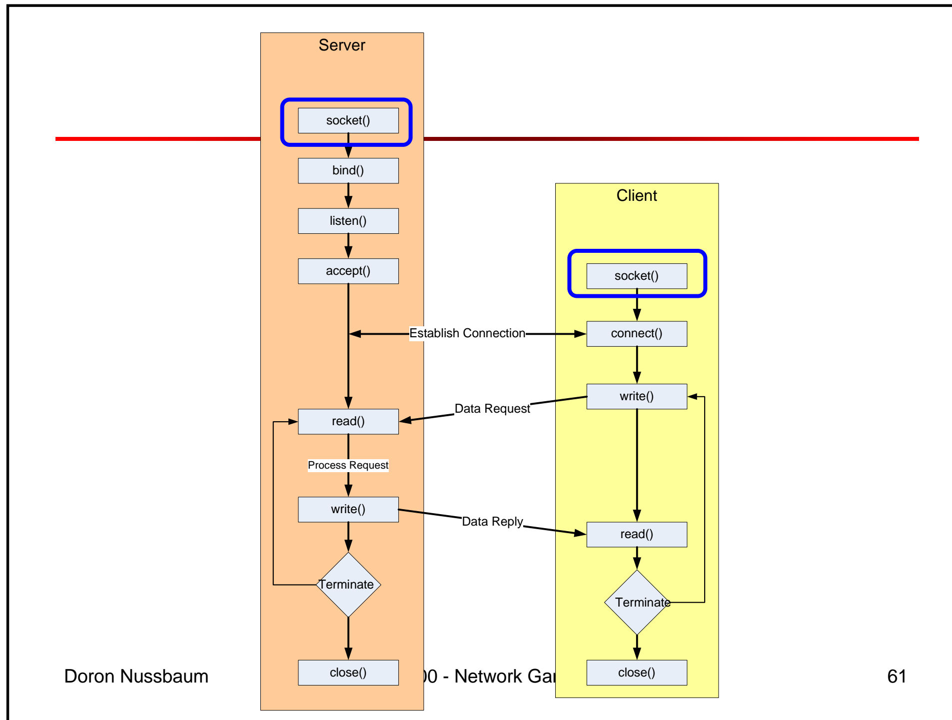
Internet Family Structure

```
struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

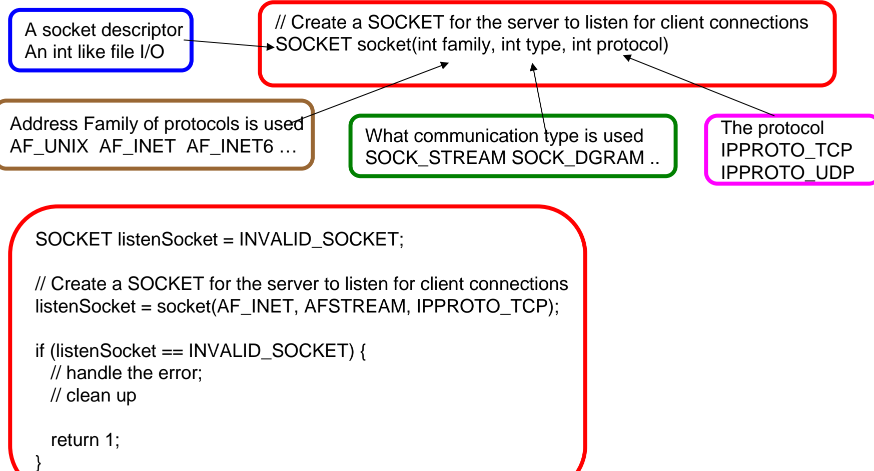
```
struct in_addr {
   u_long address;
}
```

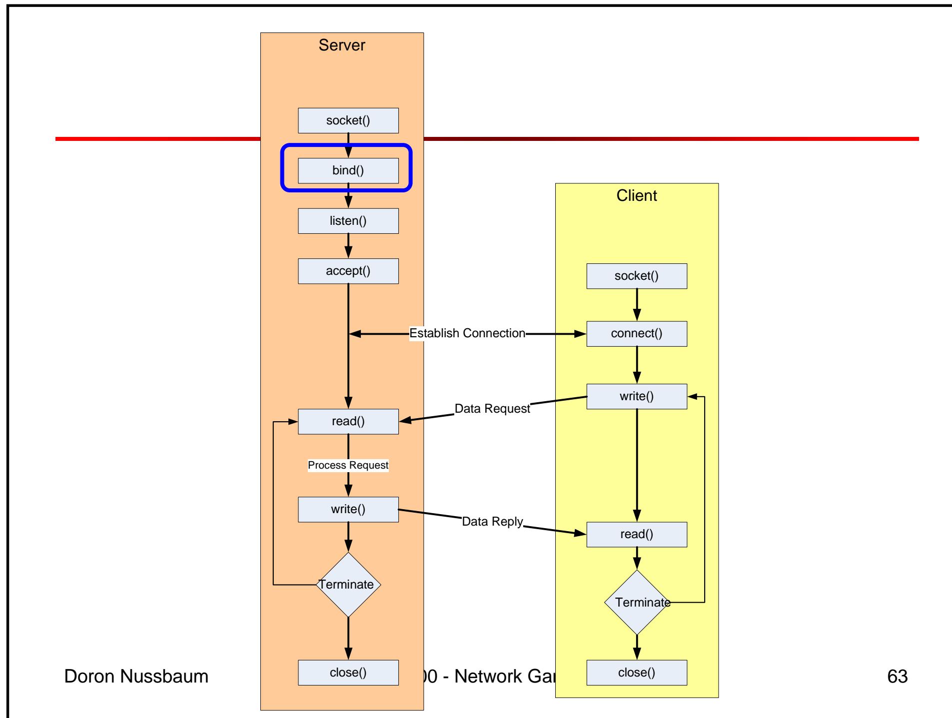


```
struct addrinfo *servAdd = NULL,hints;

ZeroMemory(&hints, sizeof (hints));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
hints.ai_protocol = IPPROTO_TCP;
hints.ai_flags = AI_PASSIVE;

// Resolve the local address and port to be used by the server
rc = getaddrinfo(NULL, DEFAULT_PORT, &hints, &servAdd);

if (rc != 0) {
   printf("getaddrinfo failed: %d\n", iResult);
   WSACleanup();
   return 1;
}
```



Creating a socket





Bind

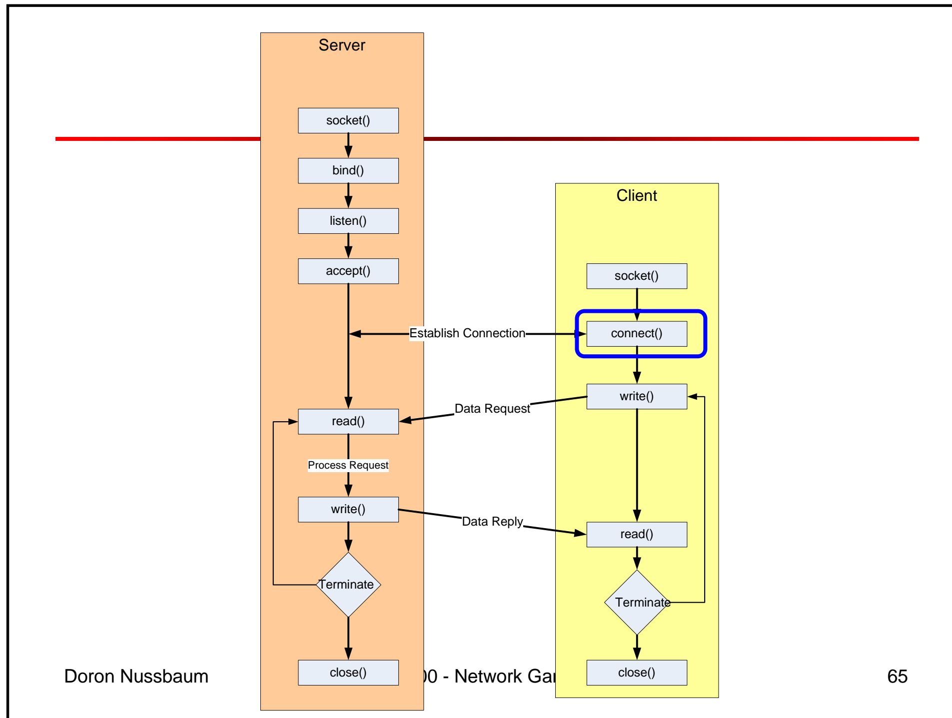
- Purpose: the server informs the system what is its address. (All messages to this address are to be delivered to the server.)

```
// Setup the TCP listening socket

int  bind( SOCKET socket, struct sockaddr *myAddr, int addressLenght);
```

Internet Family Structure

```
struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

```
// Setup the TCP listening socket

Rc = bind( listenSocket, myAddress, addrLength);
if (rc == SOCKET_ERROR) {
    // error clean up

}
```



Connect

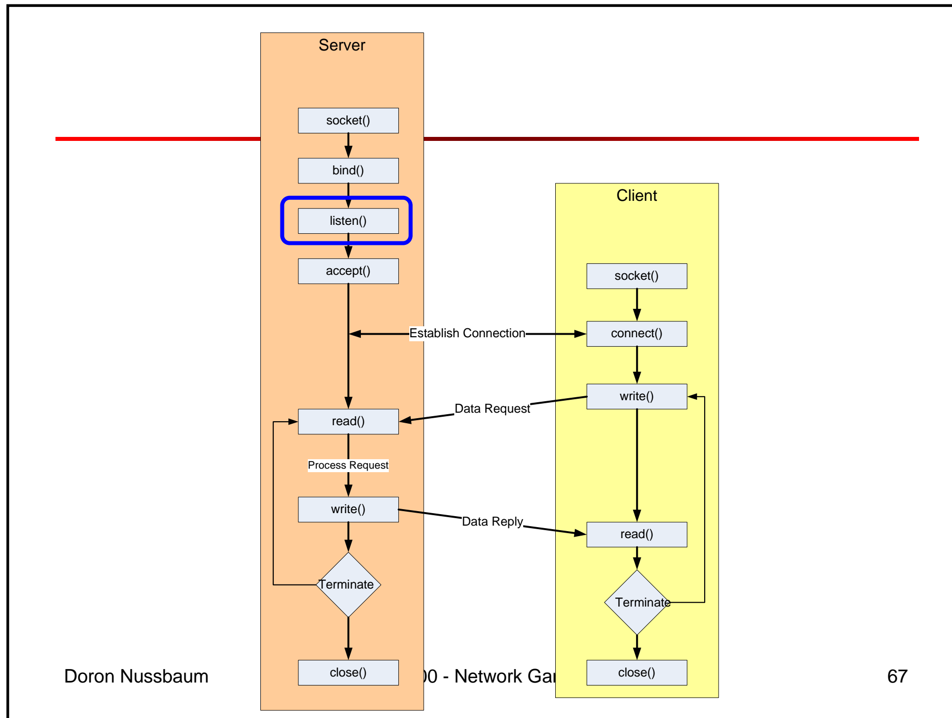
- Purpose: establishes a connection with the server.
- For connectionless the call is not required.
  - However it stores the addresss of the server for futrue calls to read, write, recev and send.

```
// Setup the TCP listening socket

int  connect( SOCKET socket, struct sockaddr *serverAddr, int addressLenght);
```

```
// connecting to the server

rc = connect( mySocket, ServerAdd, addrLength);
if (rc == SOCKET_ERROR) {
     // error clean up

}
```

Internet Family Structure

```
struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

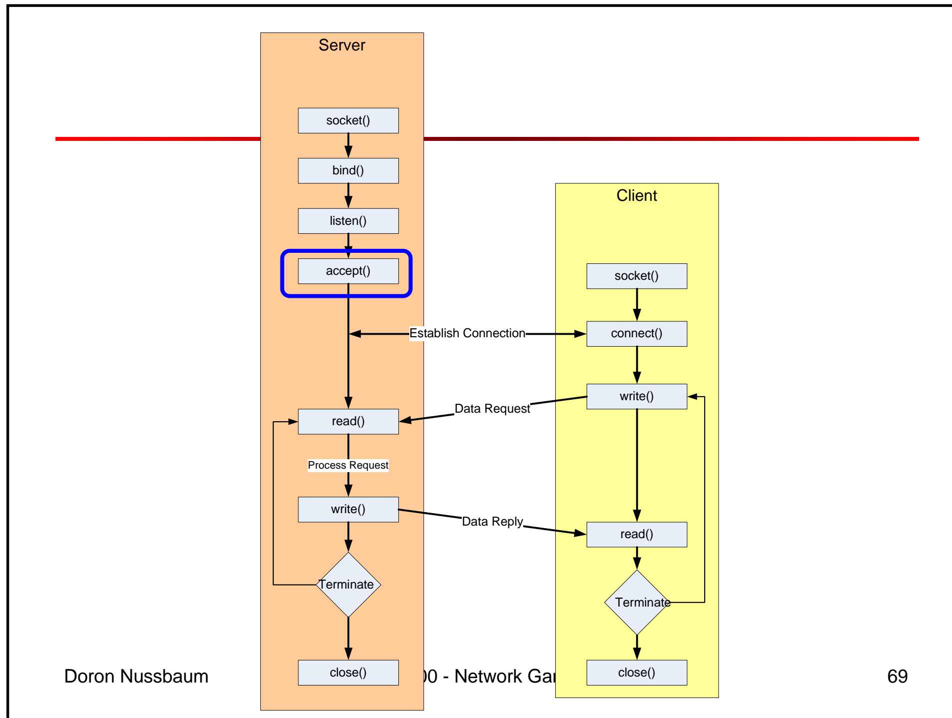


Listen

- Purpose: tells the system how many connection requests can be queued

```
// Setup the TCP listening socket

int  connect( SOCKET socket, int backlog);
```

```
rc = listen( listenSocket, 5) ;

if (rc == SOCKET_ERROR )  {
   // error clean up
}
```

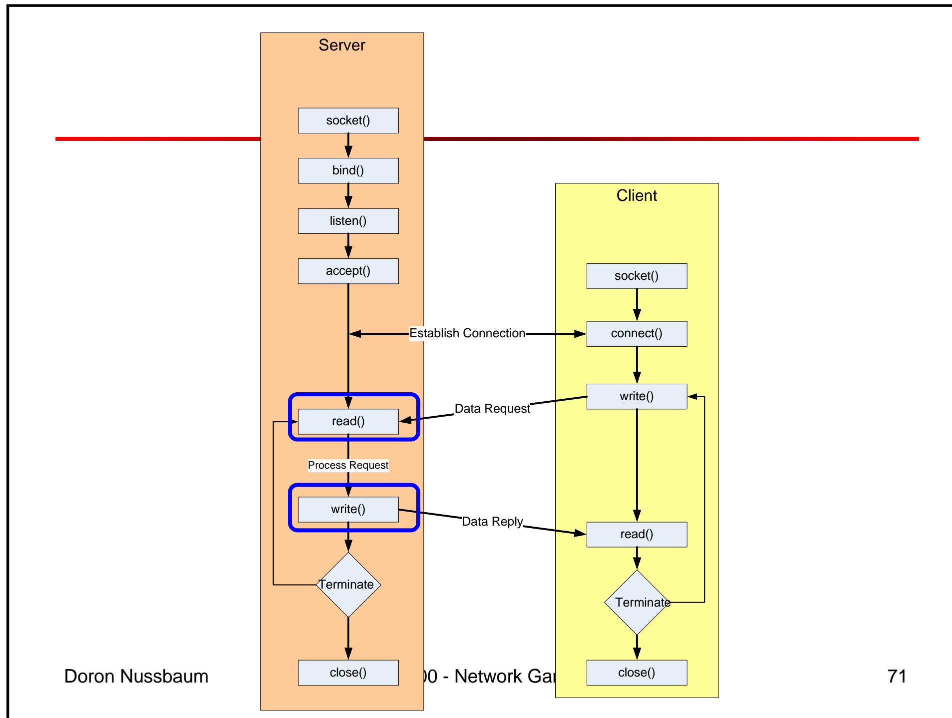


Accept

- Purpose: accepts a connection
  - This is a blocking call

```
// Setup the TCP listening socket

SOCKET accept(SOCKET listeSocket, struct sockaddr *clientAddress, int *addrlen);
```

```
clientSocket = INVALID_SOCKET;

// Accept a client socket
clientSocket = accept(ListenSocket, NULL, NULL);

if (ClientSocket == INVALID_SOCKET) {
   // error clean up
}
```



Send and Recv

- Purpose: read data from the client and send data to the client
  - This is a blocking call

Number of bytes that were sent or received

If 0 then connection was closed

```
// Sending data over the connection
int send (SOCKET clientSocket, char *buf, int bufSize, int flags);

// receiving data over the connection
int recv (SOCKET clientSocket, char *buf, int bufSize, int flags);
```

```
// Flags
MSG_OOB           send or receive data out-of-band
MSG_PEEK          peek at incoming message (recv, recvfrom
MSG_DONOTROUTE    bypass routing (send or sendto
```

Sendto and Recvfrom

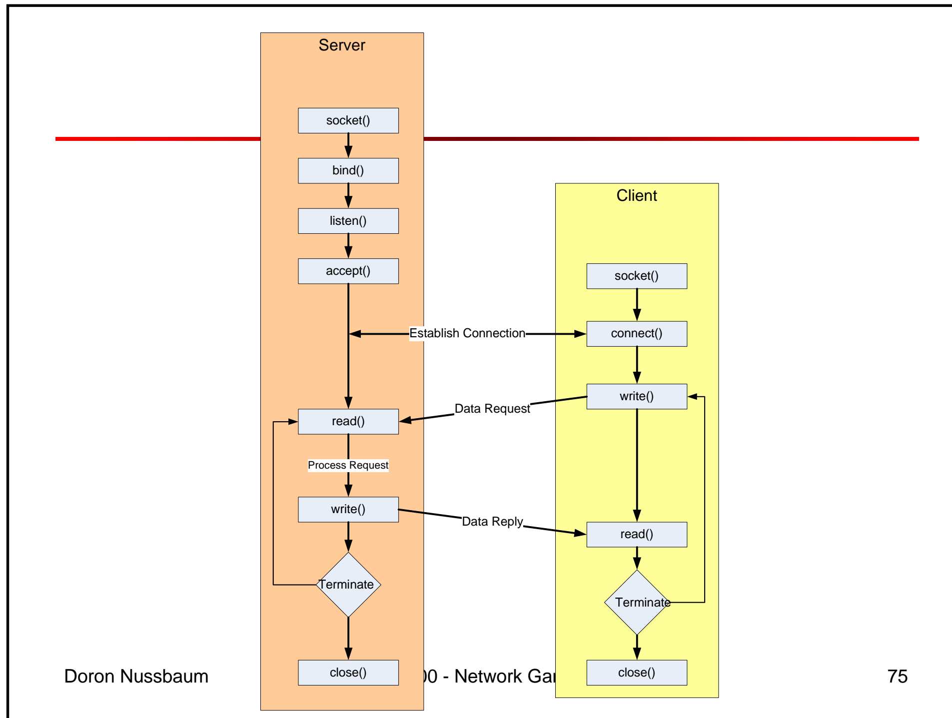
- Purpose: read data from the client and send data to the client
  - This is a blocking call
  - connectionless

```
// Sending data over the connection
int sendto(SOCKET clientSocket, char *buf, int bufSize, int flags, struct sockaddr *to, int addrelen);
```

```
// Sending data over the connection
int receivefrom(SOCKET clientSocket, char *buf, int bufSize, int flags, struct sockaddr *to, int *addrelen);
```


Send and Recv

```
#define DEFAULT_BUFLEN 512
char recvbuf[DEFAULT_BUFLEN];
int rc, iSendResult;

// Receive until the peer shuts down the connection
do {
  rc = recv(ClientSocket, recvbuf, DEFAULT_BUFLEN, 0);
  if (rc > 0) {
     // Echo the buffer back to the sender
    rc = send(ClientSocket, recvbuf, rc, 0);
    if (iSendResult == SOCKET_ERROR) {
      // error
    }
    printf("Bytes sent: %d\n", rc);
  }  else if (rc == 0) {
    printf("Connection closing...\n");  {
  } else {
    // error
  }
} while (iResult > 0);
```



Close the connection

- Purpose: close the socket

```
// close the socket
int closesocket (SOCKET clientSocket);
```

Unix call is close();

```
// allows graceful termination of connection

int shutdown SOCKET clientSocket, int howTo);
```

Flags determine how to shut down
SD_RECEIVE 0
SD_SEND 1
SD_BOTH 2

Utility Functions

The **htonl** function converts a **u_long** from host to TCP/IP network byte order (which is big endian).
u_long htonl(u_long hostlong);

The **htons** function converts a **u_short** from host to TCP/IP network byte order (which is big-endian)
u_short htons(u_short hostshort);

The **ntohl** function converts a **u_long** from TCP/IP network order to host byte order
(which is little-endian on Intel processors).
u_long ntohl(u_long netlong);

The **ntohs** function converts a **u_short** from TCP/IP network byte order to host byte order
(which is little-endian on Intel processors).
u_short ntohs(u_short netshort);

The **inet_addr** function converts a string containing an IPv4 dotted-decimal address into a proper address for the **IN_ADDR** structure.
unsigned long inet_addr(char *cp);

The **inet_ntoa** function converts an (Ipv4) Internet network address into an ASCII string in Internet standard dotted-decimal format.
char* inet_ntoa(struct in_addr in);

Windows specific for sockets

```
    // Start Winsock up
    WSAData wsaData;
    if ((rc = WSAStartup(MAKEWORD(2,2), &wsaData)) != 0) {
         cerr << "WSAStartup() returned error code " << rc << "." << endl;
       return(1);
    }
.
.
program
.
.


WSACleanup();
```

Questions?