Game Engine Programming

GMT Master Program
Utrecht University

Dr. Nicolas Pronost

Course code: INFOMGEP
Credits: 7.5 ECTS

Lecture #13

Game network programming

Introduction

- We have seen that mechanisms can be used to communicate between classes
  - *e.g.* Listener and Event design patterns
- But how to communicate between classes on different machines?
  - Use of common protocols to send / receive packets (data)
- Multi-player games use intensively multi-machine exchanges
  - local (LAN parties) or global (MMOG) network

Few words about the Internet

- The Internet is a packet-switched, fault-tolerant network
  - information is broken into small packets (B / kB)
  - sent from A to B by traversing a web-like server structure (cyberspace)
  - using different paths that adapt to network circumstances, errors, server malfunctions *etc.*
  - packets do not necessarily arrived in the correct order, they need to be identified (labeled or numbered)

Packet vs. Circuit

- In circuit-based networks
  - we own the communication circuit from origin to destination
  - access is exclusive
  - the path from A to B is unique
  - information is sent as a single block
  - used
    - in traditional land telephone system
    - in small / medium scale multi-player games (LAN)

Few words about the Internet

- Two tasks take place
  - Data is fragmented at one end and reassembled at the other end
  - Individual packets are routed through the network
- Performed in parallel by two protocols
  - Transmission Control Protocol (TCP) is the data separator and assembler
  - Internet Protocol (IP) takes care of the routing

TCP/IP

- Recommended for traditional networking
  - guarantees FIFO (buffer) operations (data arrive in the correct order) and ensures that the data sent reach their destination
  - as it allows to detect lost packets, and re-request them
  - but this protocol is slow
    - wait to receive all packets in the correct order to rebuild the initial sequence
    - secure transmission at the cost of reduced performance

UDP

- User Datagram Protocol (UDP)
  - sacrifices slowest features for speed
  - by sending fixed-size data packages
  - does not require an active connection (connectionless protocol)
  - lost packets are not recovered
  - FIFO is not guaranteed

TCP vs. UDP

- Usage will depend on the game-play
- Examples
  - Strategy game where lag is acceptable but each move (game round, order) is crucial => TCP
  - FPS with less lag as possible and exchanged data can be lost / predicted => UDP

Sockets

- Game programmers do not want to deal directly with TCP, UDP and IP
  - complex networking all over the world
  - no manual breaking of data into pieces
- They want to access the network like a local file
  - open distant site, read from it, write to it ...
- Abstraction layer: the socket interface

Sockets

- Input/output device to open a communication pipeline between two sites
  - to transfer information, both sites need an open socket aimed at the other
  - data exchange consists in writing and reading to/from the socket
  - establishing the socket is the most difficult part
- Operate in TCP or UDP modes
  - most internal differences hidden

Servers and clients

- A client application is the endpoint of the communications network
  - connected to one server
  - consumes data transferred from the server
  - can also send data to server
  - examples:
    - web browser (doing requests)
    - MMO game (retrieving data about the world and updating the server with current player state)

Servers and clients

- A server is connected to several clients
  - acts as a data provider for clients
  - manages the incoming connections
  - examples:
    - web server (such as Google search or Facebook)
    - MMO game server (dispatching world information, players joining and quitting the game, lost of connection *etc.*)

Sockets

- Socket on Windows: winsock

```
#include <winsock2.h> // contains basic socket functions and structures
#include <ws2tcpip.h> // advanced functions to retrieve IP@
```

  - processes that use winsock must initialize the Windows Socket API (WSA) first

```
WSADATA wsaData;

// Initialize winsock
int result = WSAStartup(MAKEWORD(2,2), &wsaData); // request v2.2
if (result != 0) {
    cout << "WSAStartup failed: " << result << endl;
    exit(1);
}
```

  - reference to library Ws2_32.lib has to be added

Universiteit Utrecht

Sockets

- TCP client
- UDP client
- TCP server
- UDP server

TCP client

- A simple TCP client consists of
  - Connection to the (game) server
  - Writing of data to the server
  - Reading of data from the server
  - Closing of the connection when finished
- Use one single socket interface to communicate with the server
  - server address has to be known

TCP client

- To establish the client connection using the required connection information: IP, address and port

```
int sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

  - creates a socket
  - AF_INET for Internet connection (AF_UNIX for communication within a single computer, AF_BTH for Bluetooth address family *etc.*)
  - SOCK_STREAM as we want stream-based communications (else SOCK_DGRAM)
  - IPPROTO_TCP as we want TCP as transport protocol (else IPPROTO_UDP)
  - returns a file descriptor if success, INVALID_SOCKET otherwise

TCP client

- Once created, we need the server info
  - servers are usually referenced with DNS address (*e.g.* gameserver.hostname.com)
  - we need to convert it into a numeric IP address

```
int getaddrinfo (
    char* serverDNS,                    // server name
    char* port,                         // port number
    const addrinfo* hints,              // caller type of socket
    addrinfo* result                    // response from the host
);
```

  - returns zero if success, non-zero value otherwise

TCP client

- Hints and result values are stored in an addrinfo structure

```
struct addrinfo {
   int   ai_flags;          // options (AI_PASSIVE, AI_SECURE ...)
   int   ai_family;         // network family (AF_INET)
   int   ai_socktype;       // socket type (SOCK_STREAM, SOCK_DGRAM ...)
   int   ai_protocol;       // protocol (IPPROTO_TCP, IPPROTO_UDP)
   size_t ai_addrlen;       // size of the buffer ai_addr
   char * ai_canonname;     // canonical name of the host (DNS)
   struct sockaddr * ai_addr;      // pointer to the socket structure
   struct addrinfo * ai_next;      // pointer to the next structure
};
```

TCP client

- Complete the connection using created socket and resulting connection data

```
int error = connect(sock, result->ai_addr, result->ai_addrlen);
```

  - returns zero if success, SOCKET_ERROR otherwise

TCP client

- Full client-side connection function

```
int ConnectTCP (char *host, char *port) {
    // ... assuming  WSAStartup ...
    // initialize TCP connection information
    struct addrinfo * result = NULL, hints;
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_protocol = IPPROTO_TCP;

    // resolve server address and port
    int error = getaddrinfo(host, port, &hints, &result);
    if (error != 0) return error;

    // create socket for connecting to server
    int sock = socket(result->ai_family, result->ai_socktype, result->ai_protocol);
    if (sock == INVALID_SOCKET) return sock;

    // connect to server
    error = connect(sock, result->ai_addr, result->ai_addrlen);
    if (error == SOCKET_ERROR) return error;
    else return sock;
}
```

TCP client

- Data transfer - Reading from the socket

```
int result = recv(SOCKET sock, char* buffer, int size, int flag);
```

  - sock: the socket
  - buffer: buffer where to store the data (memory must be allocated)
  - size: length of buffer in bytes
  - flag: reading options (usually 0)
  - returns the number of bytes received
  - remains blocked as long as required number of bytes not read

TCP client

- Data transfer - Reading from the socket

```
// assuming creation and connection of socket sock

const int recvbuflen = 512;
char recvbuff[recvbuflen];
int result;

do {
    result = recv(sock, recvbuff, recvbuflen, 0);

    if (result > 0) {
          // do something with the data
    }
    else if (result == 0)
          cout << "Connection closed" << endl;
    else
          cout << "Receive failed: error #" << WSAGetLastError() << endl;
}
while (result > 0);
```

TCP client

- Data transfer - Writing to the socket

```
int result = send(
   SOCKET sock, const char* buffer,
   int strlenbuff, int flag
);
```

  - same parameters as reading
  - returns number of bytes sent if success, SOCKET_ERROR otherwise

TCP client

- Data transfer - Writing to the socket

```
// assuming creation and connection of socket sock

char * sendbuf = “Data send by client”;

int result = send(sock, sendbuf, (int)strlen(sendbuf), 0);

if (result == SOCKET_ERROR)
   cout << “Send failed : error #” << WSAGetLastError() << endl;
```

TCP client

- Closing socket
  - close the sending side connection when no more data has to be sent

```
int result = shutdown(sock, SD_SEND);
```

  - close the connection (both sides if no shutdown first)

```
closesocket(sock);
```

  - clean the Windows sockets API

```
WSACleanup();
```

Data exchange

```
float x; // to send and receive
```

- Pointer cast

```
int result = send(sock, (const char *) &x, sizeof(float), 0);
```

- Convert data into char

```
ostringstream os_data; os_data << x;
string s_data = os_data.str();
const char * buffer = s_data.c_str();
int result = send(sock, buffer, (int)strlen(buffer), 0);
```

- Structure

```
struct sMsg {
   char type;
   char data[512]; // or more specific like float
}
// ... construction of sMsg object msg ...
int result = send(sock, (const char *) &msg, sizeof(sMsg), 0);
```

UDP client

- Easier than TCP
  - no explicit connection declaration and closure
  - 1 connection creation function, 1 send function, 1 receive function
- Creation function

```
int sock = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
```

UDP client

- Data transfer - Write to socket

```
void sendUDP (char *msg, char *host, char *port, int socket) {

    struct addrinfo * result = NULL, hints;
    ZeroMemory(&hints, sizeof(hints));
    hints.ai_family = AF_INET;
    hints.ai_socktype = SOCK_DGRAM;
    hints.ai_protocol = IPPROTO_UDP;
    int error = getaddrinfo(host, port, &hints, &result);

    sendto(sock,
           msg, strlen(msg), 0,
           result->ai_addr, result->ai_addrlen);
}
```

UDP client

- Data transfer - Reading from socket

```
int recvfrom (int socket, char *buffer, int buflen, int flags,
              sockaddr *from, int fromlen);
```

  - needs *from* parameter to get information on the server sending the data
  - we can use a single socket to receive data from several connectionless servers

UDP client

- We access the server name at each sending call, not efficient
- 2 solutions
  - to store the result structure outside the function
  - to use connected UDP (vs. connectionless)
    - used when the client has only one (game) server
    1. create a datagram socket with UDP
    2. use connect call with server
    3. use regular send/recv function instead of sendto/recvfrom
    4. close when finished

TCP server

- Servers must be able to exchange information with many clients at once
  - sequential scan of open sockets
  - concurrent server running parallel processes dedicated to one socket and client
- 2 architectures
  - single-peer server (two-player games)
  - multiple-peer server (multi-player games)

Single-peer TCP server

- One-to-one situation
- Server/client relationship is not symmetrical
- They play different roles and have different calls
- The server uses two sockets
  - the listen socket
    - the server creates its own socket and puts it in listening mode
  - the client socket
    - the server uses it to communicate with the client

Single-peer TCP server

- Listen socket creation

```
int listenSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

- Establishment of a relationship between the socket and an IP address and communication port
  - the server will look for connections through that IP/port

```
struct addrinfo * result = 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; // indicates future use in a blind
int error = getaddrinfo(NULL, port, &hints, &result);

bind(listenSocket, result->ai_addr, result->ai_addrlen);
// associate the local address with the socket
```

Single-peer TCP server

- Place the socket in listening mode, waiting for incoming connections

```
int listen(int listenSocket, int queuelen);
```

  - queuelen specifies the length of the connection queue (to prevent new requests to be lost during the treatment of the current request), usually up to 5

```
int result = listen(listenSocket,5);

if (result == SOCKET_ERROR)
   cout << "Listen failed : error #" << WSAGetLastError() << endl;
```

Single-peer TCP server

- The server will permit incoming connections attempted on the listenSocket

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

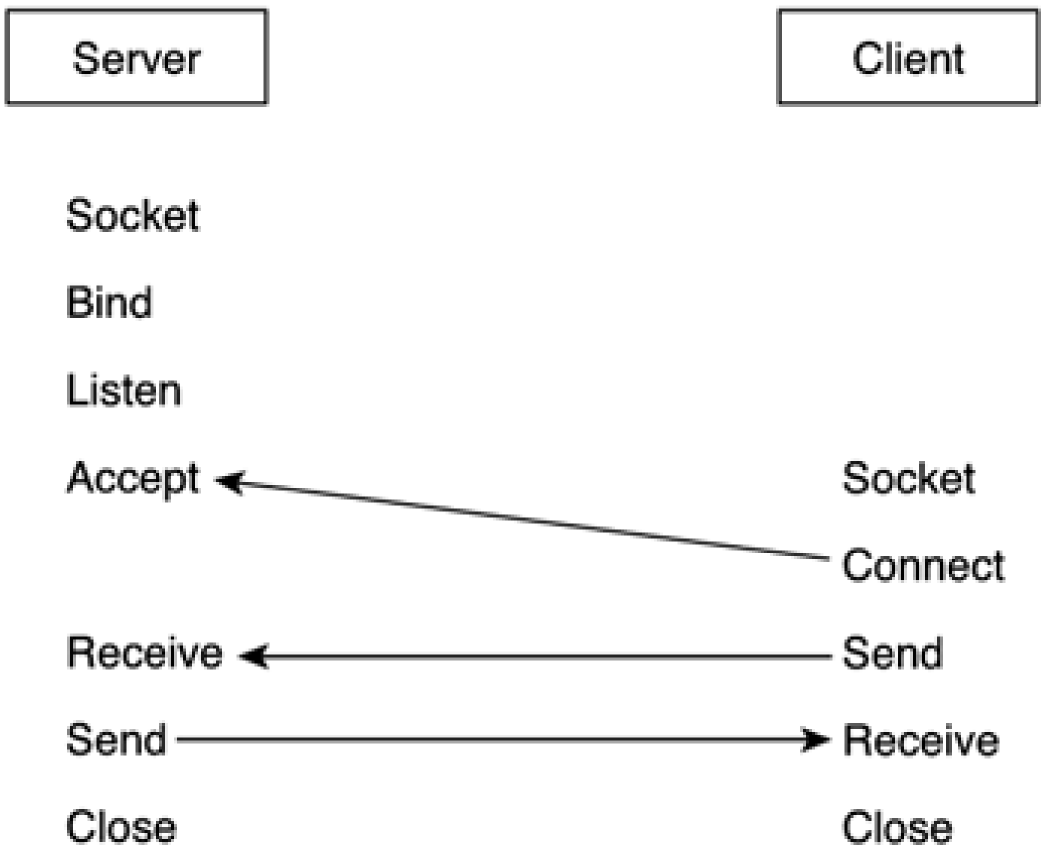
  - puts the server on hold until new connection from client if none in queue
  - returns the client socket descriptor if success, INVALID_SOCKET otherwise

```
int clientSocket = accept(listenSocket, NULL, NULL);

if (clientSocket == INVALID_SOCKET)
   cout << "Accept failed : error #" << WSAGetLastError() << endl;
```

Single-peer TCP server

- Client and server are ready to communicate (send/recv) through the clientSocket



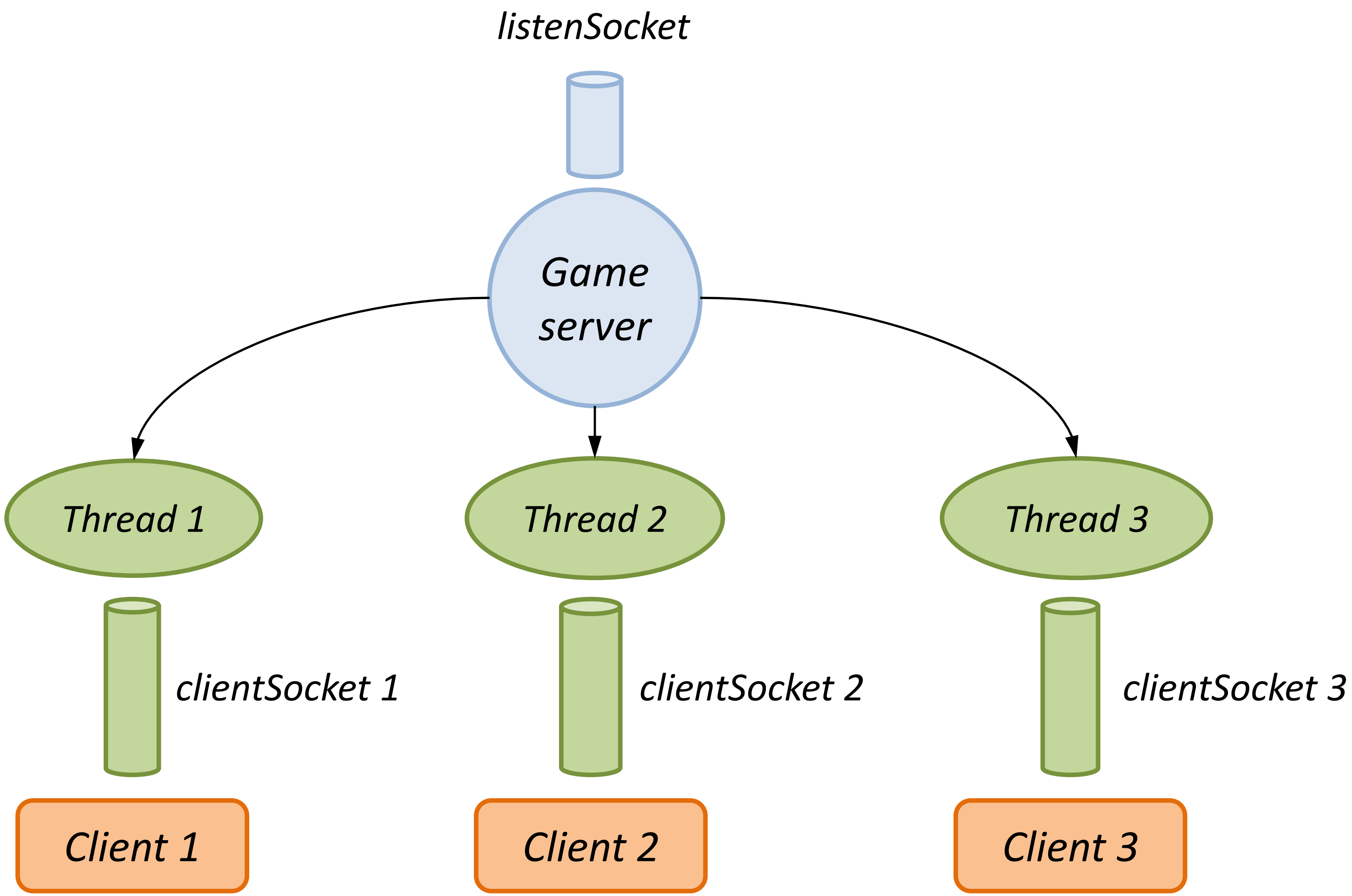
Multi-client TCP server

- In order to handle three to thousands of players in parallel
- After a successful *accept*, we lose the ability to handle more incoming connections
- We need to keep an eye on the incoming connection queue while performing data transfer with already connected clients
  - 2 solutions
    - 1 main thread waiting for new connections (in an accept call) plus 1 thread per socket to handle data transfer
    - iterative approach by checking incoming communications in a loop

Concurrent TCP server

- To spawn a child thread for each accepted connection
- Using Win32 thread API or OpenThreads library or boost::thread class *etc.*
- Algorithm
  1. Main thread creates a listening socket, and binds it to IP/port
  2. Main thread puts it in passive mode with listen function
  3. Main thread waits in a loop for new connections with accept function
  4. Main thread creates a new child thread after each successful accept
  5. Main thread goes to step 3
     1. Child thread enters the send/recv loop
     2. Child thread exits loop when connection terminates

Concurrent TCP server



Concurrent TCP server

```
// connection information
struct addrinfo * result = 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;

// get the IP/port information
getaddrinfo(NULL, port, &hints, &result);

// create a TCP socket
int listenSocket = socket(result->ai_family, result->ai_socktype, result->ai_protocol);

// setup the TCP listening socket
bind(listenSocket, result->ai_addr, result->ai_addrlen);

// put the socket in passive mode, and reserve 2 additional connection slots
listen(listenSocket,2);

// loop 'infinitely' for new connections
while (_serverIsRunning) {
    int clientSocket = accept(listenSocket, NULL, NULL);
    // we have a new connection, spawn a child thread
    // e.g. MyTCPThread childThread (clientSocket);
    }
}
```

Iterative TCP server

- Same behavior without concurrent threads
  - we must be able to check for new connections while communicating with already connected clients
- We can use the select function which allows to check several sockets at once

```
int select (int nfds, fd_set *read, fd_set *write,
            fd_set *except, struct timeval *timeout);
```

  - returns the number of active sockets
  - the 3 fd_set are the sets of sockets checked
    - read: for readability
    - write: for writability
    - except: for errors
  - timeout to avoid waiting forever

Iterative TCP server

```
int clientSocket;
fd_set ready;

listen(listenSocket, 5); // 5 socket slots in the queue

struct timeval to; to.tv_sec = 5; to.tv_usec = 0; // wait max 5 seconds
FD_ZERO(&ready);
FD_SET(listenSocket, &ready); // only own socket available

while (_serverIsRunning) { // server loop
    select(0, &ready, 0, 0, &to); // fills ready with active sockets
    // two cases:
    //   1. data in listening socket means new connection request
    if (FD_ISSET(listenSocket, &ready)) { // new connection required
        clientSocket = accept(listenSocket, NULL, NULL);
        // ... read / write data in client socket
        close(clientSocket);
    }
    //   2. data in another socket, regular data sent from existing client
}
```

Multi-client UDP server

- As information about the client is in the send/receive calls, sorting is already done!
  - but reduced reliability and security
- Example of an echo server

```
void do_echo(int listenSocket) {
 struct sockaddr source_addr;
 int sasize = sizeof(source_addr);
 char buf[SIZEMSG];
 while (_serverIsRunning) {
  int nrecv = recvfrom(listenSocket, buf, SIZEMSG, 0, &source_addr, &sasize);
  int nsent = sendto(listenSocket, buf, nrecv, 0, &source_addr, sasize);
 }
}
```

More information

- On Winsock (information and code)
  - go to msdn.microsoft.com
  - ➢MSDN Library
  - ➢ Windows Development
  - ➢ Networking
  - ➢ Windows Sockets 2

Preventing blocks

- We need extra code to ensure that the sockets respond well to everyday use
- Main issue occurs when we do not have enough data to read
  - how are we supposed to know in advance the size of non-fixed data?
- 3 solutions
  - (to read one byte at each call, but very slow and if no data is available the socket is still blocked)
  - to get information on the size of the data to read
  - to convert the blocking socket to non-blocking

Preventing blocks

- Sneak peek, using the flag in recv / send
  - 0 is the default value, no special behavior
  - MSG_OOB (Out-of-Band) is an urgent flag to retrieve the data as an individual element outside the sequence
  - MSG_PEEK is used to peek at the socket without reading data from it

```
#define BUFFERSIZE 256
char * buffer = new char[BUFFERSIZE];
int available = recv(clientSocket, buffer, BUFFERSIZE, MSG_PEEK);
recv(clientSocket, buffer, available, 0);
```

➢ never blocked by the lack of data

Preventing blocks

- Conversion from blocking to non-blocking socket, using ioctlsocket

```
int ioctlsocket(int clientSocket, long cmd, u_long * argp);
```

  - cmd is a command to perform on the socket
  - argp is a pointer to the parameter for cmd
  - returns 0 if successful, SOCKET_ERROR otherwise
  - conversion

```
// u_long argp = 0; for blocking mode
// u_long argp = 1; ( != 0 ) for non-blocking mode
int result = ioctlsocket(clientSocket, FIONBIO, &argp);
```

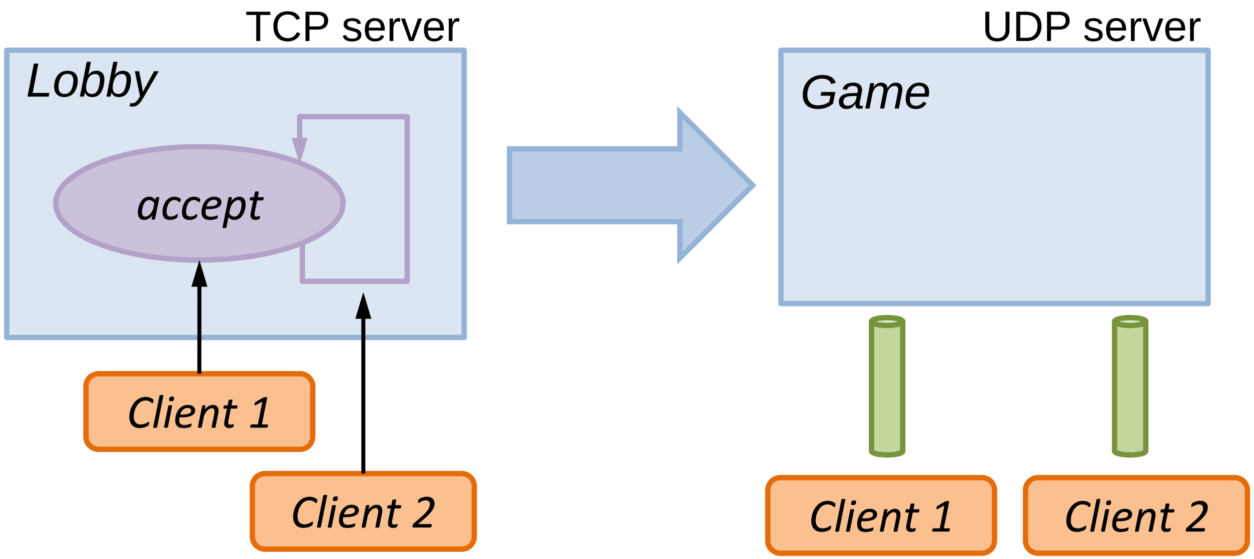
Client-server games

- For small area games (3 to 16 players)
- One player runs both server and client
  - the one with the faster computer and Internet connection
- The other players run clients
- The server initiates the game, and is placed in an accept loop (game lobby)
- When all players have joined the game, the server stops accepting new incoming requests

Client-server games

1. Server: create socket and bind to IP and port
2. Server: open game lobby and show IP/port. Listen and wait in an accept loop
3. Server: while waiting connections, two threads required
   - an interface thread running the game menu interaction
   - a thread running the accept loop
4. Clients: open socket and connect to the game server
5. Server: update screen for each accepted connection and implement the desired connection policy (iterative UDP, concurrent TCP)
6. Server: when all the clients are connected
   1. interrupt the accept loop
   2. close the server listening socket
   3. start the game with the connected client sockets

Client-server games



- Connection management works only at boot time (in game lobby)
  - The game server must ‘reboot’ (*i.e.* be back in accept mode) when a player disconnects to be able to recover the connection

MMO Games

- Many connections, Many data transfers, very restrictive time constraints (lag), in-game connection *etc.*
- Powerful computer or cluster of computers as server
- Players run clients that update the server(s) with player state information
- Servers broadcast the world state back to the players
- Additional problems raise when trying to cover thousands of players, but techniques allow to reduce the amount of information to send

MMO Games

- Data extrapolation
  - When a lag occurs, players' states are not valid anymore
  - We can extrapolate continuous values (such as player position in the world) using the few last known values
  - Jump back to real value when the next network-based value arrives
  - Works well for short lags

MMO Games

- Hierarchical messaging
  - Different gameplay elements receive different priorities
  - Elements to send are determined regarding each client connection bandwidth
  - Example for FPS
    1. enemies and other players positions
    2. shooting and state information
    3. weapon changes
    4. mesh configuration / animation

MMO Games

- Spatial subdivision
  - Games usually take place in a virtual spatial environment
  - Is it not useful to update all players with every other players’ state but only the ones spatially in the neighborhood
  - Games are usually divided in zones, and servers can calculate the N closest players
  - Save a lot of messages to send

MMO Games

- Send state change only
  - instead of sending full player state each time, send only the changes when they occur
  - save bandwidth but more difficult to maintain the synchronization between the players
- Working with server clusters
  - map the spatial disposition to the cluster to avoid data transfer between servers
- Dynamic servers
  - allow to change online the spatial dependency of a server to compensate for a high traffic

End of lecture #13

Next lecture

Scripting