Multiplayer Game Development

305900
Fall 2010
12/06/2010
Kyoung Shin Park

South Park Episode 1008

- http://www.southparkstudios.com/full-episodes/s10e08-make-love-not-warcraft

Multiplayer Game

- Expectation today is that all games come with a multiplayer option- otherwise it's $10 less.
- 2 types of multiplayer gaming:
  - 2 or more players on a single machine/console
  - 2 or more players over separate networked machines
- 2 types of issues to consider:
  - Gameplay – designing a fun multiplayer experience
  - Game development (the CS issues)

Gameplay Issues

- Designing a game to have multiplayer options cannot be an add-on after the game has been developed.
- It has to be considered early in the design phase.
- Because multiplayer can increase the # of polygons you have to be able to draw & how much of the screen you have to split up.
- Vertical vs horizontal split-screen- depends where your enemy's are most likely to appear- in nature horizontal is more important than vertical.
- Co-operative vs Melee
- Co-op:
  - Can co-op be played through the regular storyline of the game? (like Halo)
  - Or only in special multiplayer arenas? (like Quake)
  - What happens if one of you dies? E.g. Halo: Resurrect when no one is shooting your partner.
  - What happens if both of you die? Halo: Restart from the level?

Gameplay Issues

- Can I bring in customized characters? Quake allows mods.
- How do you find other players to play with on a networked game? Match making service.
- How do you find each other during a networked game? Radars.
- Is the game persistent? – e.g. Networked Quake vs Ultima Online / Everquest
- How many players can you support simultaneously?
- Will the game include NPCs during multiplayer sessions? Are NPCs your friend or enemy?

E.g. Gauntlet

- 3rd person view- everyone on the same screen
- Play is co-operative
- What happens when players walk in different directions?
- Camera zooms out up to a limit.

E.g. Quake / Jedi Outcast / Academy

- Entire screen dedicated to 1 player
- Co-op & melee play possible over network.
- Multiple types of network games- free for all; capture the flag
- Challenge: network latency; handling many many players and NPCs; handling mods for customizable game characters.

E.g. Halo

- Split screen co-op/melee and networked play
- Screen resolution limits; polygon counts; network latency; handling many players and NPCs

E.g. Ultima / Everquest / Star Wars Galaxies

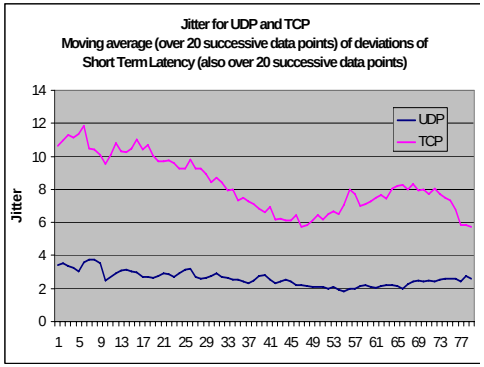
- No single user game mode.
- Entire screen dedicated to 1 player.
- All players are on the network.
- Large scale persistent RPG
- Open-ended play
- Richness and replayability of the game is enhanced by users providing the "content" thru social interactions.
- Must handle 100's/1000's of players over periods of years.
- What happens if a player has to go to bed or go to work? Who will guard their castle?
- Large reliable databases in a constant running simulation environment.
- Database migration problems as # of players grows.
- How to incorporate new game play capabilities over time as players ask for them without disrupting service.

Fundamental Networking Issues

- Protocols (TCP/IP – Transfer Control Protocol / Internet Protocol)
  - TCP – Completely reliable – data arrives as a continuous streams – connection-oriented
  - Because data arrives as a stream there is no guarantee of when each byte of data will arrive. You have to do the work of collecting enough bytes to form a message.
  - UDP – Unreliable – data arrives as discrete packets/messages – each packet is correct – but no guarantee of the order of the packet - non-connection oriented
- Bandwidth: measured in bits per second.
- Latency:
  - This is your biggest enemy!
  - Usually due to: distance; router and end-system overflows; packet loss.
  - Roundtrip times from Chicago: TransUS: 50ms; TransAtlantic: 120ms; TransPacific: 200ms
  - Measured in milliseconds
  - Latency in TCP is higher than UDP because TCP needs to acknowledge that data stream is correct.
  - 200ms roundtrip latency is considered minimum acceptable latency for tightly coupled interactive applications.

Fundamental Networking Issues: Jitter

- Variation of latency.
- Jitter is more offensive than latency because it makes it difficult for the player to predict trajectory of an object.
- Jitter in TCP is much higher than UDP.
- Jitter is usually due to TCP recovering from packet loss.
- Smooth out jitter using traffic shaping or smoothing of avatar movements by averaging position values.



Fundamental Networking Issues

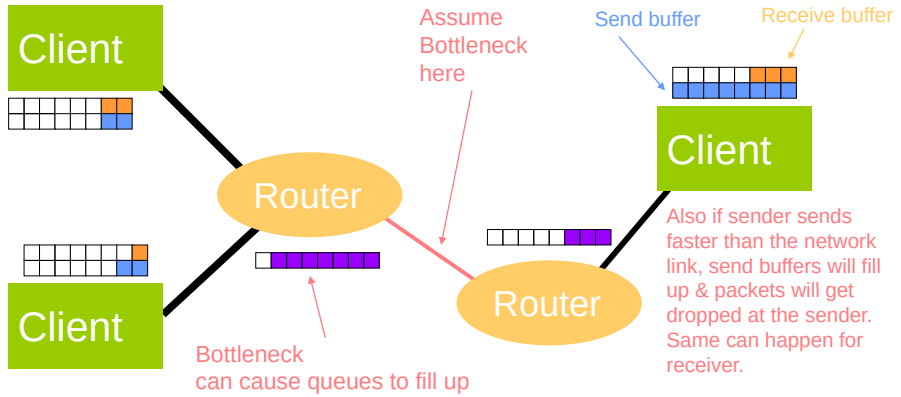
- Unicast, broadcast, multicast
  - Broadcast sends message to multiple receivers one at a time.
  - Multicast sends 1 message and allows router to replicate and send message (based on UDP).
  - Problem is, multicast is not enabled on most routers since it can flood the Internet.
- Firewalls
  - Usually networked applications operate over known port numbers.
  - Firewalls prevent unintentional network traffic from entering computer.
  - But it can prevent network traffic from entering you game application.
  - Game should provide info about port# used so that firewall can be opened.
  - Unfortunately this creates a vulnerability for hackers.

Fundamental Networking Issues

- When buffers are full, packets will get dropped.
- Effect on TCP:
  - Latency will increase becoz TCP will retransmit in order to guarantee reliable delivery.
  - Latency will increase becoz TCP will reduce transmission rate to attempt to reduce congestion
- Effect on UDP:
  - UDP will simply lose data.
- Is there an easy way to tell or determine what the best sending rate is?
- Unfortunately NO- there is no Quality of Service over the Internet – best you can do is to assume the core of the network will never slow down and you have to dynamically adjust your sending rate to accommodate the slowest player- down to a minimum acceptable threshold.
- Rate adjustment is commonly used in audio/video streaming.
- For all clients, as a rule, you should process any incoming packets as fast as possible to avoid congestion at the end points.
- But if your client spends all its time processing packets, it may take time away from making the game go! So a balance must be struck.

Networking Issues that can Ruin a Game Experience

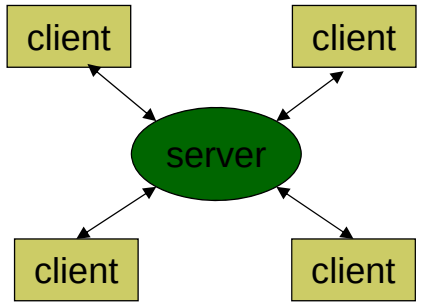
Routers and end-systems have network buffers (queues) which fill up during congestion. Congestion is usually caused by an aggregate of network traffic entering the routers which far exceeds the rate at which the router is able to deliver it..



Connectivity models

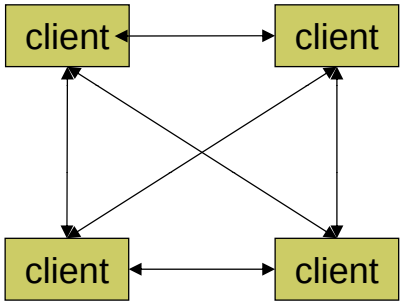
- Shared Centralized – clients connected to central server – simplest to implement & maintain consistency across all clients; but load at server can restrict # of participants that can be supported simultaneously.
- Replicated Homogeneous – clients with no centralized control (multicast) – most scalable solution for large # of clients, but multicast is not deployed across the whole Internet.
- Shared distributed with peer-to-peer updates –clients with full connectivity to each other – often a central mediator initiates the peer connections.
- Shared distributed using client-server subgrouping – subgroups for areas of interest management – most scalable for very large worlds (like online persistent worlds- Ultima Online, etc..)

Connectivity Model: Centralized



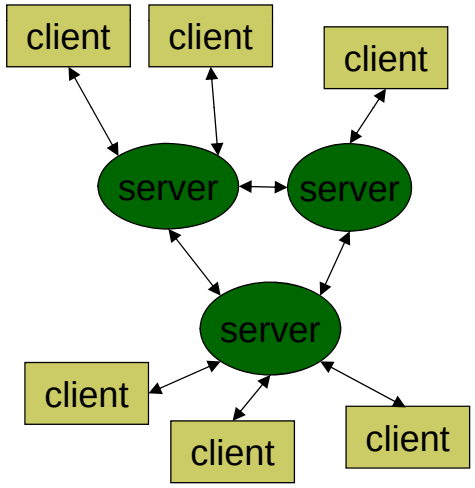
- 클라이언트-서버 모델
- 1 서버 컴퓨터가 최신 데이터를 모두 수집해서 모든 클라이언트들에게 전달함
- 단순한 구조이며, 데이터베이스 유지가 용이함 (useful for compression & admin tasks)
- 확장성이 없고, 중앙 서버가 일이 집중되므로 bottleneck이 생김

Connectivity Model: Distributed



- Peer-to-peer 모델
- 각 클라이언트가 자신의 고유 데이터베이스를 관리
- 최신 갱신정보는 다른 클라이언트들에게 전달
- 모든 연결을 관리하는 것이 어려움
- 확장성이 없고, 네트워크의 양이 많아지므로 네트워크가 bottleneck이 됨

Connectivity Model: Hybrid



- 멀티-플레이어를 위한 멀티-서버를 가진 클라이언트-서버 모델
- 게임에서 많이 사용되는 모델

Extending game loop to handle networking

- Recall multi-threaded game loop:
  - Input Loop Thread
  - Network Receive Loop Thread
  - Compute Loop Thread (put network send calls here)
  - Draw Loop Thread
  - Sound Loop Thread

Implementing a Network Receive Loop

- This pertains to Socket programming (applies to both UNIX and Windows)
- In a thread:

```
While(1) {
    Use blocking select() call to determine which incoming socket has data to
      be read.
    Read data from ready socket.
}
```

- If you do not do a blocking call, your while loop will spin and chew up valuable CPU cycles.
- In DarkBASIC/Blitz put network calls inside the one and only main loop where you first check for available messages, then read them.
- DarkBASIC/Blitz actually implements a separate thread that gathers up network messages before you can poll them in the main loop.
- To learn socket programming take Solworth's Network Systems class or read Richard Stevens book: **UNIX Network Programming**.
- Also look at the Quanta API (www.evl.uic.edu/cavern/quanta)

Network Game Management

- If you multiplayer game needs to retrieve large data files (like 3D models), cache it locally in the same way a web browser caches images.
- Visual representation of networked entities:
  - "avatar"
    - Body position & orientation
    - Head orientation
    - Pre-recorded jestures
    - Body parts list to load @ each client
  - Send absolute state data not key or button presses.
  - Use UDP to send data that is normally sent repeatedly- e.g. a player's position.
  - With UDP try to keep packets no larger than 1K bytes otherwise there is no guarantee of arrival.
  - Pack several variables together- don't send one at a time.
  - UDP does not guarantee the order of packets so include a packet# in the message so that old messages can be filtered out.

Network Game Management

- Estimating bandwidth requirements
  - For every N cycles through the game loop keep track of how many bytes are sent and received and how much time has elapsed. Print total_bytes/elapsed_time to get an estimate of bandwidth requirements. Multiply this by max# of players you want to support to get an upperbound estimate.
  - It is useful in your program to be able to print send and receive rate so that you can observe your game's network consumption during runtime.
- Reducing bandwidth requirements
  - Dead Reckoning. Send position, velocity & acceleration at every N loop cycles rather than every cycle.
  - Send only data that has changed.

How to Implement someone firing a bullet

- When you fire a bullet, broadcast a message like: BULLET position velocity bullet_type
- When clients receive the message they add the bullet to a list of agents/entities they have to manage like any other bullet in the game. Ie. Each client computes how the bullet should fly.
- Each client also figures out if a bullet has hit the client and reports this to everyone else.
- I.e. the client is the one that determines if it is dead. (Clients are very honorable ☺)

Handling Late Joiners

- Players who join the game while it's in session.
- Late joiner joins by first connecting to the main server.
- Server introduces the other clients to the late joiner by giving the other clients the IP address of the late joiner.
- Each client creates an entry in their data structure for the new joiner.
- Each client connects to the late joiner.
- Late joiner creates entries in its data structure for all the clients.
- Late joiner sends data about itself to everyone- in particular the body parts that make up the joiner's avatar.
- All clients send data about themselves to the late joiner.
- This is why it is important to always send state information, not button presses.
- Either allow the late joiner to start playing in the game using incomplete information (e.g. if you don't have a head yet, give you a temporary head).
- Or wait till all the clients and late joiner are fully synchronized. Then allow the late joiner to enter PLAY mode.
- Note: Handling zombies
  - If someone disconnects make sure to remove them from your list of avatars.
  - Since UDP is connectionless it is not possible to tell if someone has "disconnected." So for every peer, make sure to establish a TCP connection too- so that if it breaks, there is a way to know.

References

- http://www.evl.uic.edu/spiff/class/cs426/