Making Networked Games with the XNA Framework

Shawn Hargreaves
XNA Community Game Platform Team
Microsoft

DEVTEACH

Introduction

- XNA Framework 1.0 had no networking support
  - Use other solutions (System.NET) on Windows
  - No network access at all on Xbox
- 2.0 adds a new high level networking API
- Game oriented
- Built on Xbox LIVE and Games for Windows - LIVE
- Up to 31 players per session

Network session types

- To develop and test a networked game
  - Use System Link
  - Only works over a local subnet
  - Xbox requires Creators Club subscription
  - PC does not require any subscriptions
  - Test using Xbox + PC, or two PC's
- To play a networked game
  - Use LIVE PlayerMatch
  - Works over the Internet (including NAT traversal)
  - Xbox and PC both require LIVE Gold and Creators Club subscriptions

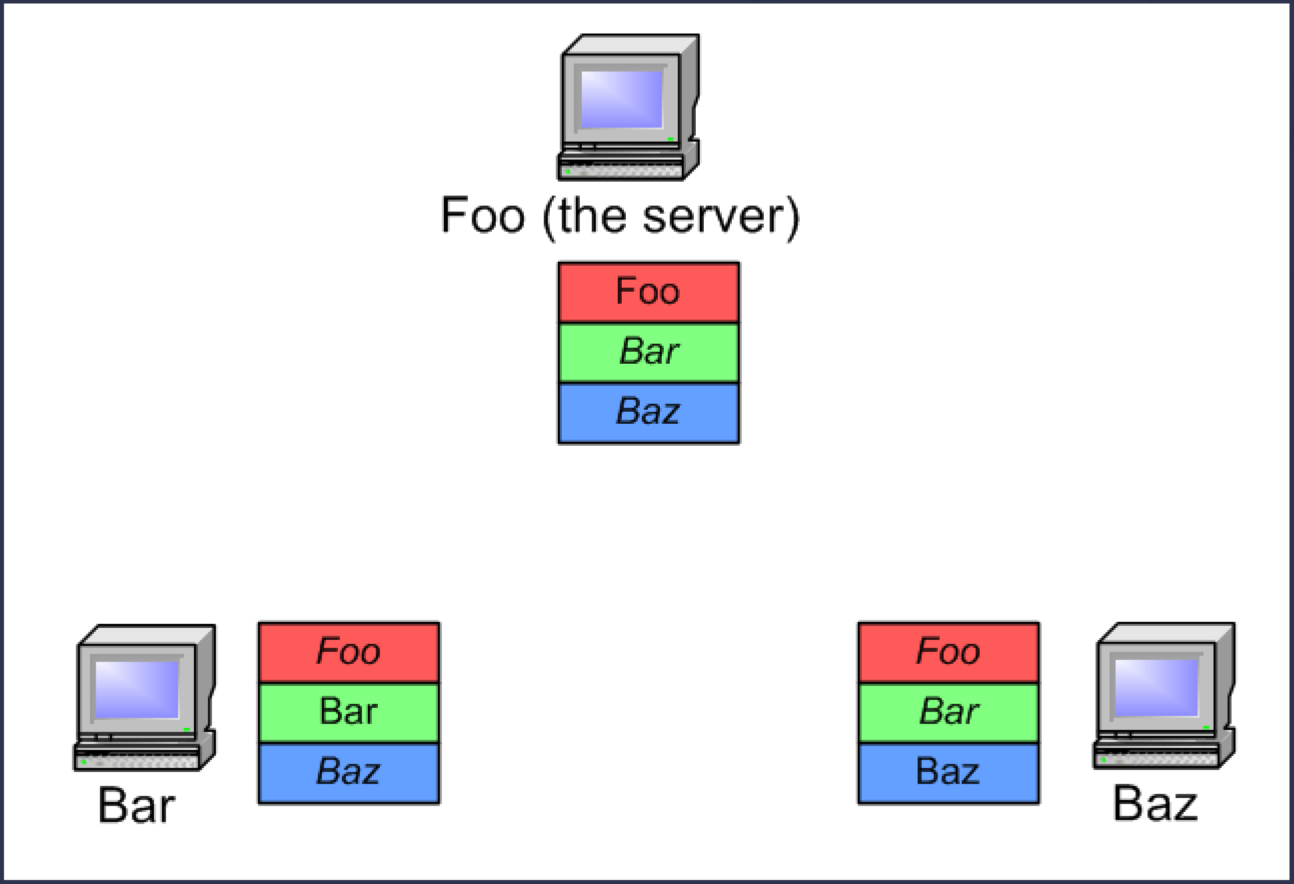
What the framework does for you

- Finding and joining sessions
  - Filtered using title-defined integer properties
- Synchronizing the list of players
  - Gamer joined / left events
- Coordinating lobby <-> gameplay transitions
- Reliable UDP protocol
- Voice “just works”
- Host migration *(partly: see later)*
- Network latency and packet loss simulation

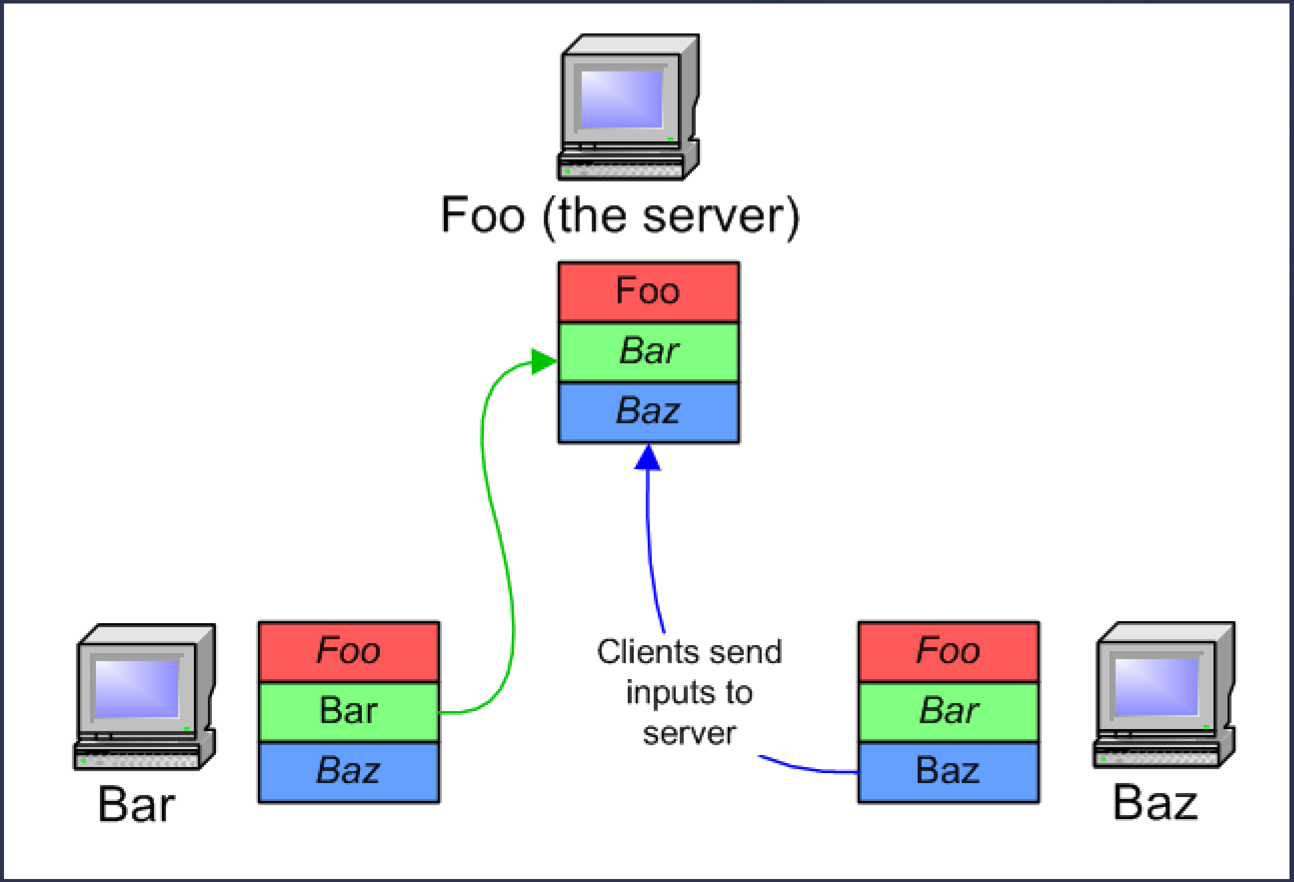
Things you still have to do yourself

- Choose between client/server or peer-to-peer
  - The framework doesn't care which you pick
- Send game data over the network
  - Compressed!
- Deal with network latency
  - Prediction
  - Interpolation
- Make host migration actually work
  - It is turned off by default

Client / server architecture

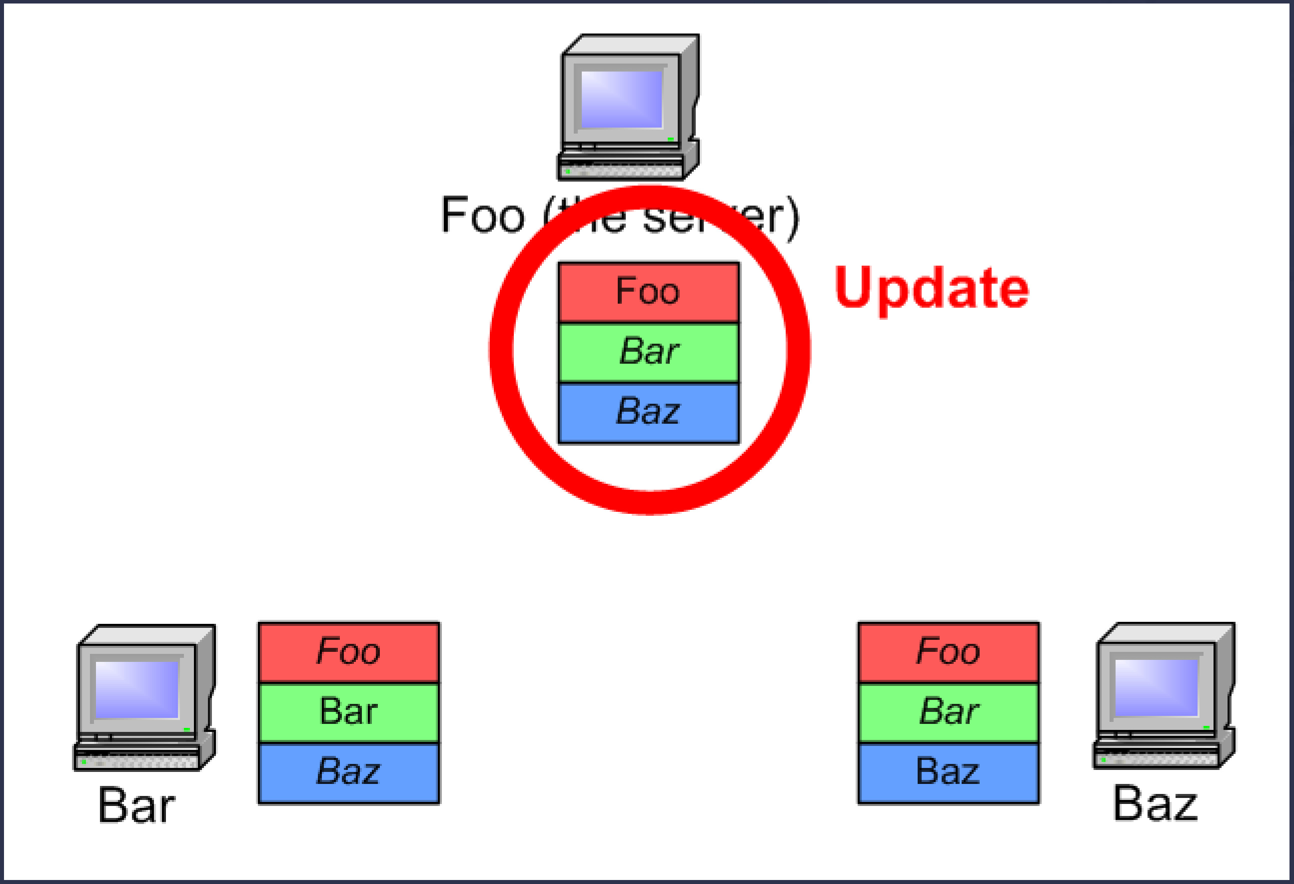


Client / server architecture

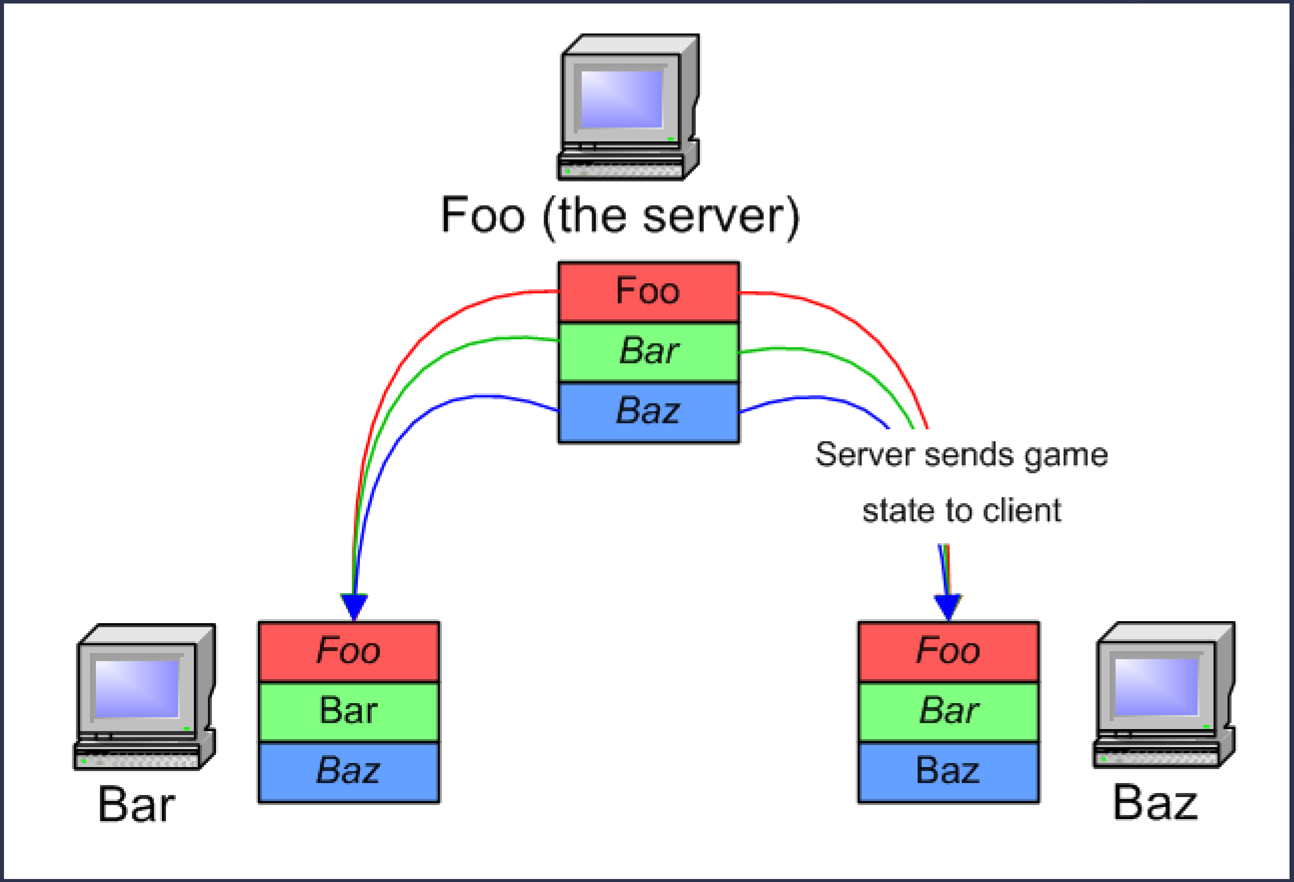


DEVTEACH

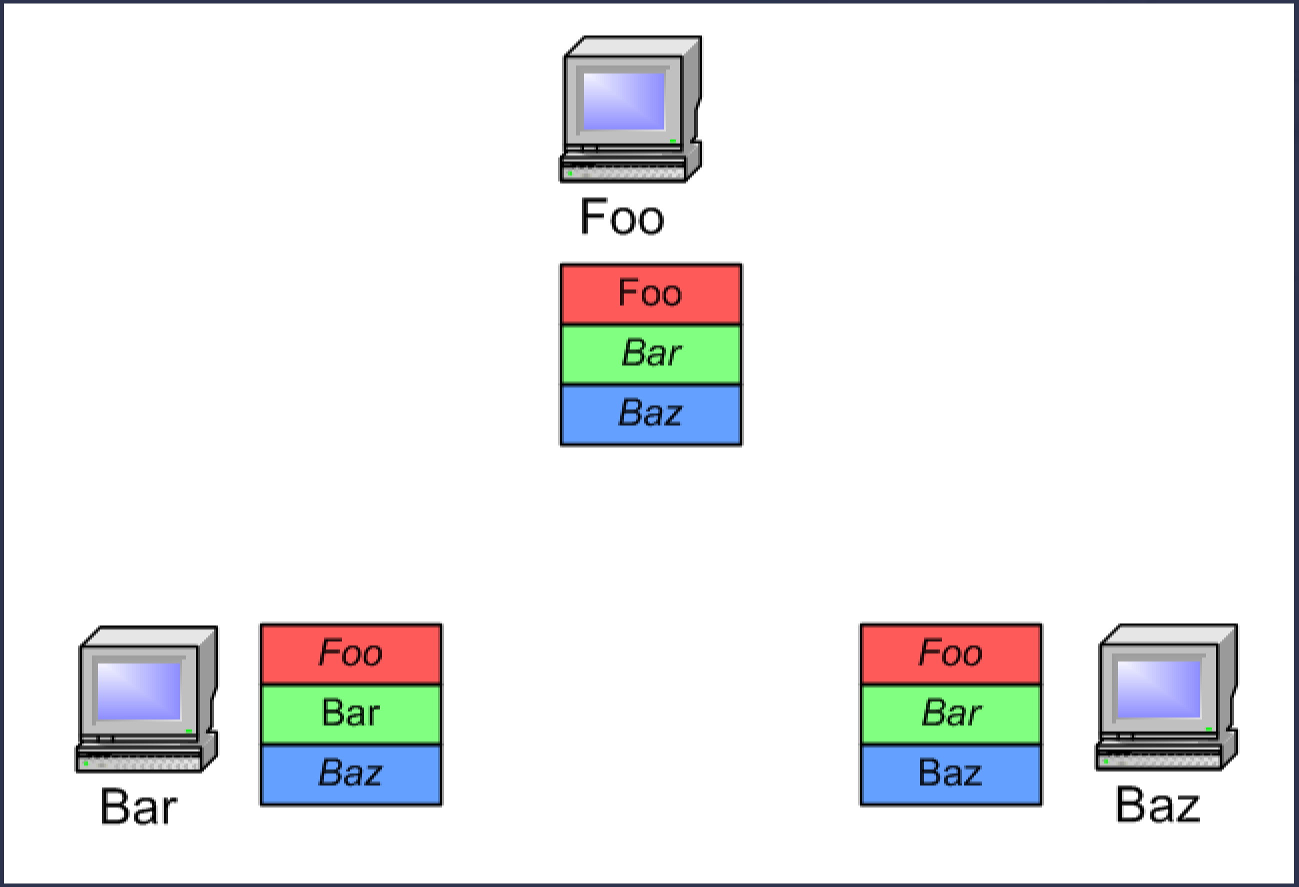
Client / server architecture



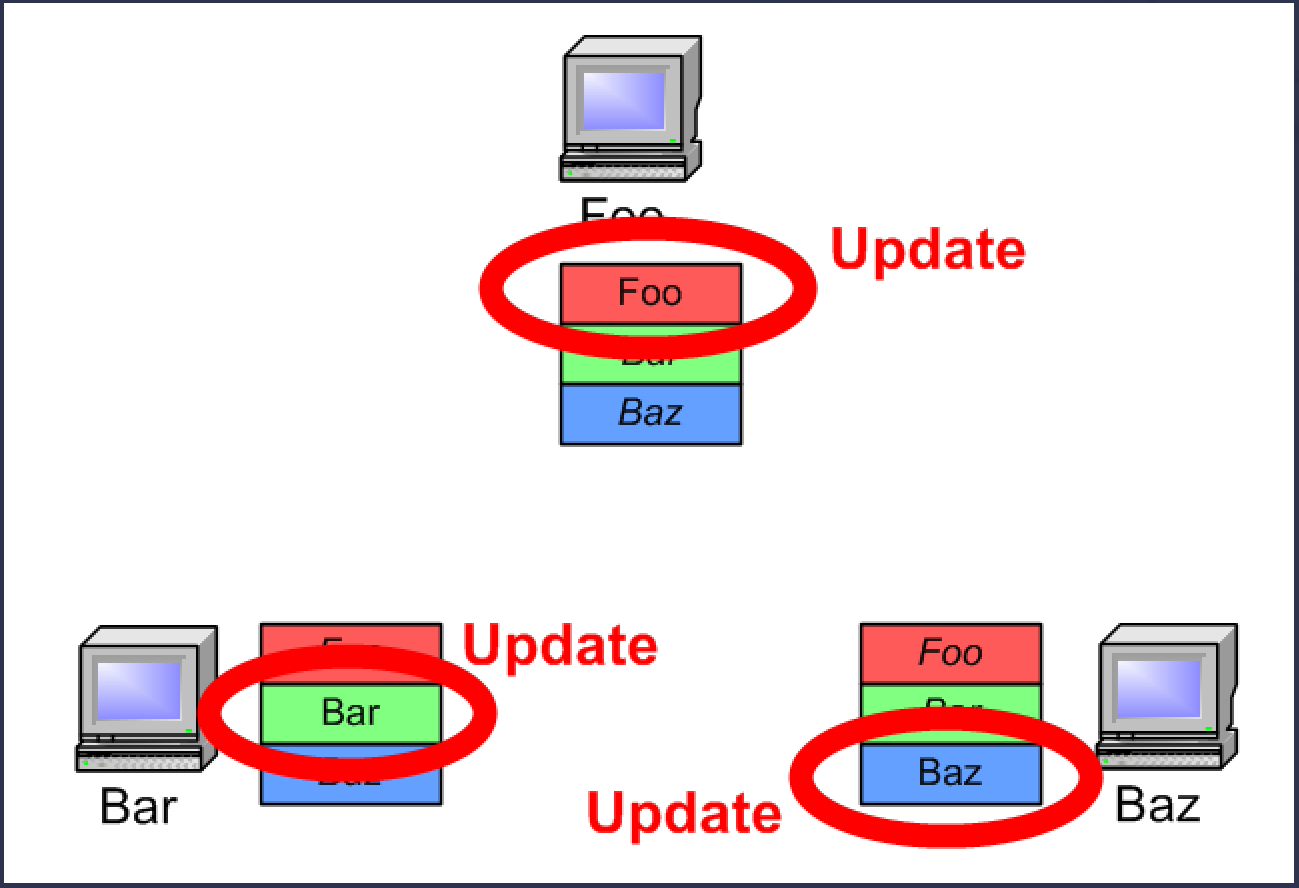
Client / server architecture



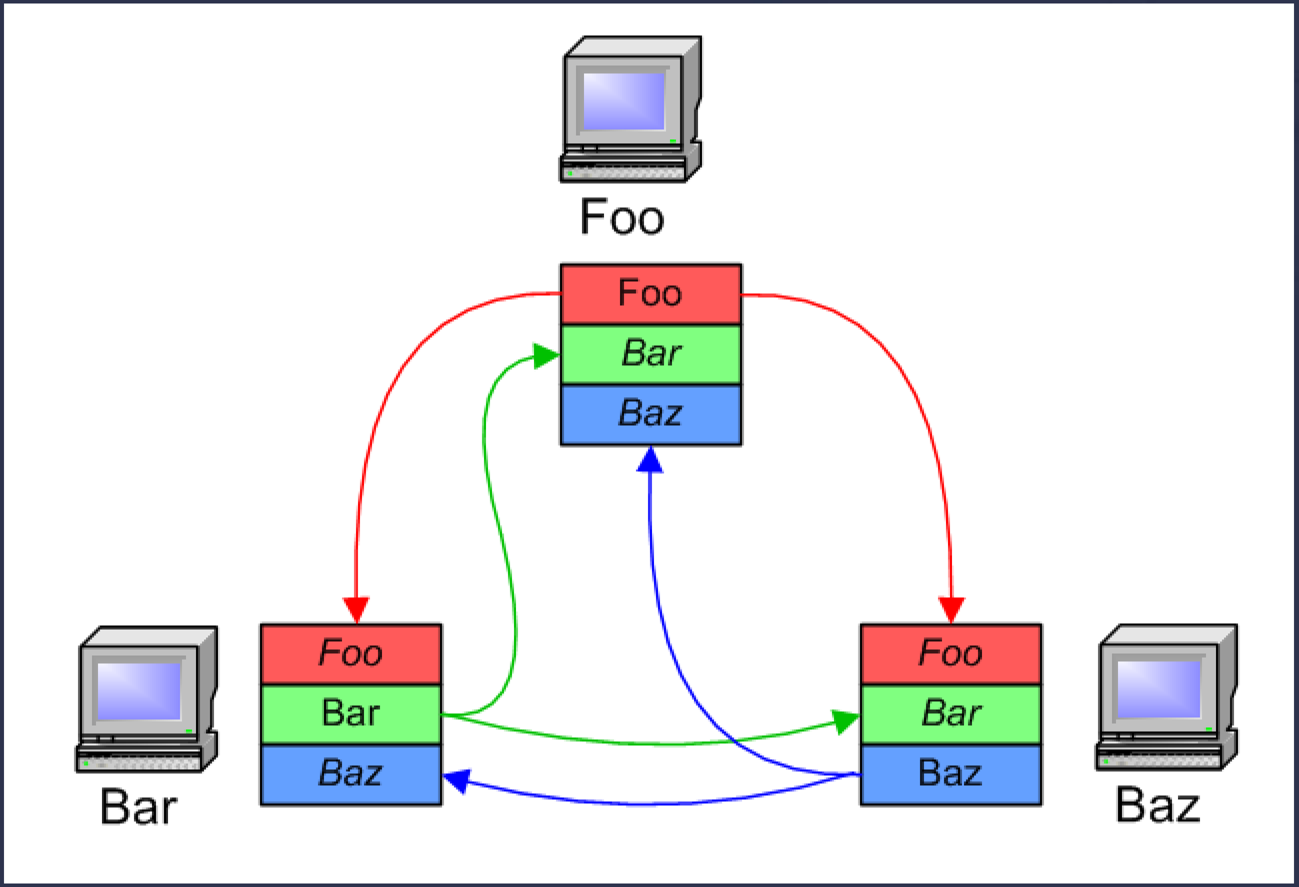
Peer-to-peer architecture



Peer-to-peer architecture



Peer-to-peer architecture



Pros and cons

- Client / server
  - Less likely to suffer consistency problems
  - Harder to cheat
  - “Host advantage”
- Peer-to-peer
  - Uses less network bandwidth
  - Workload is distributed more evenly across machines
  - No lag for local player movement
  - Easier to support host migration

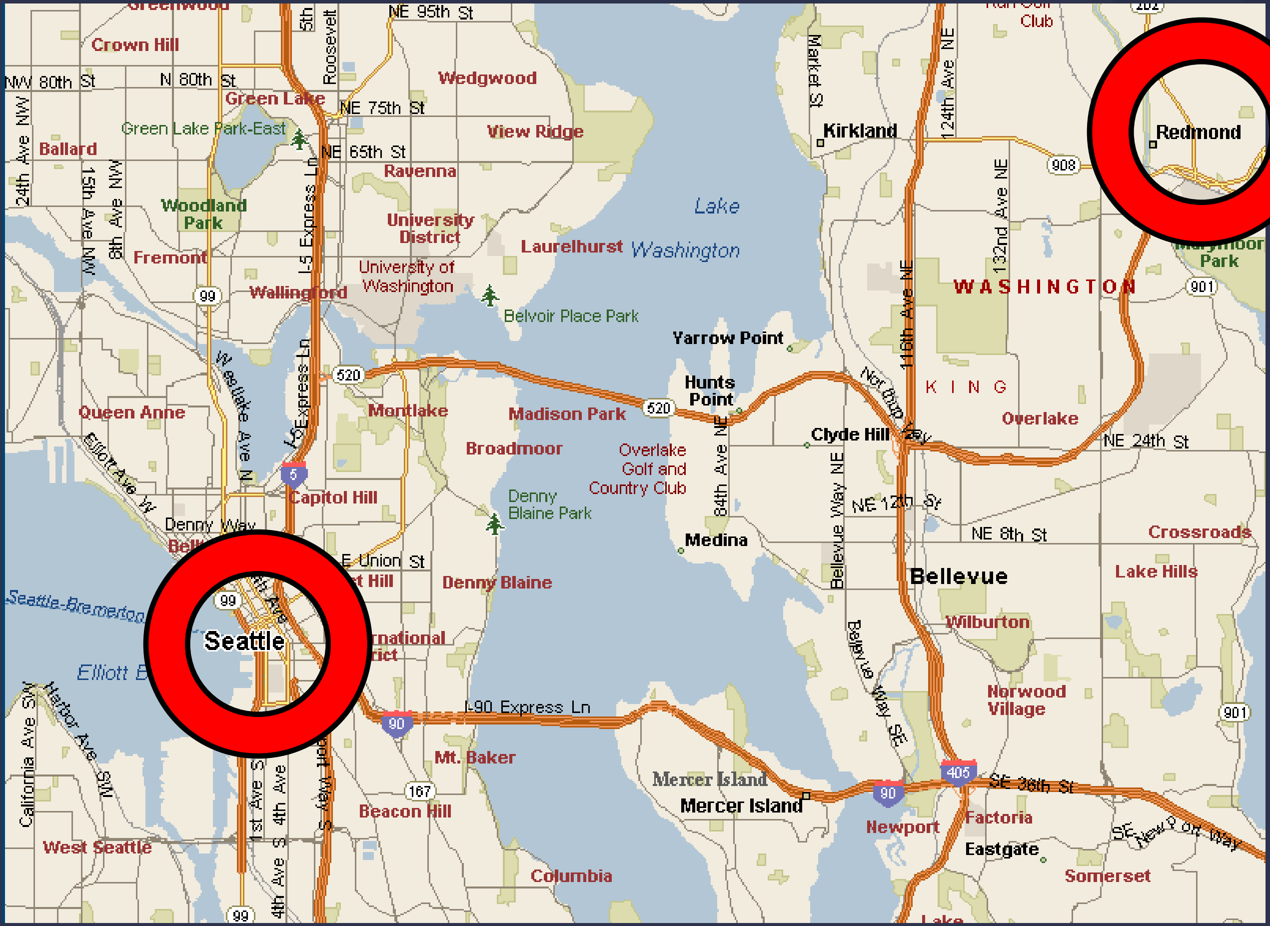
Hybrid network topologies

- Some things matter a lot
  - Am I dead?
  - Who picked up the Pan Galactic Gargle Blaster?
  - Who won?
- Some things only matter a little bit
  - Where am I?
  - What direction am I moving?
- Some things don’t matter at all
  - Is the tree branch swaying gently to the left or the right?
  - Which way did the 623rd dust particle bounce?

Network game programming is hard!

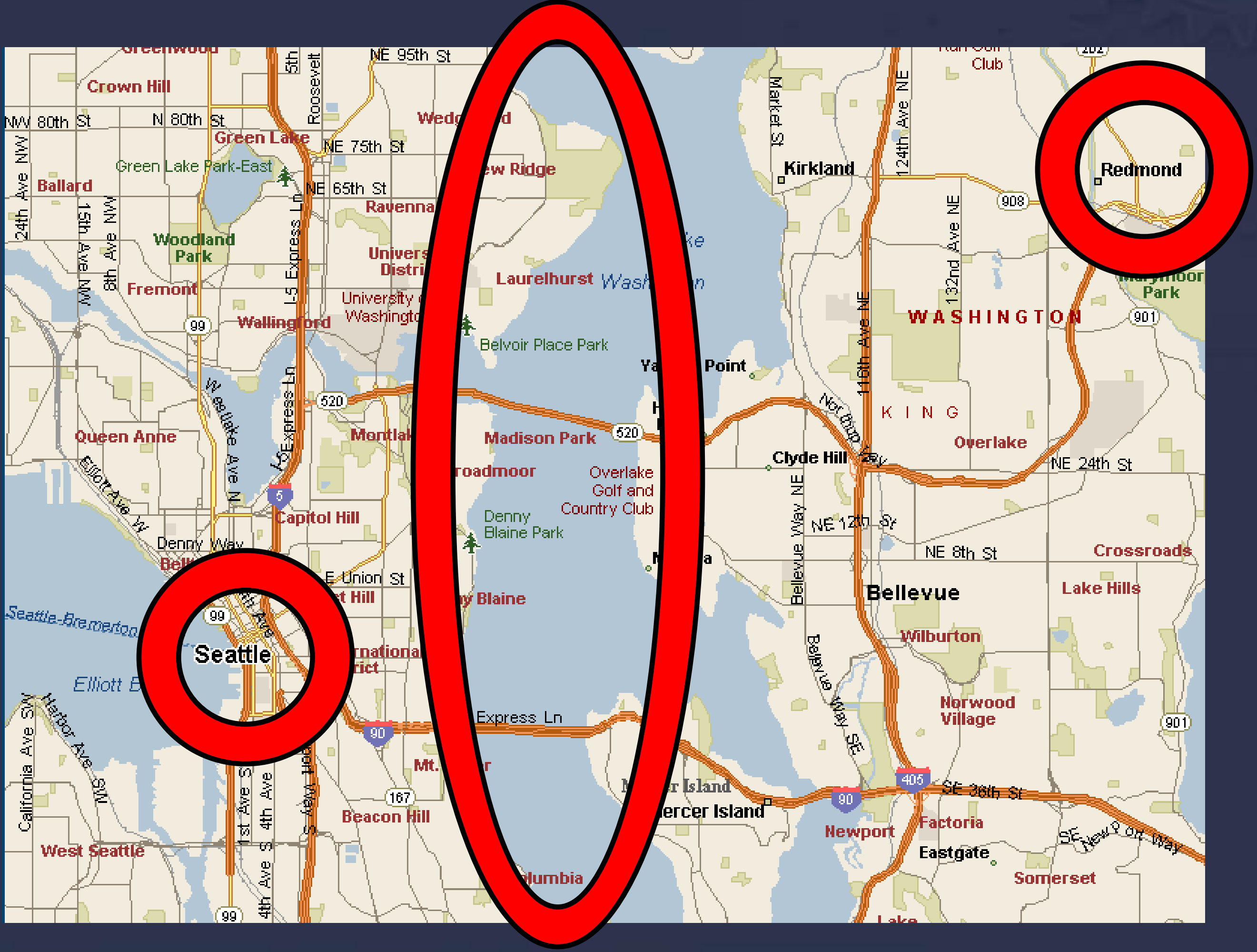
- Three unfortunate facts of life
  - Bandwidth
  - Latency
  - Packet loss

Bandwidth



DEVTEACH

Bandwidth


Crown Hill
Green Lake
Green Lake Park-East
Ballard
Woodland Park
Fremont
Wallingford
Ravenna
Laurelhurst
Belvoir Place Park
Madison Park
Queen Anne
Capitol Hill
Denny Blaine Park
Overlake Golf and Country Club
Seattle
Kirkland
Redmond
WASHINGTON
KING
Overlake
Clyde Hill
Bellevue
Crossroads
Lake Hills
Wilburton
Norwood Village
Newport
Factoria
Eastgate
Somerset
Beacon Hill
West Seattle
NE 95th St
NE 75th St
NE 65th St
NE 8th St
NE 24th St
SE 36th St
Express Ln

Bandwidth

Bandwidth

- How much is available?
  - Assume 64 kilobits (8 kilobytes) per second
  - Some players will have more
  - Often more downstream than upstream
- How much am I using?
  - NetworkSession.BytesPerSecondSent
  - NetworkSession.BytesPerSecondReceived

Packet header bandwidth

- Packet headers are bulky
  - 20 bytes for the IP header
  - 8 bytes for the UDP header
  - ~22 bytes for the XNA Framework
  - ~50 bytes total
- If you send a single bool to one other player, 60 times per second, this requires
  - 60 x 1 byte of payload data = 60 bytes
  - 60 x 50 bytes of packet header = 3000 bytes
  - Bandwidth usage: 3 kilobytes per second
  - 98% overhead

Surviving the packet headers

- Send data less often
  - Typically 10 to 20 times per second
  - Prefer a few big packets to many small ones
  - Framework automatically merges packets if you send multiple times before calling NetworkSession.Update
  - This is why games prefer UDP over TCP
- Example
  - 8 players (each sending to 7 others)
  - Transmit 10 times per second
  - 64 bytes of game data per packet
  - Bandwidth usage: (64 + 50) * 7 * 10 = 7.8 kilobytes per second
  - 44% overhead

Voice bandwidth

- Voice data is ~500 bytes per second
- By default, all players can talk to all others
- In a 16 player game, talking to all 15 other players
  - 500 * 15 = 7.3 kilobytes per second
  - Yikes ☺
- LocalNetworkGamer.EnableSendVoice
  - Only talk to players on your team
  - Only talk to people near you in the world
  - But avoid changing this too often!

Compression

- Generalized compression algorithms are not much use
  - Packets are typically too small to provide a meaningful data window
- Prioritize data
  - Send less important things less often
  - Update further away objects less often
  - Don’t bother synchronizing objects that are behind you
- Send deltas instead of complete state
  - But not if this means having to make everything reliable!
- Send smaller data types
  - int -> byte
  - Matrix -> Quaternion + Vector3
  - Avoid strings

Compression: quantization

```
float rotation; // in radians

packetWriter.Write(rotation);
```

```
rotation *= 256;
Rotation /= MathHelper.TwoPi;

packetWriter.Write((byte)rotation);
```

Compression: bitfields

```
bool isAlive, isRespawning, isFiring, hasPowerup;

packetWriter.Write(isAlive);
packetWriter.Write(isRespawning);
packetWriter.Write(isFiring);
packetWriter.Write(hasPowerup);



byte bitfield = 0;

if (isAlive)      bitfield |= 1;
if (isRespawning) bitfield |= 2;
if (isFiring)     bitfield |= 4;
if (hasPowerup)   bitfield |= 8;

packetWriter.Write(bitfield);
```

Compression: 16 bit floats

```
float angle;
float speed;

packetWriter.Write(angle);
packetWriter.Write(speed);
```

```
HalfSingle packedAngle = new HalfSingle(angle);
HalfSingle packedSpeed = new HalfSingle(speed);

packetWriter.Write(packedAngle.PackedValue);
packetWriter.Write(packedSpeed.PackedValue);
```

Compression: random number seeds

```
foreach (Star star in starField)
{
   packetWriter.Write(star.Position);
}
```

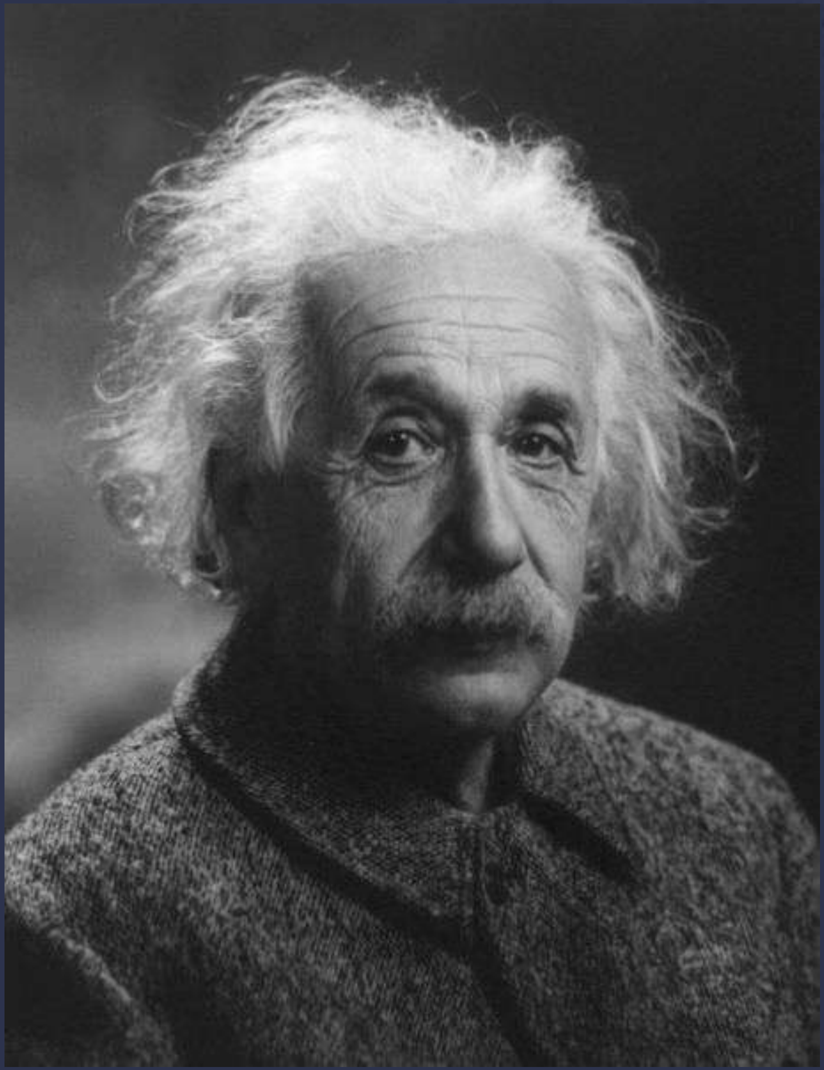
```
int seed = (int)Stopwatch.GetTimestamp();

packetWriter.Write(seed);

Random random = new Random(seed);

foreach (Star star in starField)
{
   star.Position = new Vector2((float)random.NextDouble(),
                               (float)random.NextDouble());
}
```

Latency



Latency

- Speed of light = 186282 miles per second
- Nothing can travel faster than this
- Some distances
  - Seattle to Vancouver: 141 miles = 0.8 milliseconds
  - Seattle to New York: 2413 miles = 13 milliseconds
  - Seattle to England: 4799 miles = 26 milliseconds

Latency

- It's actually worse than that
- Network data does not travel through a vacuum
  - Speed of light in fiber or copper slows to 60%
- Each modem and router along the way adds latency
  - DSL or cable modem: 10 milliseconds
  - Router: 5 milliseconds on a good day, 50 milliseconds if congested

Latency

- So how bad can it get?
  - Xbox games are expected to work with latencies up to 200 milliseconds
- How can I try this at home?
  - NetworkSession.SimulatedLatency

Dealing with latency

- Machine A is controlling object A
- Machine A sends a packet to B, containing
  - The position of A
  - The velocity of A
- Machine B reads the packet
  - Uses NetworkGamer.RoundTripTime to guess how old the data is
  - Estimates the current position of the object
    - currentPosition = packetPosition + velocity * estimatedLatency
- Needs lots of damping and smoothing to look good

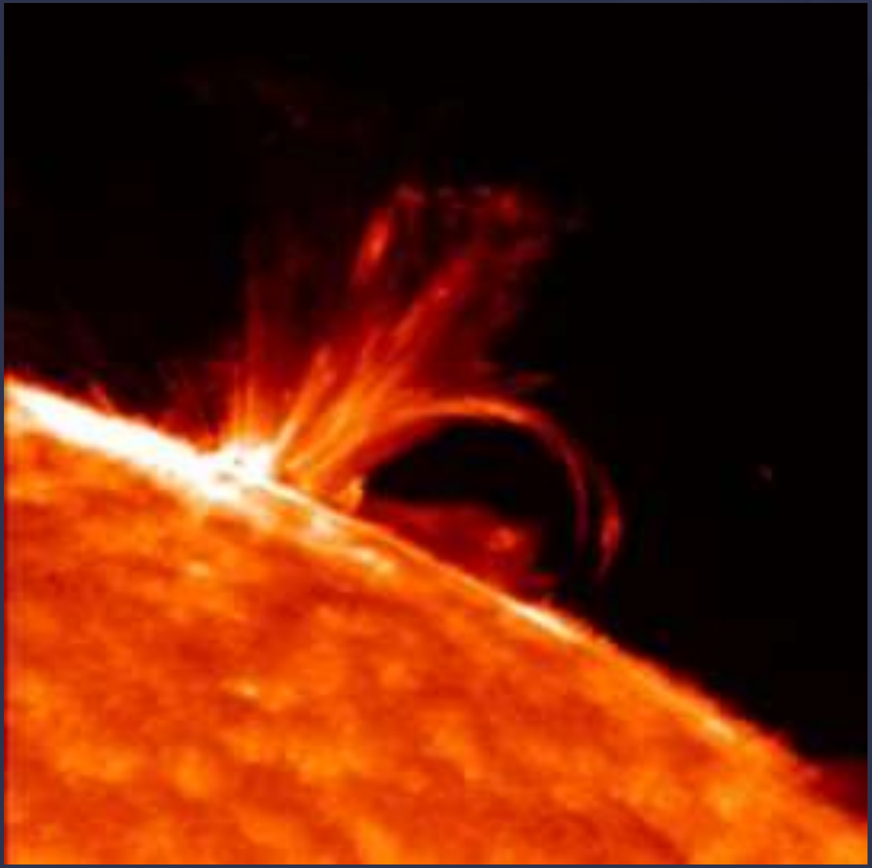
Dealing with latency: improved

- Use the game simulation to predict object movement
- Machine A sends a packet to B, containing
  - The position of A
  - The velocity of A
  - Current user inputs controlling A
  - Any other simulation or AI state which could affect the behavior of A
- Machine B reads the packet
  - Resets local copy of A to the state described in the network packet
  - Runs local update logic on A to “catch up” to the current time

```
for (int i = 0; i < estimatedLatencyInFrames; i++)
    a.Update();
```

  - Smooths out the result as before

Packet Loss



Packet loss

- Traditionally, games had to worry about
  - Packets never being delivered
  - Packets being delivered in the wrong order
  - Corrupted packet data
  - Packets being tampered with by cheaters
  - Accidentally reading packets from some other program
  - Packet data being examined in transit
- The XNA Framework helps with all of these

Packet loss

- Traditionally, games had to worry about
  - Packets never being delivered - reliable UDP (optional)
  - Packets being delivered in the wrong order - in-order delivery (optional)
  - Corrupted packet data - secure packets
  - Packets being tampered with by cheaters - secure packets
  - Accidentally reading packets from some other program - secure packets
  - Packet data being examined in transit - secure packets
- The XNA Framework helps with all of these

Packet loss

- To avoid packets being delivered in the wrong order
  - SendDataOptions.InOrder
  - This is very cheap
  - Once a later packet has been received, earlier ones are simply discarded
- To make sure packets are delivered at all
  - SendDataOptions.Reliable or SendDataOptions.ReliableInOrder
  - More expensive
  - Can cause additional latency
- Recommendation
  - Use SendDataOptions.InOrder for most game data

Packet loss

- How bad can it get?
  - Xbox games are expected to work with packet loss up to 10%
- How can I try this at home?
  - NetworkSession.SimulatedPacketLoss

THE END

QUESTIONS?

DEVTEACH