Who is this guy?

- Ben Deane
- Programmer at Blizzard on the Battle.net team
- Lifelong* network game programmer

At least for my working life. . .



What has he done?



I’ve done RTS games. Populous 3 was first game I did that was properly architected for Internet play.

What has he done?



I've done FPS games. Firewarrior was Europe's first PS2 Internet-playable game. It used a peer-to-peer network topology. Goldeneye: RA was a poorly received game, but the network game had some fun features and some interesting development stories I can tell you about.



What has he done?



I've done other types of games, and games where I worked on more network infrastructure issues than actual gameplay networking. For Theme Park World, I wrote the online park publishing & visiting code, basically a glorified chatroom. For D3 my team did the whole Battle.net service.

## What's in this lecture?

- Real world examples
- Practical advice
- Some war stories
- Spartan slides

"Experience is simply the name we give our mistakes."

*– Oscar Wilde*

My aim for this lecture is to expand on what you've already learned in the academic context, show you how it's applied in the industry, make you aware of messy real-world issues, and tell you some stories of where things have gone right for me, and where things have gone wrong.

Sorry about the spartan slides. They're sort of deliberate - most of the important stuff is in the talk. Although I regret not having more pictures.

I'm concentrating mostly on expanding and giving context to Magda's lectures to do with the major problem of networking game programming: how to hide latency.



## Why Network Programming?

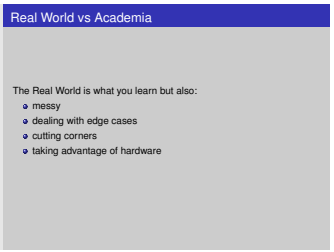
| Year | CPU (MHz) | Memory (MB) | Typical RTT (ms) |
|---|---|---|---|
| 1995 | 90 | 8 | 300 |
| 2000 | 400 | 32 | 300 |
| 2005 | 1400 | 256 | 300 |
| 2010 | 2660 | 4096 | 300 |
| 2014 | 3330 | 16384 | 300 |

- Networking programming stays interesting and challenging
- Hiding latency is the constant problem to solve
- Non-network programmers just discovered concurrency?

Network programming doesn't really get easier as machines get faster, unlike most other programming disciplines.

Network programming is messy in ways that other programming disciplines aren't. Errors are normal.

You think you have threading issues? That's nice. Welcome to my world. You're not even concurrent across different machines - try spending 3 weeks poring over hundreds of MB of logs for a problem that occurs 0.1% of the time in a 16-machine network game, then tell me how hard your threading bug is.



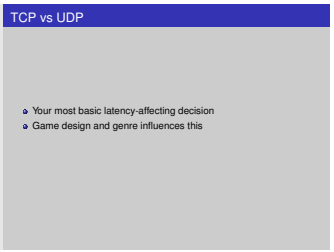

Everything you learn in school is applicable to the real world. But errors and edge cases happen all the time. You can't hope to cover all the weird and wonderful setups that players have, in your dev environment.

On the plus side, you can cut a lot of corners. The objective is to make a game fun, not to model the real world accurately or have any kind of internal consistency.

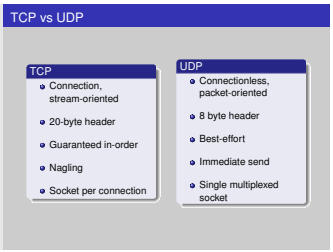
Good example (although not network-related) of where the real-world differs from academia is from UE3 frustum culling code. Conventional algorithms class would cover spatial subdivision of the world (eg quadtree) and teach how to test the view frustum against that to get good performance. All fine.

But in the real world, your game might only have a couple of hundred entities to test (eg Gears of War). Your fancy algorithm isn't going to beat a simple linear test of all the objects, especially if you can take advantage of cache prefetching and/or offload it to a fast piece of hardware (PS3 SPU).



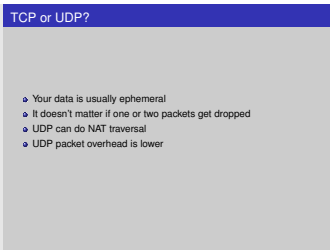



(none)



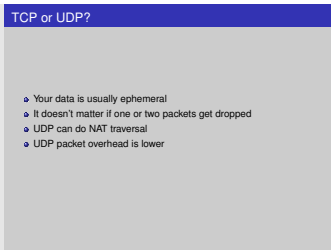

(none)






You all know the difference between UDP and TCP.
It's "received wisdom" in action games to use one's own partially-guaranteed protocol over UDP, a few of reasons for this:
1. Your data is usually ephemeral and what's valid to send this frame will be invalidated next frame: you don't want to block waiting to send data.
2. For the same reason, it doesn't matter if one or two packets get dropped as long as the game state converges.
3. UDP has a big advantage in network topology which is that it is possible to do NAT traversal using STUN or some variant thereof.
4. UDP packet overhead is lower.

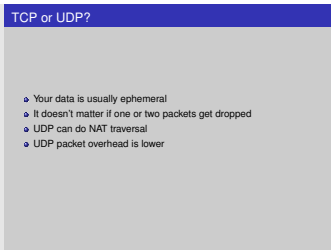
However, look again at point 1 and consider real life network behaviours. How common is sustained or sporadic loss?




I remember a gd-algorithms group thread from some years ago in which it was argued that packet loss is not normally nicely sustained at a low rate, but is bursty. That is to say that a network dropout of a second or two would probably stall your game just as badly on UDP as on TCP.
(However, UDP recovery is better because you don't waste time resending the packets that are out of date.)
This was/is probably true in the US and other countries with mature internet infrastructure.

However, our experience of Chinese networks has shown that it is not uncommon to have sustained high packet loss (~20%). Even with relatively high speed, high bandwidth connections.






The fundamental issue here is that TCP solves a different problem. TCP solves the problem of efficient link utilization, not your problem of timely packet delivery. It may have occurred to you that some of your data needs to be reliably delivered and some needs to be timely but unreliable. Using TCP and UDP together is also problematic - TCP tends to affect the timely working of UDP.

However, TCP can be used even for action games! SOCOM (an early PS2 FPS) used TCP.

Synchronizing Time I

Method 1. An NTP-like algorithm
- Estimate RTT with smoothing
- Adjust clock by (time on wire)/2
- Part of connection establishment
- Sync to epoch (eg. start of level)

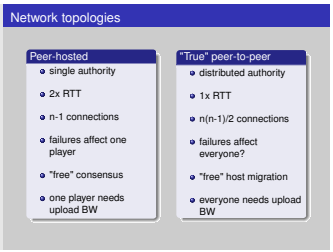
Games often usually do pretty much what you'd expect.
Send a packet, record RTT, subtract time at the remote end, divide by two. This gives you a rough estimate of your one-way trip time.
Do that a few times to try to get a reasonable average, discard outliers, etc. Very simple statistical smoothing.
Sometimes it's enough to sync once, other times it's at the beginning of a level. Generally it depends on the game and when the connection is made.



Synchronizing Time II

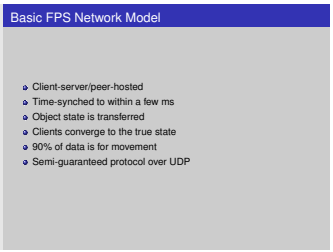
Method 2. Iterative approach
- Client guesses time on server
- Server tells client how wrong it is
- Client adjusts its clock and repeats
- Stop when you're within tolerance

This method converges pretty quickly. This is a quantitatively different approach from method 1.
Method 1 tries to measure accurately, then calculate. Method 2 tries to guess and improve the guess. To my mind, method 2 is the simpler method. I find that iterative algorithms are often overlooked as a solution.




Peer-to-peer code is more complex. Client-server gives a nice model of authority. (The server can cheat vs anyone can cheat). Or in a more relaxed view of things, the server has an advantage.
Peer-to-peer gives you half the latency because there is no round trip; each packet only travels across one link.
Peer-to-peer is more brittle. If your game can't tolerate connection drops very well, you'd be advised to minimise the number of connections made.
Peer-to-peer is harder to establish the mesh especially in the presence of NAT.
(Firewarrior NAT negotiation story)
Peer-to-peer makes some things easier (eg. logic for host migration). But other things are harder: determining consensus among the players.
True peer-to-peer requires that everyone have enough upload bandwidth to send to every other player. This might be an issue, especially since most ISPs offer asymmetric plans.
Peer-to-peer doesn't scale.






Object state is transferred vs inputs being transferred. This is not a parallel simulation. There are typically only a few dozen networked objects alive at any one time.
The game state does not really exist in its true form on any one machine, rather, all machine are continuously converging to the correct state.

Typical FPS Choices

- Two bullet types
- High fidelity human animation (=> head shots)
- Relatively few active objects at a time
- High render rate, low logic rate
- Available headless server
- Simple/Nonexistent AI

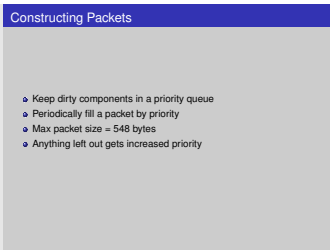
Lightspeed bullets vs projectiles. Lightspeed bullets are interesting for prediction models.
Bullet hits typically require interaction with the animation system.
FPSes typically run at high frame rate but they do relatively little logic.
The logic (eg pathfinding) can run at a low Hz. With a decent network engine, the frequency of packet send can be dialled down also (eg 10Hz or even lower).



Example Semi-Guaranteed Protocol

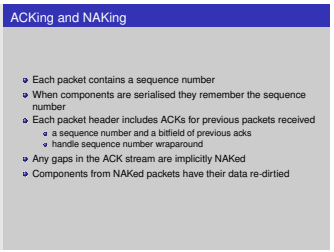
- Entity-component model
  - Movement/Position/Rotation
  - Animation state
  - Health/Armour/Death state
- Components are marked dirty as their state is updated
- Components map to network "channels"
- Network channels are given priorities

When a component is dirtied, it gets assigned a send priority based on its network channel priority.

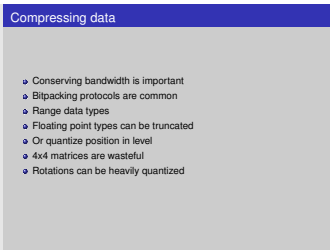



Dirty components are kept in a priority queue to send.
(Can anyone tell me why the max packet size should be 548 bytes?)
576 bytes is the minimum IPv4 datagram size that all hosts must accept (RFC 791).
Actual data size is 576 - 20 - 8 = 548. (ie. minus size of IP and UDP headers).
Amount of priority increase and priority of the channel are policy values that make sense for the game. eg. Health is high priority.



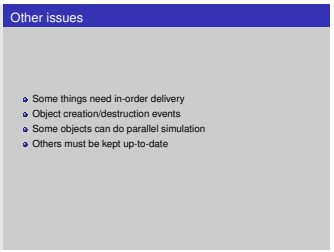



Most components are continually being re-dirtied anyway.
You can also use ACK tracking to continually monitor RTTs and notice when things are getting bad so that you can back off sending frequency.

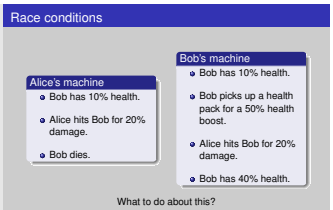



It is usually important to conserve bandwidth as much as possible. This was true 15 years ago and it's true now. If bandwidth creeps up to near link capacity, it starts to make latency worse real fast. Many people these days use their network connections for other purposes during gameplay - sometimes on different machines. eg. VoIP clients, or someone else in the household watching Netflix.
Generalized compression is sometimes used, although less often than you'd think.
Range-bounded integers can use no more bits than you need.
Position can be converted to fixed-size grid coordinate within a level. Take care over the origin offset though - it's common for levels to be built nowhere near (0,0).
Height coord in particular is often suitable for quantization. We mostly live on a 2D plane, and engines can automatically move players to a sensible ground height.
Matrices can become quaternions. (16 numbers -> 4 numbers).
It is hard to notice artifacts in rotation even using just a byte.



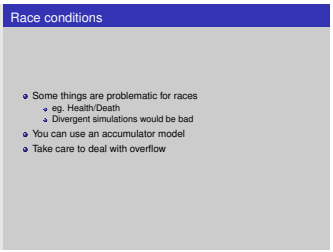



So, some things get troublesome if you use a simple model of dirtiness/ephemeral updates. Some things are order-dependent. eg. High frequency weapons are often handled with a firing on/off message. You don't want to get them stuck on. (This is a very common bug.)
It's usually important to impose an ordering on object creation and destruction - objects can't be destroyed before they get created. Short-lived objects can be problematic. So this is an area where dirty objects can't fully die but must become ghosts until their dead state has been fully ACKed.
Some objects just need a creation packet and then can be simulated independently on every machine. eg. short-lived ballistic projectiles (grenades) or stationary things (timed mines). Yes, it's possible that something could get in their path and result in two machines having divergent simulations, but if the projectile is going to explode soon anyway, odds are nobody will really notice.



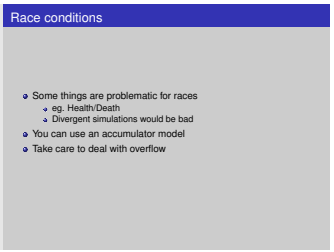

Neither client here really knows what's going to happen on the server. So Alice can't pretend Bob is dead, and Bob doesn't know whether he's alive. He's Schroedinger's Bob! It's a straight race, and the server must decide.

2014-02-23
Networking in the Real World
FPS issues
Race conditions




Either way, you need the server to adjudicate, and you want it to be as fair as possible. You will note that the network model as described is designed to send absolute state. Because we're working with an unreliable transport layer, it doesn't deal well with discrete events, especially not when they are very impactful to the game. It's hard to reconcile a divergent simulation of life/death.

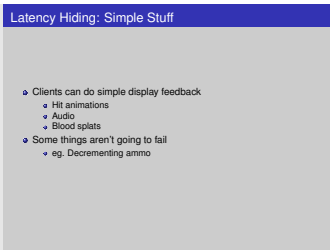
One way to solve this issue is to use an accumulator model for health and separately for damage. All the health Bob has accumulated this life is one variable, and all the damage he's sustained is another. Both of these variables can be easily dealt with in our existing replication model. The server merely has to adjudicate Bob's life, i.e. each frame determine whether the total damage he's sustained exceeds the total health he's accumulated. And the event of Bob's demise can be separated from the idea of his health and damage.



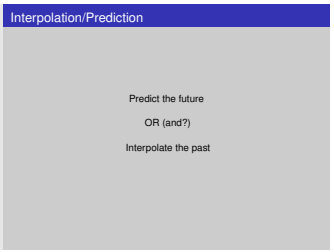

In general my approach to this sort of thing (where there is a policy decision to be made) has been to favour life over death. Players get more frustrated if they die; they are more willing to forgive someone miraculously living through a hail of bullets.

Likewise races occur over collecting powerups. In such cases my policy has been simply to give both players the powerup. Make everybody happy.





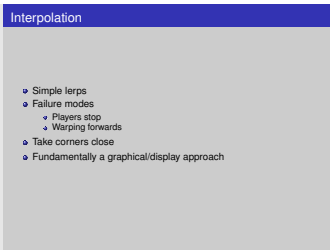

The simple stuff for latency hiding is in the audiovisual feedback that the client can give. You can show stuff that doesn't affect the game state - particles, audio cues, sometimes hit animations.




Your basic choice is this: to try to predict what is happening now based on the last information you received, or to treat the information received as a future event and interpolate towards it.

Both methods are viable depending on your game type. You might use both methods for different subsystems, eg. predict movement but interpolate animations.



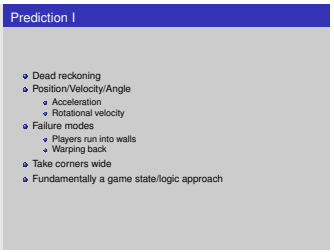



With an interpolation approach, you are always interpolating towards your current information. How divergent your current information is from your current game state generally controls how aggressive the interpolation has to be. (Alternatively, how broken it is going to look.)

The failure mode (if you don't get data) is that players approach their goal positions and stop - this generally looks OK. The recovery is likely to look weird though - a long warp forwards.

Generally because the player's actual position is ahead of where you are interpolating to, the standard systemic inaccuracy of this model is that players tend to cut corners close.

This can be done entirely as a graphical effect if your engine is architected that way. It might make sense to do this for things which are primarily graphical, ie. animation posing.



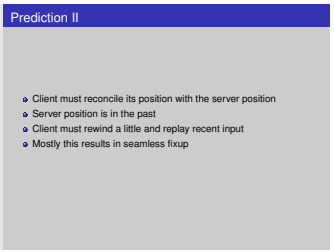

Dead reckoning can usually be made to look good based on just position, velocity and rotation angle. Occasionally a game will require predictions of higher-order dynamics - acceleration and rotational velocity.

Your primary failure mode is going to be that things overshoot and continue, run into walls, etc. If you have no caps on motion in the absence of timely data, this can look quite bad.

The standard systemic inaccuracy of this model is that players tend to take corners wide.

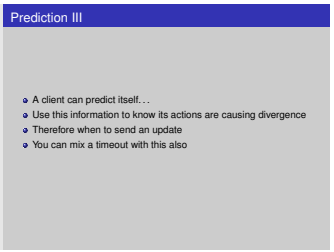
This can't really be done as a graphical effect in the same way as interpolation, because in the case of interpolating movement, you know that the player has already taken a path almost the same as the one you're interpolating. In the case of prediction, there might be obstacles ahead that prevent motion, and if you don't take account of them, you're going to clip through walls. So this approach is much more integrated into your game logic.






Typically the client can keep a circular buffer of recent inputs, rewind into the past to apply the server position, then replay. If the character simulation code is deterministic, this is usually pretty accurate and seamless.

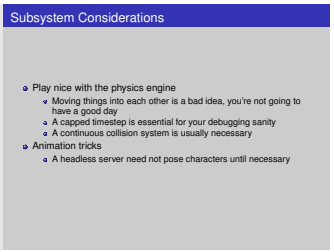
There may still be scenarios where a client needs to do further fixup because its position was altered by an external force. In such cases the fixup can be applied over several frames to avoid excessive warping.



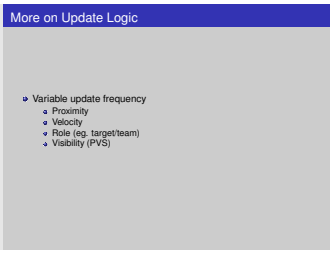

This can be a useful alternative to save bandwidth over regular updates. But it should be used with care - the divergence thresholds shouldn't be too large, considering that a roundtrip will still be incurred from the point where divergence is detected.

Last thoughts on prediction: important events may require rewinding time/snapping objects eg. death positions. There are circumstances when somebody is going to see/experience the "wrong" thing. Oh well.



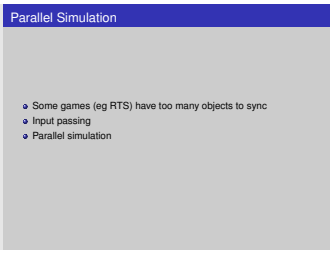



It's bad news when things interpenetrate; large forces usually result. On a headless server, animation can be optimized. It is possible for the server to slide the characters around and do only broadphase collision on their bounding boxes. At the point where a bullet collision occurs with a character, only then does the server need to compute the character pose to do the narrowphase collision (to determine whether it was a head shot).

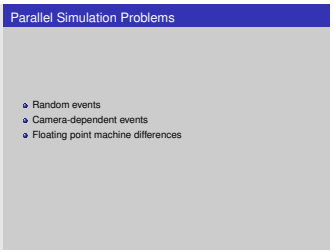



There are a lot of ways you can structure the update frequency and priority of your various game entities.
You can update based on proximity: this is a basic way to favour fidelity of close objects.
Another basic thing to do is to update faster objects more frequently, to hopefully achieve higher fidelity and fewer warping artifacts.
Proximity-based logic fails when you have scenarios like sniping where you're looking at something far away, or when keeping track of team-mates, so you can mix in role-based update logic.
Another thing to consider is visibility-based updates in general and the maintenance of PVS/occlusion information on the server. Again, if your game has ways to see remote locations this is something to keep in mind. And the fact that this imposes more work on the server.



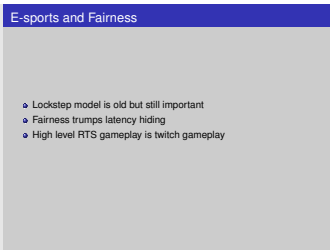



In an RTS, you have hundreds of units with complex state and AI. This is just too much to use a network model that sends state over the wire. So the solution is to send player inputs and make sure that machines can simulate the game deterministically in parallel.

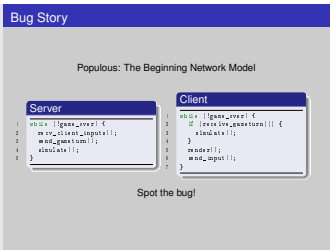



It is a very difficult and painstaking process to ensure that a simulation can proceed on two machines identically. The entirety of the network game is at the mercy of all the programmers. Many of whom don't have a sensibility for things that will break the network game.
The simulations will need to have synchronized random number pools. It is worth having two RNGs, one which is synchronized (for events that actually have gameplay impact) and one that isn't (purely for display effects).
Be wary of things that are triggered by players looking at them.
Using floating point is problematic and best avoided if possible. Machine differences, optimizations, instruction sets etc, mean that it is difficult to ensure identical results of floating point calculations. Physics engines are also difficult to constrain to be deterministic - they tend to have different behaviours on different speed machines. (This is also a problem with non-networked games: if your cutscenes use physics, prepare for one-in-a-thousand failures). (MoHPA story)
There is no real way to fix all the errors that cause network state divergence, other than by combing the code and logs.



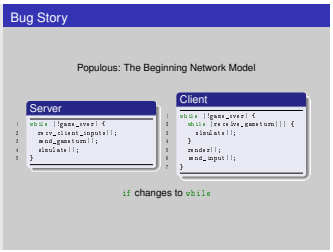



It is an interesting thought that FPS and RTS gameplay types swap over between low and high levels. ie. Low level FPS gameplay is twitch gameplay, and low level RTS gameplay is slow-paced strategic gameplay. But when you get to high level play, FPS is strategic - primarily based around knowing where your opponent is through audio cues and circuit timings - and RTS is twitch play: split-second microing of units and responding to resource challenges.
E-sports require fairness over and above latency hiding. It is not acceptable for the clients to miss a gameturn (input opportunity). So the old lockstep model still exists in e-sports titles: the server waits to gather all clients' inputs before processing and sending out the game turn. And the clients send null packets for game turns where there is no player input.



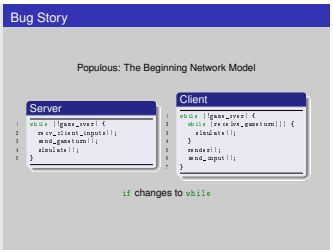

(Can anyone point out the bug?)






This bug was quite subtle and hard to spot because it took a while to show up in games. Games had to last a while before the bug gradually took hold. But the effect was that slowly, clients would get further and further behind the server.

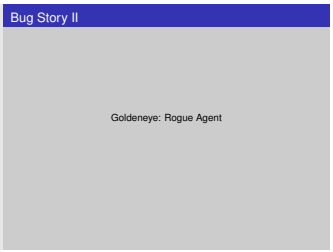
If a client had a spike such that a frame took a little longer to process or render, an extra gameturn could arrive from the server in that time. Now the client had two gameturns in its network queue. But it only ever processed one per render. So it would never catch up, and every time it took that little bit longer to process a frame, it would get that little bit more behind.




In one case we saw a client get behind by a full minute! The other player had already finished the game, and the player behind was still playing! I knew then that the code was robust!

Coincidentally I had this bug twice in my career: 8 years after Populous: The Beginning, I was working on Goldeneye: RA and I had a variant of this same bug. In that case it was easier to recognize because the more clients that joined a game, the more the clients would get behind.



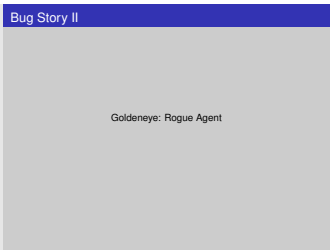



Goldeneye was a round-based FPS with up to 8 players. It was peer-hosted and ran on XBox and PS2.

8-player network games would sometimes ( 1 time in 500) hang on going into a round. One of the clients would simply stop after loading the level, stay on the loading screen and never actually enter the game.

I spent ages tracking that bug. I made rounds automatically cycle after 10 seconds, to give a higher repro rate. I put in a ton of logging. I scheduled after-hours play sessions where I would send out new builds to the team and set up 8-player games cycling to test.

I read the code forwards, backwards, and in my sleep. Finally I tracked it down to the point where I knew what the problem was; I put in some code to fix it, pushed out a new build, made my tests, and it was fixed.

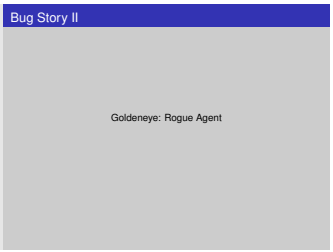



That was Thursday. Everything was quiet over the weekend. On Tuesday testers reopened the bug. It was still happening.

At this point I was sure it couldn't happen. And this wasn't hubris - I had been through that code every which way for the last 3 weeks. I fully understood the bug, and I had fixed it. And yet, it still happened. So I did everything I could to eliminate other sources of error. QA was using the right version. Nothing seemed to be amiss. Finally, I went back to the code, set a breakpoint in the function I'd fixed, and went to the disassembly view.

My fix was in the source. My fix didn't show up in the disassembly. I called over a couple of other programmers to check that what I was seeing was real.

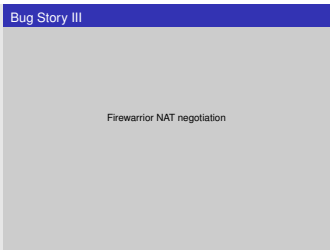
2014-02-23
Networking in the Real World
RTS issues
Bug Story II




Somehow the compiler was just ignoring my code. One of the other engineers checked the file in source control and found that the line endings were messed up. My single-line fix was preceded by a single-line comment - the compiler was eliding the lines so that it didn't see my code!

We fixed the line endings and the bug was finally dead.

Goldeneye gave us lots of bugs to fix. New gameplay features were going in quickly and I was constantly having to make them work over the network. Such was (is?) the life of a network programmer - many projects got networking bolted on near the end.

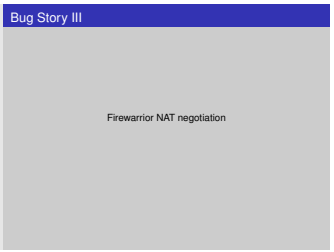



Firewarrior used a peer-to-peer network model, and we had to establish the mesh of connections, so NAT negotiation was pretty important.

We were using a GameSpy SDK to do NAT negotiation. And it just didn't work at all, really. Most NAT boxes (home broadband routers of the time) weren't letting us make connections. But some were. It was very puzzling.

This was a bug I just solved by thinking. There was nothing to be done in terms of gathering data. Eventually I realised that I was misunderstanding how NAT negotiation had to work - I was trying to get machines A and B to connect to each other simultaneously. What I had to do was to impose an ordering such that one machine was the initiator and one was the respondent.

2014-02-23
Networking in the Real World
RTS issues
Bug Story III




Once I did that, the bug was solved as if by magic, and the programmer who saw me fix the bug just by thinking was suitably awestruck!

This was a project where we had lots of issues with broadband routers. Compatibility testing was tough. Router firmware was sometimes buggy.

(You still can't assume that a player behind a NAT will keep the same connection tuple during a "connected" UDP session...)

## Thanks

The Real World: like Academia, except with smoke & mirrors & cutting corners & messy stuff.

Thanks for listening

bdeane@blizzard.com

Slides & notes available at

http://github.com/elbeno/networking-in-the-real-world

(none)