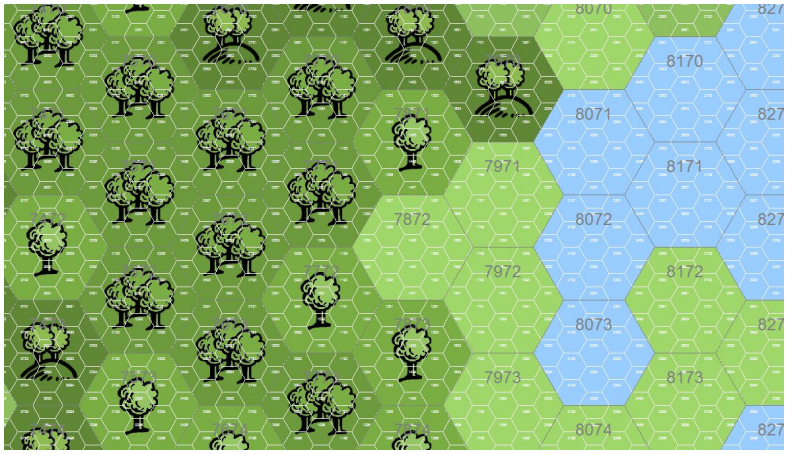
Networking Tutorial 2: Networking Issues in Games



Summary

Issues arising from the application of network technology to computer games are discussed. Focus is placed on the complexities of distributed simulations, with a focus on satisfaction of both real-time and consistency constraints. Zoning and Interest Management approaches are presented, followed by a discussion of dead reckoning.

New Concepts

Types of network game, the real-time condition, the consistency condition, zoning, interest management, dead reckoning.

Introduction

The first tutorial in this series provided an idea of how to communicate between systems across a network protocol. This tutorial explores why we might choose to do that, and the manner in which it is influenced by the restrictions our game's engineering places upon us.

We begin by discussing a little historical perspective regarding network gaming, and its evolution from turn-based tabletop RPG simulators to action-packed, real-time first person extravaganzas. We consider the different forms, in conceptual terms, that most network games can be divided into, and look at their specific requirements.

With particular attention paid to the requirements of real-time, distributed virtual environments, we explore methods of satisfying the real-time condition while still addressing consistency of world view across multiple clients. Specific approaches to this, including methods of zoning and interest management, and dead reckoning-based physics approximations, are presented in detail.

Network Games

The first networked games allowed multiple users to interact in a single gaming environment via text-based communications. In this sense, they were little more sophisticated than an IRC client (and, indeed, many IRC client variants were developed with this purpose in mind).

Multi-User Dungeons (or MUDs) popularised the concept from the 1970s onwards. To begin with, these were also text-based, following the tropes (and often the rule sets) of the established pen and paper role-playing franchises of the era. They often defined a 'character' with attributes which evolved or improved over time, and a list of abilities which expanded, or became more effective in some way, the more the player participated.

As with tabletop RPGs, games lasted a long time - a number of multi-hour sessions spread over weeks, months or years. And as with tabletop RPGs, people generally wanted to keep playing the characters they had invested as much time in, leading to high retention, if limited replayability - there are only so many ways you can shift the goalposts to keep material challenging.

While these early MUDs sometimes relied on a player adopting the role of Dungeon Master, later games such as DikuMUD included automated questing mechanics which heavily influenced modern, AAA MMORPGs. In fact, the term 'graphical MUD' was used to describe games like EverQuest and Ultima Online before the acronym MMORPG came into vogue.

Classifications of Network Game

Network games can be considered a sub-category of networked virtual environments (net-VEs), a term which describes two active areas of computing science research:

- **Distributed Interactive Simulation (DIS)** - In a DIS, the emphasis is placed on the accurate modelling of real-world scenarios. A lot of research into this area connects to military application, but the principles applied here are those most commonly associated with modern net gaming.
- **Collaborative Virtual Environments (CVE)** - In CVEs, the focus is on encouraging human interaction in collaborative working scenarios. Online games often do this by the nature of the mechanics employed (consider team-based PvP, or raid-level PvE content).

Online gaming borrows heavily from both of these areas, but generally in different fashions and to different levels, depending on the genre requirements of the game being made.

Distributed Interactive Simulation

The engineering employed in DIS technology tends to be most readily applicable to online gaming. The focus here is on presenting a virtual environment which can be manipulated and affected in real-time, and have that effect witnessed by (and impact on) other players.

Funding for research in this area is often military in origin, where the interest is in accurate, real-time warfare simulations for the training of troops, vehicle operators, and so forth. Of particular interest, and where the 'distributed' element comes in, are scenarios where your simulators aren't geographically co-located (e.g., your tank simulator might be in Reading, and your flight simulator might be in Geneva). It's unsurprising, given the popularity of FPS wargames online, that there's a good deal of cross-pollination between the military application and the commercial.

Collaborative Virtual Environments

In this area of research, the collaboration is key, and real-time considerations are less important. By its nature, a CVE enables geographically distributed users to cooperate towards a common goal. As such, in its purest form, a Skype call to a project member is a CVE, and many CVEs employ traditional, collaborative techniques (such as lectures, virtual workshops, board meetings, and so on).

In its more advanced form, a CVE might leverage VR technology to allow architects from different parts of the world to walk through an as-yet-unbuilt town-house. Similar techniques can be used to aid physicists or biochemists in the visualisation of complex molecular models.

The emphasis in this technology is improving the collaboration between users, rather than provisioning a real-time simulation. In this sense, online games can often be seen as borrowing design concepts from CVEs, and engineering approaches from DISs.

So, what about our game?

The nature of the network engineering issues a given software project needs to address is dependent upon the mechanical nature of the project. Consider a turn-based, multi-player civilisation-builder game, where all events in a given turn are considered to have occurred simultaneously (for the purposes of triggering a new turn), and the actual order in which they are resolved is dependent on player order (e.g., player 1's actions are resolved, then player 2's, and so forth).

In this game, there is little need for real-time constraints to be applied. As the time-step can be defined as 'however long it takes from the turn beginning to the last player clicking *End Turn*', the maintenance of a uniform frame-by-frame world view is unimportant.

But what if the actions of a player can affect another's mid-turn? For example, if Player 1's soldier can attack Player 2's builder as a part of Player 1's turn, surely Player 2's builder can't construct a road at the same time? In this instance, there is a need for some maintenance of consistency of world view - but not a maintenance of real-time simulation. We need to know that the attack is taking place *now*, so that it can be resolved (e.g., through a triggered 'combat' screen) before Player 2 ends his turn believing his builder is happily building. But Player 1 doesn't care if the sprite animation of Player 2's builder is at the same frame on his screen as it is on Player 2's.

Now, by contrast, consider a multi-player beat 'em up played across network. Here, twitch reflex matters, and actual position of entities on the screen needs to be as near as possible to accurate every update. We need to know if Player 1 has stunned Player 2 as soon as possible, so Player 2 doesn't think she got her shot off first. We need to know where Player 2 is, so if she is out of range of Player 1's stun she won't be caught by it.

These simple examples serve as a jumping-off point for you to consider what constraints might apply, and what conflicting priorities might come into play, for any game you want to include multi-player network functionality for.

Constraints of Network Gaming

The regrettable truth is that for almost every networked game, there will be some level of both consistency and real-time requirement. The nature of those requirements, and the level to which we prioritise one over the other, makes the process a complex balancing act.

If real-time requirements aren't met, realism suffers. If consistency requirements aren't met, user interaction becomes difficult or unresponsive (which also means that realism suffers). It's just as frustrating to miss a shot due to a lag spike, and then get shot, as it is to miss a shot because the server decided after the fact that you were shot first.

Real-time Problem

The real-time condition can be viewed through the following example:

- If player $\mathbf{U_1}$ fires a gun by time t, then **all** users should see $\mathbf{U_1}$ fire a gun by time t.

Consider the example of three users playing an online FPS. Player 1 runs in a straight line through the environment, appearing at six locations in consecutive frames. Players 2 and 3 will both have line of sight on Player 1 at the sixth frame, but message latency means Player 3's client receives the

update before Player 1, thus giving Player 3 and unfair advantage.

The first consideration when approaching solution of the real-time problem is to ask whether or not it needs resolving for the element under consideration. By this, we mean 'Is this element of the simulation one which requires real-time updating?'. So, an example of a consideration which wouldn't require real-time updating would be anything which is a constant for the simulation (a trivial example would be gravity, assuming it always points downwards).

The next consideration is to ask whether or not it is something which can be resolved in real-time client-side, without the need for the network to update the property. Examples of this are common - the evolution of a time-reactive skybox might possibly need periodic guidance from the server (to synchronise the clock), but the update itself can be managed client-side and the player won't notice any latency.

If the element does require real-time resolution, there are two approaches we can use to try and facilitate it:

- Zoning and Interest Management
- Predictive Modelling (e.g., Dead Reckoning)

Consistency Problem

The consistency condition can be viewed through the following example:

- If player $\mathbf{U_1}$ shoots player $\mathbf{U_2}$, then **all** users should see $\mathbf{U_1}$ shoot player $\mathbf{U_2}$.

Addressing this consideration is a function of both network engineering and game design. In the context of network engineering, it normally relates to the socket type (from Tutorial 1) we choose to employ. A trivial approach to satisfying ordering (maintaining consistency) is to use Transmission Control Protocol (TCP) to ensure FIFO (first-in, first-out) ordering, with a central server arbitrating all messages. This approach has been used by popular MMOs (e.g., World of Warcraft).

The central server, alongside a guaranteed receipt protocol, ensures all nodes receive the same messages in the same order (ignoring node failure, which is handled as a special case). The issue with this approach is that it runs contrary to the real-time requirement (the necessity to confirm receipt of packets before they can be committed to the queue and acted upon).

Datagrams (e.g. UDP) are used in place of TCP for games where twitch reflex is important (most FPS games). The latency TCP inserts can prevent FIFO ordering when using internet communication protocols. UDP does not have this problem, but adds the issue that messages can be lost and neither server nor client will know this has occurred. While messages can be lost with TCP, at least their disappearance does not go unnoticed.

In terms of game design considerations, we can often engineer elements of our game where consistency is crucial to minimise their real-time element. Consider the practise of looting in common MMOs - the artefacts claimed from a corpse are very rarely represented as physical objects in the game world until they have been allocated and equipped, meaning we avoid the issue of both player 1 and player 2 believing they picked up the Sword of Goblin-Cleaving. Instead, the server allocates the sword to the winning player's inventory, and the sword only becomes a physical artefact once the inventory equipment system places it on the character avatar.

Addressing the Real-time Problem

The remainder of this tutorial focuses on approaches for handling the real-time problem, as it is the issue most closely connected to actual engineering solutions (the consistency problem being more readily addressed by design decisions).

Additionally, many of the engineering solutions to address the real-time problem, and its impact on scalability, also assist in addressing the consistency problem by proxy. This is due to the fact they tend to focus on reducing the amount of information being transferred from server to client (or peer to peer), facilitating quicker updates and (ideally) minimising conflicts of world-view between clients.

Zoning

Zoning and interest management are approaches by which the amount of information required by a host to successfully enable local objects to participate in a virtual environment can be optimised. We can view this as careful selection of the data we choose to broadcast from host to client (or peer to peer). This is an attempt to address the scalability issues inherent to maintaining multi-player scenarios across a network.

The difference between zoning and interest management can be considered analogous to the difference between broad phase and narrow phase physics checks. Zoning reduces the number of clients that a server communicates with. Clients in different zones will never interact, in the same way that objects in different octree regions will not interact. Interest management is closer to a narrow phase check, where clients might interact, and we need to work out if they do.

High Level Zoning

This is the most straightforward approach to interest management, in that we functionally decompose our virtual space into clearly defined regions. This approach is commonly employed by MMORPGs, and the decomposition is often contextual. For example, in a science fiction MMORPG, we might segregate our zones as being different planets - a player stood on Hothderaan has no need to know any real-time information about a different player stood on Tatooruscant. Similarly, there is no means by which they might interact save one or both moving to a different planet (or zone).

Sometimes the changes in zone might be more mechanical in nature, and simply be a wibbly barrier at the edge of one zone, triggering a loading screen as we transition to another zone. The principle, however it might be shielded by design choices, remains the same in both cases.

We can consider high level zoning to be analogous to fixed, world-space partitioning in broad phase collision detection. The premise generally assumes a normal distribution of players throughout the game world, thereby a normalised distribution of server workload across multiple servers. The ability to leverage multiple servers reduces the workload per server, making real-time constraints more manageable and, by extension, helping alleviate consistency problems.

On the other hand, this approach is vulnerable to the same problems as world-space partitioning. If all players gravitate to one zone, you gain the overhead of management with none of the benefits. As such, where this approach is connected to progression (and it often is, with recognised 'high level zones' in many MMOs), it can artificially engineer its own worst-case scenario, with all players eventually reaching maximum level and occupying the same zones.

An approach to dealing with this issue which is increasing in popularity is to normalise difficulty across zones, through player character 'scaling', and include content that compels players to revisit 'lower level' areas in general play.

Spatial Zoning

Spatial zoning is more complex than high level zoning, and is generally employed beneath it. In general terms, it attempts to subdivide the open world of a virtual environment without the 'loading screen'. In this sense, the server must recognise when a player is likely to enter its management area, and adapt to that, without the crisp delineation of a load-screen facilitated hand-over.

The map is logically divided into multiple zones, with each zone encompassing players in the same vicinity. A player moving from one zone to the next is disconnected from one server, and joins another, with a brief period of overlap (akin to overlapping broad phase regions). Zones are directly geographically connected, often using simple geometric shapes (rectangles, squares, hexagons). Simple shapes

are chosen because they enable simpler position checking, and are easier to interconnect than more complex polygons.

The density and shape of zones can be varied based upon the rate at which a client is expected to move between them, and the predicted occupancy of the zone compared with the load the server is able to bear. The goal of the approach, from a design perspective, is to ensure that a client never 'feels' like she is changing servers - no loadscreens, seamless transition, no performance lag.

Sustaining this can be difficult, requiring some overlap between zones, necessitating duplication of data to avoid players 'popping' into existence as they cross a boundary. As the servers represent fixed geographical regions, however, design decisions can be taken to minimise the effect of this. For example, in some early and successful MMOs, random boulders would be placed at the point of transition, with corridors formed from natural features (e.g. mountains) around them - this prevented clients from one region seeing the other region until they were around the boulder, while still maintaining the illusion of continuous play.

Megaservers - Zoning, Backwards

These approaches can also be used to merge low-population virtual spaces, particularly in games where sharding has been employed and the same virtual space is duplicated several times. Many MMOs follow this model, and doing so both reduces the overhead cost of spinning up unneeded server resources, and improves gameplay experience (by improving the odds of encountering other players - which is rather important for a multi-player experience).

Interest Management

The simplest form of interest management can be considered analogous to spatial zoning described above. In this section, we discuss more reactive approaches to interest management.

Behavioural Modelling

We can adapt the area of interest of a client based on the client's current behaviour or context. Let's consider the example of a client flying a plane, and another client driving a jeep. The two clients are, for a given instant, in the same spatial zone.

The client in the jeep travels at a maximum velocity of 100kph, while the client in the plane might travel as fast as 500kph. The client in the jeep is close to the ground, giving it a poorer field of view than the client in the plane, meaning it can see fewer significant features even with equivalent draw-distances. And, of course, the plane has air-to-surface missiles, while the jeep has a tail-mounted machine-gun - as such, the plane has a far greater area of potential influence than the jeep.

As a result, the area of interest of the client in the plane should be modelled differently to the area of interest of the client in the jeep. This sort of area of interest modelling is not particularly well-explored outside of academic research, primarily because it is more difficult to leverage meaningful performance benefits with this approach than it is with, for example, geographical partitioning.

Publisher-Subscriber Model

This common approach to interest management treats clients as both publishers of their own actions, and subscribers of other clients' (or server-managed) actions. The principle as applied to network gaming is that a client becomes a subscriber to the events published by those clients that it can perceive (through one or more domains, such as visibility, hearing, etc.). When a client can no longer perceive the events published by another client, it ceases to register them.

As well as reducing network traffic (at the expense of increased server overhead), this approach can be used to counter cheating in online games, as the server dictates not only what events the client software shows on screen, but what information concerning events is sent to the client software. It is possible without an approach of this type that additional information concerning other players could be accessed by third party software (e.g., letting a client 'see' a notionally invisible enemy).

Aura-Nimbus Approach

This approach is a commonly employed means of identifying relevant publishers and subscribers. Under Aura-Nimbus, a client is considered to have two radii:

- **Aura** - the client's range of influence
- **Nimbus** - the client's range of interest

If client **A**'s range of influence intersects with client **B**'s range of interest, client **B** receives updates regarding client **A**. In many ways, this is analogous to a sphere-sphere check, though the regions can be defined as design requires. Additionally, this approach can be contextually enhanced (for example, an invisible enemy can be identified by a simple boolean flag; a client with reduced visibility might have its nimbus region narrowed).

The drawback of Aura-Nimbus is that it lacks scalability. If all entities auras and nimbi overlap, there is no performance benefit, only the overhead of performing the associated checks.

Other Perception-Based Approaches

By now, you should have realised that all area of interest optimisations ultimately come down to the reduction of network traffic based on the perception (and possibly predictions relating to the perception) of a client. As such, you can employ any of the appropriate algorithms you've learned over the course of the programme in an effort to manage what data hosts send to clients.

Some examples are visibility checks, reachability checks, or simple proximity checks. As with physical interface algorithms, these are a toolkit from which you can choose the most appropriate selection to meet the requirements of your project. Also, as with physical interface algorithms, you need to make a sensible judgement between the overhead each of these checks generates, versus its performance benefit to you in terms of client update rate.

By now, you should certainly appreciate the utility provided by being able to dynamically adjust the size of data packets sent between host and client (as introduced in the previous tutorial). Without this ability, we would greatly reduce the benefit to our simulation provided by many of the techniques discussed here.

Dead Reckoning

Dead reckoning as a concept attempts to address a key issue of distributed real-time simulations: there can be no 'god-like' view of every object's absolute position at any given time in a networked game, which is shared by the server and all clients in the same instant. Such a scenario can only occur if you have a zero-latency network, and the speed of light (not to mention engineering reality) strictly prohibits that.

As such, all any client has at any given time is its own perceived 'truth'. It follows from this that the best-case scenario is a believable 'best-guess' as to where any entity ought to be based on what we can infer about its movement. The term *dead reckoning* itself comes from 'deduced reckoning', and is essentially an application of the same principles we have employed in the construction of our physics engine - which, we recall, is about making a believable, best-guess as to where entities ought to be, based on what we can infer about their movement.

Prediction in Dead Reckoning

General solutions to dead reckoning employ derivative polynomial equations, in the same way that our physics engine does. The 0th order polynomial (not a derivative, obviously) would simply refresh position directly from a server broadcasted value for position - we call this frequent state regeneration, and it is as inappropriate in our dead reckoning system as projection-based collision response is as the basis of our physics system.

Instead, we might opt to use the first order differential - object's instantaneous velocity - in our computations. For example, we know an entity's velocity $\mathbf{v_t}$ at time t, where t was the time the entity's owning client broadcast that velocity to the server. Until we have a further velocity update from the server at some unspecified future time *(t+x)*, we will simply update the entity based on $\mathbf{v_t}$.

More commonly, second order differentials, basing motion on the entity's acceleration, are employed. The principle remains the same, save that our Newtonian updates for the entity's behaviour are based upon the server broadcasting its acceleration. The approach taken should be based upon which variable in our simulation is least volatile - in a game where acceleration oscillates heavily every fraction of a second, but the overall changes to velocity are minimal (e.g., altitude maintenance in a flight simulator), velocity might be a better predictor of motion.

It is important to remember that our entities often have angular motion to consider as well as linear, and in many modern games prediction of this is also crucial to gameplay experience. In an FPS, for example, you want to know that the enemy you believe has his back to you actually *does* have his back to you as you sneak up for a stealth kill - not that he's actually facing you, and orientation just hasn't refreshed yet.

It is important to remember that dead reckoning in this fashion is most appropriate for entities whose motions are difficult to predict (player avatars freely moving through an environment). We do not need to employ dead reckoning of this sort if a player avatar is sat in a monorail car - we can instead update the player's position with a reasonable degree of certainty client-side based upon the known travel path of the monorail. Similarly, we should keep in mind that 2D predictions can often suffice in 3D simulations, if characters are all running across reasonably flat terrain.

It should be obvious to us now that dead reckoning potentially inserts errors into our system in an effort to maintain the real-time conditions necessary for playability. We now discuss how those issues are addressed.

Convergence of World Views

Consider the example of two player-controlled aeroplanes which are on a collision course with one another. The clients are geographically distant, and communicate via a central server. Player **A**'s client performs its dead reckoning update the instant prior to collision, and predicts that the planes have collided.

Player **B**, however, swerved at the last moment. As a result, player **A** and player **B** have conflicting world views - **B**'s client shows both planes evading one another, while **A**'s shows both dying in a fireball. These two conflicting world views cannot be reconciled and, as such, we need to try and design our game to prevent them occurring.

Sometimes, approaches to this are completely design-related and preventative. An example is cast bars in MMOs. These serve the purpose of giving the server more time to verify that an event is going to occur, and provide a time in advance that it will occur, limiting the possibility of conflicting world-view. Of course, it can still happen - try walking the split second before your cast-bar fills up in an MMO, and you'll often find the casting completes for the server and your opponents, even though you've notionally interrupted it.

Similarly, often we have no option but to employ frequent state regeneration for certain variables - like `AmIDead`. In the case discussed above, we would likely wait for the server to confirm the destruction of the planes before client **A** showed the explosion. Alternatively, we could limit the motion of the planes in such a way (say, giving them a slow angular velocity) that **B**'s evasion would be impossible in the first place, meaning the server could happily know an update or two in advance that they're going to die.

In scenarios that are less dramatic, however, we have other options to converge world view. Let us consider the case that, instead of colliding, **A** and **B** are travelling in formation. **B** adjusts attitude slightly on her client, while **A**'s client continues to update **B**'s position using dead reckoning. This

leads to a small discrepancy between where **A** believes **B** is at this instant, and where **B** believes she is.

When the server broadcasts the new information regarding **B**, **A**'s client recognises this discrepancy, and needs to adjust to it. One simple approach is the zero-order (snap) approach, which would simply move **B** to the position the server says she should occupy. This would necessarily be adjusted using dead reckoning for the period between **B** sending this update to the server, and the server broadcasting it to **A** - otherwise **A** might see **B** leap to where **B** was a half a second ago on **B**'s client, which would be even more inaccurate.

Some MMOs employ this approach, particularly when addressing lag-spikes for player character motion in open world. Generally, though, this is only appropriate in scenarios where you do not care about maintaining real-time believability (or, as in the case of the lag spike example, believability has already been lost and you just want to get the simulation back onto an even keel).

Alternative approaches involve interpolating the results over time. A linear interpolation, for example, would identify some convergence point in the future based on the latest predicted path of **B**, and linearly animate **B** towards that point over the intervening period. While better than snapping, this approach can still generate unnatural motions as entities suddenly change direction.

More modern engines generate a spline than smooths the animation between the current position at the computed convergence point. This is more expensive (with expense connected to the smoothing of the curve), but generally provides better results.

All of these approaches, however, are likely to be inaccurate. By the time we have reached the convergence point, **B** might well have changed direction again. Of course, we can react to that in the same way, computing a new convergence point and spline to reach it, but we will only ever reach a truly consistent world view if both **A** and **B** fix their motion (i.e., they either land and come to rest, or set a fixed velocity/acceleration and orientation and never adjust it).

In this sense, our virtual environment is an eventually consistent system, similar to the linear solver employed to resolve constraints in our physics engine.

Implementation

Consider the concept of dead reckoning. Explore the implementation of various orders of dead reckoning within your project.

Tutorial Summary

In this tutorial, we discussed the issues directly affecting game development when introducing multiplayer technology. We presented some problems that specifically affect real-time distributed simulations, and outlined some popular approaches to accounting for them.