Multiplayer Games Programming Literature Review - Jacob Costen (23025180)

Introduction

Developing networked games introduces significant challenges, from management of game state within the game client, to synchronisation of the game state between clients and servers. Multiplayer programming requires heavy use of concurrent programming which comes with its own complexities, and different game genres often present different requirements which developers must accommodate when designing networking code (simulation games such as Factorio present different challenges to first-person shooters such as Call of Duty). This review will consider some of the problems and techniques that arise both with multi-threaded programming, and networked multiplayer game programming.

Literature Review

Networked games require the use of concurrency, since network events happen asynchronously to game events and updates. Concurrent code introduces new programming challenges which arise from multiple execution threads accessing shared resources simultaneously.

Race conditions occur when multiple threads try to access and modify data simultaneously (Netzer and Miller (1992))[1]. For instance, where two threads attempt to increment a counter `i` simultaneously, `i` is read by thread A, then by thread B, then each thread adds one and writes the result back to `i`. The counter only increased by one, because the read operation on thread B occurred before the write operation on thread A. When the two threads are almost synchronised with one another, this race condition becomes apparent and one of the increments is missed (potentially causing further problems later in the code).

The simplest synchronisation structure which is used to prevent race conditions is a mutex. Mutices (a contraction of 'mutual exclusion') may be 'locked' by one thread, and any other attempt to lock that mutex will block until that mutex is released by the original thread. In C# mutices can be constructed automatically using the `lock(object) { }` syntax, and will be released when execution exits the code contained in the lock block (Agafonov (2016))[2]. However, according to Voss, Asenjo, and Reinders (2019)[3], the use of mutices and other synchronisation structures should be minimised when building high-performance code, since they often introduce bottlenecks where concurrent threads waste time waiting for resources to become available. Instead, algorithms should be restructured to eliminate the need for these structures.

The use of locking structures as described above introduces a problem called deadlocking. This occurs when thread A locks resource A, and waits for thread B to release resource B in order to lock it, while thread B is simultaneously waiting for thread A to release resource A before it will release resource B. The result is a deadlock, where both threads wait on each other endlessly. Zhou, et al. (2017)[4] develop a method for detecting potential for deadlocks based on the key property that locks must be nested (one resource held while waiting for another) in order to generate deadlock conditions; however they also note that detecting deadlocks reliably is very difficult, especially using standard testing, due to the fact that both threads need to acquire their first resource before either tries to acquire their second resource. The authors further conclude that the best way to resolve deadlocks like this without significantly redesigning the software is to merge multiple timing-sensitive fine-grained locks in to a single coarser lock: meaning that when thread A locks resource A, it is reserving access to resource B at the same time, through a single synchronisation object (e.g. a mutex).

When creating a networked game, developers must choose a transport layer protocol over which to communicate gameplay data. The two predominantly used protocols are TCP (transport control protocol) and UDP (user datagram protocol). While older games - such as World of Warcraft, as studied by Wang, et al. (2012)[5] - tend make

use of TCP, many other titles use UDP instead. The key difference between the two protocols is that TCP guarantees the arrival, integrity, and order of packets as inherent functionality, while UDP leaves the implementation of this up to the developer. This means that while TCP might be more reliable, it is often slower, requiring transport-level acknowledgement packets to be sent back and forth (Glazer and Madhav (2016))[6]. Thus, the use of UDP is preferred, as it gives more control over exactly how much reliability is prioritised over speed.

While a combined approach may initially appear ideal for a game where different packets have varying priority, Saldana (2015)[7] finds that the use of TCP and UDP together actually harms network performance, since TCP connections tend to fill a connection's bandwidth to the point that the time taken for multiplexing those connections with background UDP traffic leads to poorer network performance and increased delay in the game.

A challenge faced by online multiplayer games is in realtime replication of game worlds. Specifically, when a gameplay element changes on one client (for instance, a player moves in the world), that information must be sent to other clients in order to correctly replicate that movement. However, periodically updating the remote player's position results in jerky movement, so prediction techniques must be employed to move the remote player continuously, even though position packets only arrive periodically.

One of these techniques is dead reckoning, which estimates an objects position based on past motion or velocity data and the time since the last position update (Shi, Corriveau, and Agar (2014))[8]. This minimises the jump in position when the next update arrives. Dead reckoning also has the advantage that it does not require network packets to arrive any time the player changes direction or velocity, minimising the effects of packet loss over the network.

More precise algorithms may be applied depending on the specific use case; games like snake maintain continuous movement and so simple continuous-motion prediction may be used. Conversely, many gameplay environments (particularly first-person shooters) require more advanced algorithms to accurately predict player movement, such as the 'EKB' algorithm proposed by Shi, Corriveau, and Agar (2014)[8], or the neural-network-based approach taken for predicting high-speed vehicle motion in multiplayer games by Walker, et al. (2024)[9]. Both of these examples illustrate that the best algorithm is often one which is developed specifically for the game it is to be used in; this is something machine learning offers potential with due to its ability to be trained on existing data.

Maintaining fairness in multiplayer games is one of the largest challenges faced by developers. This manifests in two key ways: cheating and lag. Players who experience greater lag (usually measured as higher ping, the time required for a packet to make a round trip between the client and server) are significantly disadvantaged compared to other players. Geographical distance to the server, quality of network hardware, and the type of physical link (both inside and outside the player's home network) contribute to this, and this is often felt disproportionately by poorer players. One way game developers can combat this is by normalising player lag to prevent players on faster connections having a significant advantage. Zander, Leeder, and Armitage (2005)[10] developed a tool which is able to reside between the game server and the internet to implement this behaviour.

Cheating is a wide field of problems in the multiplayer games industry, and these are mitigated in a variety of ways. Lehtonen (2020)[11] categorises cheating into soft (making use of bugs or exploits to gain an advantage) and hard (making modifications to the game client or using additional tools) cheats, and notes that preventing hard cheats is the objective of most anti-cheat systems. Often soft cheats are fixed by developers releasing game patches to remove exploits.

Anti-cheat software may be applied either server-side or client-side (often both): server anti-cheat operates on the principle that the client should never be trusted; every action and packet should be validated before it is accepted (for instance, many Minecraft multiplayer servers feature anti-cheat tools which will disconnect a player if they have been airborne for too long and are deemed to be 'flying', a common form of cheating). Server-side anti-cheat may also involve analysis of activity logs to detect suspicious patterns, such as the use of bots, which may produce excessively regular or perfect movement, aim, or activity.

Client-side anti-cheat usually involves having a side-process which monitors the state of the game client to detect changes to its memory contents. For instance, an anti-cheat system may detect when a cheater attempts to inject new code into the game client, and report this activity to the server (Lehtonen (2020))[11]. However, this often introduces serious security problems, since in order to do analyse the game client's memory, the anti-cheat tool must run at the kernel level of the host system. Poorly secured, unstable, or resource-intensive kernel-level anti-cheat systems may cause harm to players' machines, and provide a vast attack surface for hackers via the 'Bring Your Own Vulnerable Driver' method for gaining access to a system; this is of particular concern since these anti-cheat tools interface with the network constantly (Basque-Rice (2023))[12].

When dealing with cheaters, live service game providers such as Blizzard and Riot Games will often ban cheaters in waves, rather than individually as they are detected. This has a number of advantages: dramatic monthly ban numbers have a greater psychological deterrent effect for potential cheaters; cheat developers may be financially damaged by an influx of refunds from players who bought cheating tools from them; and cheat developers have a much harder time identifying why or when their tools were detected by anti-cheat systems, slowing down the development of more advanced, harder to detect, cheats (Laato, et al. (2021))[13].

Conclusion

The state of multiplayer gaming has changed significantly over the last two decades, following improvements in network infrastructure, development of standardised libraries and protocols, and changes in studio priorities and consumer habits. In particular, intrusive anti-cheat systems have become increasingly common.
While many of the underlying challenges involved with synchronising dozens of players from geographically different places to within fractions of a second of one another remain largely the same, machine-learning-powered prediction algorithms present an interesting avenue for improving in-game experience for players.

Bibliography

[1]: Netzer, R.H. and Miller, B.P. (1992) 'What are race conditions?', *ACM Letters on Programming Languages and Systems*, 1(1), pp. 74–88. doi:10.1145/130616.130623.
[2]: Agafonov, E. (2016) 'Multithreading with C# Cookbook'. *Packt Publishing Ltd.*
[3]: Voss, M., Asenjo, R. and Reinders, J. (2019) 'Synchronization: Why and how to avoid it', *Pro TBB*, pp. 137–178. doi:10.1007/978-1-4842-4398-5_5. [4]: Zhou, J. *et al.* (2017) 'Undead: Detecting and preventing deadlocks in production software', *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 729-740. doi:10.1109/ase.2017.8115684.
[5]: Wang, X. *et al.* (2012) 'Characterizing the gaming traffic of World of Warcraft: From game scenarios to Network Access Technologies', *IEEE Network*, 26(1), pp. 27–34. doi:10.1109/mnet.2012.6135853.
[6]: Glazer, J.L. and Madhav, S. (2016) *Multiplayer Game Programming: Architecting Networked Games.* New York: Addison-Wesley.
[7]: Saldana, J. (2015) 'On the effectiveness of an optimization method for the traffic of TCP-based multiplayer online games', *Multimedia Tools and Applications*, 75(24), pp. 17333–17374. doi:10.1007/s11042-015-3001-y.
[8]: Shi, W., Corriveau, J.-P. and Agar, J. (2014) 'Dead reckoning using play patterns in a simple 2D multiplayer online game', *International Journal of Computer Games Technology*, 2014, pp. 1–18. doi:10.1155/2014/138596.
[9]: Walker, T. *et al.* (2024) 'Predictive dead reckoning for online peer-to-peer games', *IEEE Transactions on Games*, 16(1), pp. 173–184. doi:10.1109/tg.2023.3237943.
[10]: Zander, S., Leeder, I. and Armitage, G. (2005) 'Achieving fairness in multiplayer network games through automated latency balancing', *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pp. 117–124. doi:10.1145/1178477.1178493.
[11]: Lehtonen, S. (2020) 'Comparative study of anti-cheat methods in video games', *University of Helsinki, Faculty of Science.*
[12]: Basque-Rice, I. (2023) 'Cheaters Could Prosper: An Analysis of the Security of Video Game Anti-Cheat', *Doctoral dissertation, Abertay University.*
[13]: Laato, S. *et al.* (2021) 'Technical cheating prevention in location-based games', *International Conference on Computer Systems and Technologies '21*, pp. 40–48. doi:10.1145/3472410.3472449.