Networking in Early Video Games

Benjamin Schmidt
University of Calgary
Calgary, AB, Canada

John Aycock
Supervisor
University of Calgary
Calgary, AB, Canada

ABSTRACT

We discuss three of the earliest networked video games – *Maze* (1973), *Sopwith* (1984), and *SGI Dogfight* (1985) – each with distinctly different networking architectures. After analysis of the source code, we break down implementation details and compare similarities and differences in their approaches to real-time communication between players. We also use first-hand evidence backed up by implementation details to shine light on flaws in the systems.

CCS CONCEPTS

• **Social and professional topics** → **History of software**.

KEYWORDS

Networked video games, Networking, History of video games

ACM Reference Format:
Benjamin Schmidt and John Aycock. 2024. Networking in Early Video Games. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Since the very early days of video game history, players have competed or collaborated with each other in multi-player game modes [?]. This was achieved by connecting multiple separate input devices or a single shared device to a single computer, and proved effective for players near enough to the computer but inadequate for any distance beyond the length of an input device's cable. It was also infeasible for more than a handful of players to play together. Modern networked video games have solved these problems, allowing hundreds to millions of players to play together simultaneously from anywhere with an internet connection. We will discuss the early steps of the journey to achieve this by looking at three of the earliest networked video games – with source code available – from the 1970s and 1980s.

These early video games are *Maze*, *Sopwith*, and *SGI Dogfight*. *Maze* and *Sopwith* were created before the ubiquity of the TCP/IP stack, and *SGI Dogfight* in the very early days of TCP/IP. We will discuss their solutions for networking without reliance on the standard protocols we know and rely on today. This will be achieved through static analysis of the source code and reports from the original developers or players. We aim to emphasize the significance of each game, its unique solution to networking, and any issues that it may have faced.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 RELATED WORK

Much of the discovered work in the field of networked video games describes systems with broad application that can be implemented in a number of ways. For example, **?** describe a general architecture – NetEffect – with which multi-user video games or virtual worlds can be created, giving the example of a particular implementation built to demo NetEffect. The authors describe a number of interesting ideas, such as distributing the virtual world over multiple servers which handle subsets of that world, peer-to-peer voice chat, load balancing, and a number of other techniques primarily intended to reduce network traffic, especially inter-server traffic. However, it doesn't appear that NetEffect was used widely, and its emphasis on scalability was not something we found in the earlier networked video games that this paper analyzes. Furthermore, we will discuss concepts specific to the various games studied and perform a comparative analysis on their implementation, effectiveness, and flaws from a contemporary perspective.

? follow a similar pattern. They describe Virtual Environment Realtime Network (VERN) – an extension of DARPA's SIMNET which enabled physically distant users to interact and communicate in a virtual environment. VERN, like NetEffect, is a framework in which to build virtual worlds, and so this paper is once again different to what we present.

In other literature, the focus is on more specific solutions rather than generalized frameworks. **?** describe an alternative approach for positional information transmission – instead of sending real-time position data or using dead-reckoning, position data with a timestamp is periodically sent instead, and clients estimate the object's position, velocity, and acceleration accordingly. They claim this solves an problem exhibited by *SGI Dogfight* where the frequency of packet transmissions (tied to frame rate) affects smoothness of the rendering of other players. It would also mitigate the network saturation caused by the game's traffic when enough players were present.

?, a Software Engineer at Valve Corporation (then Valve Software) presents solutions for latency compensation in the context of an architecture with a single authoritative server and many "dumb" clients. In essence, he describes how a user can feel as if they are interacting with other clients in real-time as they would in a peer-to-peer architecture, having the security benefits of a server-orchestrated architecture – clients cannot broadcast their own state as an invalid one, for instance teleporting their character or changing the state of their score – while mitigating the downsides involved with high latency or connections.

? and **?** both note the use of dead-reckoning, in which clients only broadcast changes in direction or speed of a controllable game

object, and are otherwise assumed to be moving at the same speed and in the same direction as they were at the time of the last broadcast [?]. This is in contrast to earlier video games which instead updated the character's current location and direction every time it changed, which naturally used significantly more bandwidth than dead-reckoning.

Overall, the body of literature covers generalized topics – either broad systems or specific implementation details which may avail developers when building a networked game.

Other related works discuss the history of video games, in general or for specific games. For example, the IEEE Annals of Computing History Volume 31, Issue 3 is dedicated to the history of video games and contains articles detailing the history of *Pong* (1972) [?] and *Nimbi* (1963) [?], and how various display technologies were used in video games [?] among others.

Our work is adjacent to the cross-disciplinary field of archaeogaming, where old video games are treated as archaeological artifacts to be preserved, restored, and studied using archaeological techniques within the context of their time and the people who played them [?]. We lack the archaeological connection, but hope to accurately portray the technical details of the networked video games we discuss.

Works in this field consist of uncovering implementation details of retro games as well as the human aspect behind their creation and enjoyment. [1] For instance **?** reverse engineer the central maze-generation algorithm behind *Entombed* (1982) for the Atari 2600. By discovering a bug in this code, they were able to show reuse of this code in other Atari games, and used this as a vehicle to discuss the programmers behind it.

3 PRELIMINARY

In this review, we will examine the networking implementations used by a variety of early networked video games, assess their implementation details and speculate on their performance and potential flaws. For the purposes of this research, a networked video game may be defined as a video game in which two or more independent clients running the same code interact over a real-time link. Each client therefore handles I/O, networking, and at least some portion of game logic.

Many early video games supported multiplayer functionality, allowing players to compete or co-operate in real time. However, these early solutions were typically multiple input methods connected to a single device with a single display. Although this was effective – and indeed is still used in contemporary games – geographically separated players had no means by which to interact. The PLATO system featured the ability to host multiple graphics terminals (thin clients) which served to simply relay input and output to a mainframe which performed the processing. A number of games were built using it, allowing users to play together even if they were in a different room, and as many mainframes running PLATO were networked together, the terminals were effectively networked together as well [?]. Despite the importance of these games, they do not fall under our specifications of a networked video game – the network code is restricted to the PLATO system on the mainframe itself rather than the client.

Within the scope of our definition, there were some common technologies used by the games we will examine. RS-232 serial connections were used, either as a direct connection between two clients (the serial cable functioning as a null modem), or connecting all clients to a central mainframe which could would broadcast data received from one client to all others. According to the specification, these cables were limited to 20kbps transfer speed [?], although many were used in unintended ways and specified limits and uses weren't always followed. These cables were effective as null modems when connecting two clients, but this is not part of the RS-232 standard, and had speeds lower than alternatives (like Ethernet), making them not suitable for network transfer beyond simple peer-to-peer two-player implementations.

In 1980, Ethernet was published as an open standard by a partnership between Xerox, DEC, and Intel [?] , being standardized by the IEEE in draft form as IEEE 802.3 in 1983 [?]. Early Ethernet used a shared cable where all connected devices would receive the same data, resulting in collision risk and shared bandwidth. Despite its limitations, Ethernet was simpler, cheaper, and more effective than its competitors, and it became the *de facto* standard for wired local communication [?].

Lastly, Transmission Control Protocol and Internet Protocol (TCP/IP) was made the sole transmission protocol on the ARPANET in January 1983 [?], which persisted as ARPANET transitioned into the public internet, and of course is still in use today.

Before discussing the games, we will elaborate on the methodology used to create the final list. Wikipedia was chosen as the source for finding games due to the amount of data available and well-organized categorization. A Python program (Source Code available on GitHub) was written by me, making use of the `pywikibot` library [?] to pull the `.wikitext` file for every article in the categories "Category:<year> video games" for every year between 1970 and 1989 (inclusive) i.e., the article for every video game published between 1970 and 1989. Once this list of nearly four thousand games was extracted, keywords were searched for within the article text by the program searching for at least one reference to a keyword in the text. More complex means of identification were attempted but were found to reject a large number of valid candidates, and so the aforementioned method was used in conjunction with a manual review of the final short-list of just under two hundred video games. This manual review consisted of discarding articles in which the only references to the keywords were unrelated to the game (e.g., citations of publications with "network" in their name). Sixteen games were left, and of these, five were chosen for their available source and significance. The three games discussed here are the first three in chronological order by release year.

4 FINDINGS

4.1 Maze

Maze (or *Maze War*) is the oldest of the video games we discuss in this review. First created in 1973 by high school students working at NASA in the summer of the same year [?], it is the first

[1] This description does not encompass all the work involved in archaeogaming. See [?] for a more detailed explanation of archaeogaming.

networked video game by our definition. [2] Players would simultaneously occupy a shared maze, and had the ability to move forward, backwards, left, and right, rotate in 90° increments, and fire missiles in the direction they faced. The objective was to shoot other players, which would increase the shooter's score if successful, with scores being displayed in an in-game leaderboard. *Maze* also supported a few additional features including text chat.

Maze was written for the Imlac PDS-1 in its native assembly language, and initially supported multiplayer functionality by connecting two PDS-1s with a RS-232 serial cable [**?**]. In 1974, development continued at MIT, where multiple PDS-1s were connected to a central DEC PDP-10 server. This version of *Maze* is what we will discuss henceforth, as the original source code is fully available and has been open sourced.

We will briefly discuss the server's function. The server could support up to eight PDS-1s, each connected to it by a separate serial cable, and enabled initial setup and message "routing" [**?**]. Upon connection, the game and a custom maze (if specified) were downloaded from the server to the client, and players were connected together. From this point on, the server acted as a repeater of characters from a PDS-1, forwarding them to all connected devices except the sender. All game logic and network code is therefore handled by the client, with the exception of "robots" – characters controlled by the server instead of a human player. As PDS-1s were common in institutions that formed the ARPANET, cross-institutional play was possible for mainframes connected to it, and so the server would handle this as well.

Maze used the *Maze* Protocol to send and receive information about player and game state. Each client would maintain a copy of the global state locally. Information about other players was updated when receiving messages as well as when the local client shoots another player. Clients would simply receive characters from the serial connection by calling an instruction to move the input from the buffer to a register, masking to the lower 7 bits – discarding the most significant bit to fit the ASCII standard – then handling that character as the input. This would happen in an infinite loop, no different than standard practice for modern real-time networking. A single character was not enough for useful information, so multiple characters would be sent in this structure: `<command> <arg1> <arg2>...<argn>`. Each character was sent separately, and the receivers would wait for all the arguments to arrive (if needed) before processing the command. With this in mind, it would theoretically be possible for characters from multiple clients to be interleaved, with bytes from one player being mixed in with those from another, causing unpredictable behaviour. It is unclear if the server had a role in mitigating this, but we do know that *Maze* had error handling for unexpected data. For example, player IDs are checked for validity (between `0o001` and `0o008` inclusive) and that they are not the receiving player's ID. There are also checks for turning in an invalid direction, or moving to a location outside the maze.

The following is a description of the six possible legal message types. In the description below, we separate each character by whitespace. No arguments are optional. [**?**]

`0o001 <id>`
Player `id` left the game.

`0o002 <id> <direction> <x> <y>`
Player `id` moved to position `(x,y)` and turned to `direction`.

`0o003 <id> <target id>`
Player `id` shot player `target id`.

`0o004 <id> <uname> <# of hits> <# of deaths>`
Player `<id>` with name `uname` joined the game.

`0o014`
Erase display ring buffer, clears message history.

`>0o014`
Print this character in the display ring buffer. For example, if `0o015` is received, a carriage return is printed.

These are mostly self-explanatory, but some further detail is needed for message types `0o002` and `0o004`. When a number is sent (e.g., `# of hits`), the `0o100` bit is set to distinguish this from an ASCII character. If characters are received when numbers are expected, the message is discarded by error handling.

A player's `uname` is always 6 characters long with spaces used to pad shorter names, and the count of hits and deaths are sent as two characters each, the first character carrying the high order 6 bits, and the second the lower order 6 bits. These would be combined to form the full 12-bit number. As a result of these, message type `0o004` would send 12 bytes total.

Message type `0o004` is sent when a new player joins to announce their presence and username. In response, all clients would reply with a message of the same type so the new player would know all other players' usernames and scores.

As mentioned earlier, the PDS-1 was common in the ARPANET, and according to **?**, half of the traffic between Stanford and MIT were allegedly from *Maze* games in progress. The considerable distance between these two institutions (~ 4000km) created considerable latency which affected gameplay, though this was fixed in a later version by sending data about the relative motion in type `0o002` messages.

Latency was an issue due to the client-side hit registration. For instance, player 1 knows it has shot player 2 based on the latest location it has of player 2. However, if player 2 moved out of the line of fire before receiving the full message informing them of their death, from their perspective it would appear as if they were hit by an enemy who shouldn't have been able to see them. This would give an unfair advantage to the shooter when latency was sufficiently high.

Other issues may have also included the aforementioned message interleaving, though many cases would likely have been handled by the error handling in the client code.

4.2 Sopwith

Released in 1984 by Canadian company BMB Compuscience as a demo for Imaginet – their proprietary networking system – *Sopwith* is a simple video game where up to four players fly their own Sopwith Camel aircraft from a side-on 2D perspective. A more modern version of the code written by David L. Clark is available

[2] *Spasim* (1973) was released in the same year for the PLATO system with support for up to 32 simultaneously players, though exact release dates are unclear.

and open source, but the original networking components are still present, which we will look at.

Sopwith itself does not implement much networking, and heavily relies on the Imaginet protocol, so we will discuss this first. Imaginet was a hardware adapter built by BMB Compuscience which could connect Atari STs, IBM PCs, and other popular computers of the time [**?**]. MS-DOS introduced native networking later in 1984 with DOS 3.1, but DOS 2.1 was still sold in alongside it [**?**], and IEEE 802.3 was still in its infancy, so Imaginet found a niche in the market. Imaginet was essentially a single floppy drive shared by all connected clients. The host would either have a physical drive or a virtual floppy drive image on a hard disk drive. Clients would have hardware installed in the place of a floppy disk controller, making Imaginet fully transparent to clients, appearing to the computer as a normal floppy drive. This meant it could be compatible with any device that supported floppy drives, regardless of operating system, and any software could become networked by addressing the shared disk. To prevent race conditions when multiple devices try to access the same file simultaneously, a microprocessor in the hardware controls access to the disk, marking it as `BUSY` while in use [**?**].

As for *Sopwith*, it makes use of Imaginet using a custom interface called Diskette I/O or DKIO. The implementation is very simple – if the game is in multiplayer mode, the multiplayer state is saved to and loaded from the virtual floppy drive. The state includes all relevant multiplayer information like player count, max players, state, physics, fuel, score, and physics information of each player. The state in the local memory and the state on the server is synced once per game loop, i.e., at frame-rate. An unusual implementation detail is that all clients write to the same specific virtual "disk sector" rather than using some kind of file system lookup, an interesting example of the transparency of Imaginet. This would prove problematic if the sector stored other data, overwriting or corrupting it, but since *Sopwith* was intended as a demo, this was likely not a concern.

In more modern times, Imaginet has been emulated by a project called Imaginot [**?**]. Imaginot works by intercepting and responding to low level DOS system calls, sending the inputs over a specified network protocol, allowing *Sopwith* to be played in multiplayer mode without needing the original Imaginet hardware. Although this is beyond the scope of this project, it is interesting since Imaginot is transparent to Imaginet, which itself is transparent to *Sopwith*, and so running *Sopwith* now requires two layers of networking technology which act as virtual devices, a good example of the challenges in early video game networking before TCP/IP became standard.

4.3 SGI Dogfight

After its standardization on the ARPANET in 1983, TCP/IP was first used in a networked video game by *SGI Dogfight*. Originally created as a demo of the graphics capabilities of Silicon Graphics Inc. (SGI) workstations as well as SGI's IRIS GL – their proprietary graphics API which later became the open standard OpenGL – *SGI Dogfight* is a 3D flight simulator which was included with SGI workstations [**?**]. At first it was limited to single-player gameplay, but multiplayer was added in the form of a direct serial link (similar to how *Maze* networking began), then over Xerox Network System (XNS) multicast over Ethernet. No interaction between players was possible, but other player-controlled planes were visible. In 1985, `dog` was created, which allowed players to engage in dogfights, and UDP networking was added in 1986. This is significant as it marks the first known time the TCP/IP stack was used in a video game. SGI made the source code available on request, and many developers reportedly used the code to teach themselves how to use UDP packets [**?**]. Source code dated to 1988 is available and open source [**?**]. Though some changes may have been made from the original, the networking code appears to be unchanged and exhibits issues noted by developers and players, so this appears to be a representative example of the original UDP code.

Like *Sopwith*, *SGI Dogfight* sent and received multiplayer data once per frame. Communication was achieved by using UDP broadcast packets through port 5130, sending them to every client on the network. If other clients were running *SGI Dogfight*, they would interpret the packets to update their local state. This implementation would prove problematic as we will discuss later.

SGI Dogfight was built for IRIX, SGI's UNIX-based operating system. As such, it uses the `<netinet/in.h>` POSIX header to interface with the kernel-level networking. On initialization, structures are created for each player-controlled plane which stored the same full set of information as the local plane. The broadcast socket and host's IP address are found and stored for subsequent use. Each plane is given an ID (the client's IP address), and lookups are performed by iterating through the plane structs until a plane with the matching IP address was found and returned. Depending on the number of players present, it might have been more efficient to create a mapping of numerical IDs to IP addresses and directly address the corresponding element in the array when looking up a specific plane, though this would trade off a small amount of memory.

Sending and receiving is implemented in a simple way – sending uses the `sendto` function directly, and receiving reads from the socket in a loop until no more data is available in the socket buffer. The packet structure sent is always a plane struct, regardless of message type, so the `cmd` field dictates this message type, represented by a macro corresponding to an integer. The following is a description of each of the 4 packet types. [**?**]

`DATA_PACKET 0`
Standard plane struct with all plane-related information.

`MSG_PACKET 1`
Contains chat message and addressee; if no addressee is specified the message displays for all players.

`SUPERKILL_PACKET 2`
Special admin message which exits the game for the addressed player or all players if none is specified.

`KILL_PACKET 23`
Exits game if game version of sender is greater than the receiver's.

Despite using the plane struct, chat messages are just written to the struct from a certain point onward, ignoring the normal structure. Regardless of whether an addressee is specified, the message is

still broadcast to all players, and the receiving code decides whether or not to display it.

SGI Dogfight has handling for mismatched versions as alluded to in the packet schema – if a message is received and the sender's game version is strictly less than the receiver's version, the receiver will respond with a `KILL_PACKET` addressed to that sender. This ensures that all players must have the same version as the player with the newest version and so achieves a form of eventual consistency of version.

`DATA_PACKET`s corresponding to the local plane struct are sent every frame to update the locally stored information for all other players. When receiving a `DATA_PACKET`, the receiver also checks if a certain field in the packet is the same as their ID in order to run the function to crash their plane.

This implementation has a few problems. Since source code was available, it would have been trivial to modify the program to be able to send a `DATA_PACKET` to another player which would crash their plane without needing to fire a shot. Malicious users could also send `SUPERKILL_PACKET`s to force others games to exit.

More notably, a game of *SGI Dogfight* with sufficient players acts as an unintentional Distributed Denial of Service (DDoS) attack on other devices attached to the network. This problem was exacerbated by *SGI Dogfight*'s inclusion in all new SGI workstations, making it very easy to access. One `DATA_PACKET` is a struct of size 112 bytes, sent once per frame, and workstations could run *SGI Dogfight* between 7 and 15 frames per second depending on hardware. This means one player broadcasts $784 - 1680$ bytes per second during gameplay. The total traffic on the network is of course multiplied by the number of players, and the volume would cause congestion in a network if enough players were in a session, blocking traffic for other devices and even freezing them outright [**?**]. The game even includes an error message which warns of this:

> WARNING some machines can not handle large numbers of udp broadcast packets. If you have machines from other vendors on your network, running dog on your network may bring them to a halt. VAXes are known to have this problem.

[**?**]

5 CONCLUSION

We have described three of the earliest networked video games, detailed how they worked, and discussed the historical context they existed in as well as issues that manifested in the implementations.

Maze is notable for being the very first networked video game, targeting the IMLAC PDS-1 and using a serial connection and a server to repeat messages from a client to other connected clients.

Sopwith is a demo for BMB Compuscience's proprietary Imaginet networking software – a transparent, operating system-agnostic shared drive which allowed any software to use its network by reading and writing to a virtual floppy drive.

SGI Dogfight is significant for being the first video game to use the then-nascent TCP/IP stack which is now the underlying architecture for the internet and networks which interface with it.

Networked video games gave rise to a new means for players to interact with each other, no longer confined to a single display, but free to compete or collaborate from anywhere in the world.

At the time of writing this review, we are closely following the timeline elaborated in the proposal, and we are on track to complete the full work within the given time. The final report will include at least two more games – *Habitat* (1986) and *Netrek* (1988) – examined in a similar fashion as above.