Sardines: A Networked Game

Sardines is a system of communication for 2-8 players. Each player controls a submarine in real time, viewing their surroundings through the lens of a retro sonar system. These submarines communicate in morse code, firing soundwaves with thicknesses representative of dots and dashes to one another across the map. While this system has ultimately been designed for integration into a larger, more complete game, *Sardines* presents itself as a sandbox, to best restrict attention to the networking techniques at play.

Architecture

Sardines' network uses a straightforward client-server architecture, with the server's 'master' game state well-positioned to minimise disputes (see 'Prediction'). Furthermore, the architecture uses local input processing - each client takes responsibility for processing their own player's actions and transmitting the resulting changes, rather than leaving the central server to compute a new master state directly from raw input. This distributes the game's physics-based movement calculations in a far more even fashion; an authoritative server with less trust in the player would better prevent cheating (Gambetta 2010), but 'cheating' will be of little concern while *Sardines* remains a sandbox game.

Bauer et al. (2004) evaluate the scalability of this architecture, finding that with n entity states (for the purposes of the report, players), client-server costs grow at order $O(n^2)$, compared to peer-to-peer's $O(n^3)$. They ultimately conclude that "The client-server architecture exhibits the lowest growth in overall system cost, however, with the disadvantage that the entire growth must be handled by the central server."

In light of their findings, this report proposes a hybrid architecture for *Sardines* to adopt at scale. While it ultimately proved too ambitious for this assignment, the original idea for the game was that *multiple* players work together in piloting each submarine: a co-operative exercise with a navigator relaying key information about the surroundings, and a small team of other crewmates individually controlling acceleration, steering, etc. With only navigators witnessing the global game state first-hand, Figure 1 positions these clients as 'local' servers, processing (up to, say) 3 players' inputs simulataneously, and relaying major changes to the game state back down to this crew. Not only does this topology require less of the central server's bandwidth (it still maintains 2-8 connections, while each navigator has up to 4 and other crewmates exactly 1), but demands far less calculation than if all 32 players send their highly-individualised updates directly to the master state.

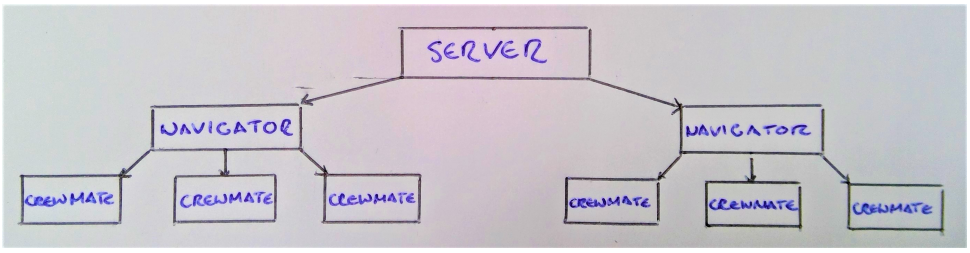

Figure 1: Potential hybrid architecture for a scaled version of *Sardines*.

Protocols

Transport Layer In the application, client and server communicate through TCP connections, chosen for their robustness. As described in RFC 798 (1981), where Postel sets out the transport protocol,

> Very few assumptions are made as to the reliability of the communication protocols below the TCP layer. TCP assumes it can obtain a simple, potentially unreliable datagram service from the lower level protocols. In principle, the TCP should be able to operate above a wide spectrum of communication systems.

That the protocol guarantees reliability in-and-of-itself makes it perfect for *Sardines*' purposes: for any two players to meaningfully communicate in morse code, they must be able to trust that their dots and dashes reach one another in order and without error.

There is, of course, an argument for updating positions via UDP. Suppose a submarine starts at point A, and fails to update to point B - whereas a TCP connection will resend and resend this update until it succeeds, its ordered nature potentially holding up more important or immediate data, UDP assumes it *has* sent correctly and moves on to the next point C. The minimalism of the protocol lends itself to the continuous, incremental nature of movement, but *Sardines* will not make use of it. Submarines in this game travel slowly, with trajectories needed reliably but not immediately, and given a robust prediction system (see 'Prediction') updating position via TCP every 0.2s should be infrequent enough to avoid the backlog described above.

Application Layer While it needn't consider the internetwork and hardware layers of the protocol stack, the application layer of *Sardines*' network necessarily interacts with the transport layer directly below it. Data is sent in a stream of `SendablePacket`s, consisting of a fixed-length `HeaderPacket` and one of several structs encoded as a `serialisedBody`. The header, always read first, contains a `bodyID` that tells the receiver how many bytes of the body need deserialised; as discussed in the CMP501 labs, this application handles variable-length messages in much the same way as the DNS protocol.

At a more granular level, a `serialisedBody` may encode one of many structs, for one of many purposes:

- `SyncPacket` Contains a `long syncTimestamp`. Sent on connection to estimate the travel time, `delay`, between client and server.
- `IDPacket` Contains an `int clientID`, and a `char[] clientIP` that requires, by definition, no more than 16 characters. Sent to initialise a client's unique identity, and let players track who else is playing.[1]
- `SubmarinePacket` Contains an `int submarineID`, the controller's `clientID`, and additional variables for use in as-yet-unimplemented features. Similarly initialises a unique identity for each submarine.[2]
- `PositionPacket` Contains a submarine's `x`-, `y`-coordinates and rotation `theta` at a certain time `timestamp`. Sent client-to-server as updates to the master game state, and server-to-client for prediction and rendering.
- `MorsePacket` Similarly contains the necessary parameters to render a soundwave on the receiver's screen.
- `EmptyPacket` Contains no variables. Sent when the `bodyID` corresponds to a function with no arguments (e.g. `bodyID 2310` tells a server in lobby mode to initialise a game).

`bodyID`s adopt an informal naming convention: IDs `1XXX` refer to clients connecting to/disconnecting from a server, `2XXX` to server functionality while in lobby mode, `3XXX` to server functionality while in a game mode, and `4XXX` to client states and actions while in-game.

While there isn't the space to break down every protocol in precise detail, all have been designed with the same underlying philosophy. Consider, for instance, the process of joining a lobby:

1. *The client registers a TCP connection with the server, and sends a* `syncPacket` *(ID 1000).*
2. *The server receives a* `SyncPacket` *from the client, and returns it, via* `Receive1000()`*.*
3. *The client receives a* `SyncPacket` *from the server, estimates travel time between the two devices, and sends an* `IDPacket` *(ID 1001), via* the client version of `Receive1000()`*.*
4. *The server receives an* `IDPacket` *from the client, changes the* `clientID` *if it is unrecognised,*[3] *and returns it, via* `Receive1001()`*.*
5. *The client receives an* `IDPacket` *from the server, a formal acceptance into the lobby, via* `Receive1001()`*.*
6. *All clients receive (further)* `IDPacket`*s from the server (IDs 1002). These tell players in the lobby who has just joined, and vice versa, all via* `Receive1002()`*.*

The process is designed for ease of programming: treating major protocols as a chain of smaller, simpler steps, it becomes far easier to manage - and document - the application layer (even if there is some unnecessary back-and-forth to the above).

API

Sardines is written in C#, for the Godot engine. It uses System.Net.Sockets to handle networking, and System.Runtime.InteropServices to serialize/deserialize packet structs. As noted in the documentation (n.d.), System.Net.Sockets implements conventional Berkeley sockets.

[1] In the hybrid architecture discussed above, the IP would also be used to establish crew-to-navigator connections.
[2] And similarly, the distinction between `clientID` and `submarineID` is only meaningful with multiple clients per submarine.
[3] Or banned! In this case, the server returns an `IDPacket` with a strictly negative `clientID`, which cues the client to disconnect.

Integration

Each client organises its networking components in a `Handler` class, along with a 'local' copy of the game state. This is treated as a black box by the game at large, with local processing entered in and external updates read back out. Consider, for example, how the `NavigationDisplay` interacts with the `Handler`. Every frame, `UpdatePosition()` pushes the player's new position into the local state and sends it on to the server; then, `Render()` pulls that same data in displaying (a relative view of) their surroundings onscreen.

Client and server communicate 'as and when'. Since TCP connections risk blocking either component, both rely on I/O multiplexing. The server uses System.Net.Sockets' `select()` function to monitor all connections simultaneously, set to time out after $1ms$. Each client does the same with its single server connection, to avoid freezing the broader game - `select()` will still block, though, sometimes resulting in an unfortunate buffering effect under poorer network conditions.[4]

Prediction

As discussed in 'Architecture', clients only send position updates every $0.2s$. What this report has so far failed to consider is how this appears to other clients - they experience what should be a smooth, continuous movement as discontinuous jumps over $0.2s$ intervals! *Sardines* approaches this problem with a variety of techniques.

When players choose to move forward, they do not jump to a constant speed but slowly ramp up from zero; the game makes a second-order, quadratic prediction to best approximate the second-order derivative that is acceleration. Given a submarine's three most recent positions $\mathbf{r}_0$, $\mathbf{r}_1$, $\mathbf{r}_2$ (corresponding to times $t_0 > t_1 > t_2$), clients can average the velocities from $\mathbf{r}_1$ to $\mathbf{r}_0$, from $\mathbf{r}_2$ to $\mathbf{r}_1$, and the acceleration from $\mathbf{r}_1$ to $\mathbf{r}_0$ as

$$\mathbf{u}_0 = \frac{\mathbf{r}_0 - \mathbf{r}_1}{t_0 - t_1}, \quad \mathbf{u}_1 = \frac{\mathbf{r}_1 - \mathbf{r}_2}{t_1 - t_2}, \quad \mathbf{a}_0 = \frac{\mathbf{u}_0 - \mathbf{u}_1}{t_0 - t_1}, \quad \text{respectively.}$$

These parameters define the quadratic model

$$\tilde{\mathbf{r}}(t) = \mathbf{r}_0 + \mathbf{u}_0 t + \mathbf{a}_0 t^2.$$

In contrast, the rudder controlling a submarine's rotation θ *is* controlled at a constant speed, so *Sardines* only uses linear prediction to approximate

$$\tilde{\theta}(t) = \theta_0 + \dot{\theta}_0 t$$

(taking extra care to handle jumps from $\theta \approx 0$ to $\theta \approx 2\pi$, and vice versa).

If prediction is how one waits for data, then integration is what one does on its arrival. Receiving a new `PositionPacket` at time t_0, a programmer may wish to switch to new quadratic model $\tilde{\mathbf{r}}_{\text{new}}(t)$ immediately, but if positions $\tilde{\mathbf{r}}_{\text{old}}(t_0)$ and $\tilde{\mathbf{r}}_{\text{new}}(t_0)$ are visibly far apart then the player will see the corresponding submarine make an instantaneous jump across the screen.[5] Instead, one takes a set time T to linearly interpolate from the old trajectory to the new:

$$\tilde{\mathbf{r}}(t) = \begin{cases} \tilde{\mathbf{r}}_{\text{old}}(t) & \text{if } t < t_0 \\ (1 - q(t))\tilde{\mathbf{r}}_{\text{old}}(t) + q(t)\tilde{\mathbf{r}}_{\text{new}}(t) & \text{if } t_0 \leq t < t_0 + T \\ \tilde{\mathbf{r}}_{\text{new}}(t) & \text{if } t \geq t_0 + T \end{cases}, \quad \text{where } q(t) = \frac{1}{T}(t - t_0).$$

In *Sardines*' particular implementation, `PositionPacket`s are sent via TCP every $0.2s$; interpolation therefore takes place over a strictly shorter interval $T = 0.1s$.

A classic problem in networked game design is one of conflict resolution - if client A hits client B with a projectile in their own local state, but misses in B's, whose account does the server take as true? In *Sardines*, the projectiles concerned are soundwaves. The visual language of the game, where soundwaves from external sources only become visible on collision with the player, provides a clear approach: the sender unequivocally takes precedence. Only when a player sees their soundwave hit another is a `MorsePacket` sent from their

[4] In hindsight, the function should have been set to time out immediately, at `0`.

[5] This might be regarded interpolation over $T = 0.0s$!

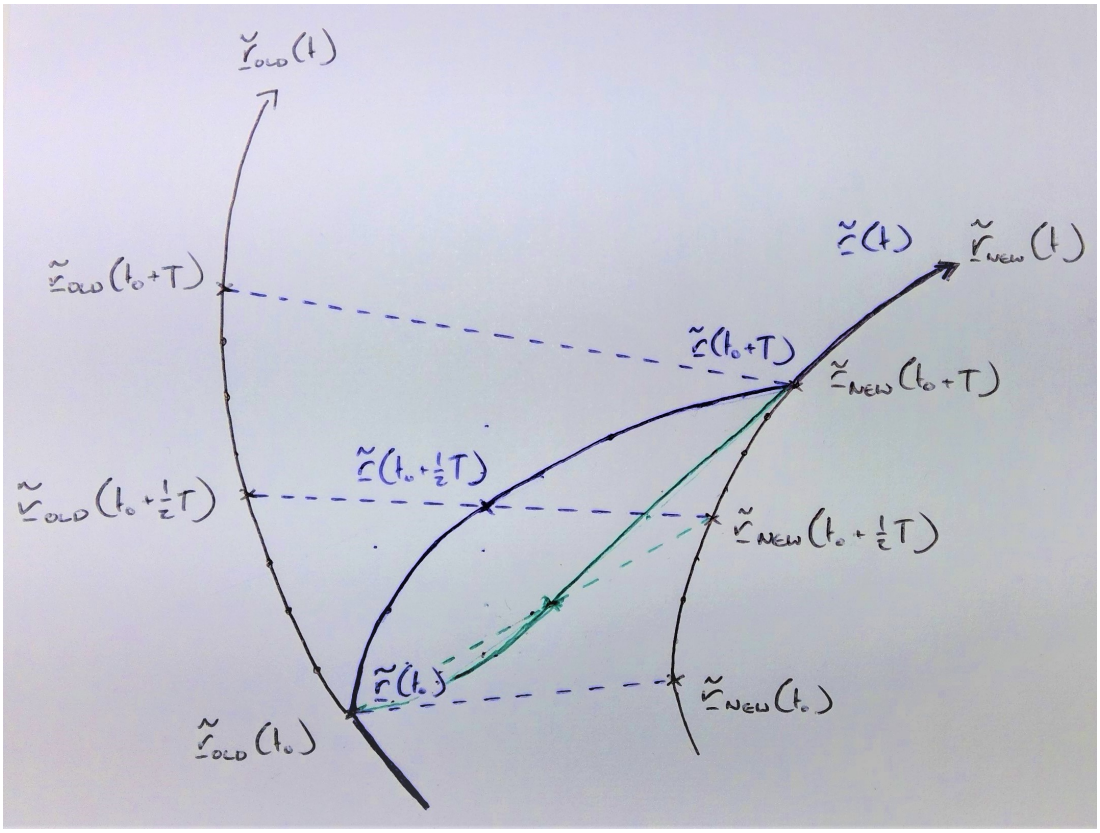

Figure 2: Interpolation from trajectory $\tilde{\mathbf{r}}_{\text{old}}(t)$ to $\tilde{\mathbf{r}}_{\text{new}}(t)$ (blue); chosen over $\tilde{\mathbf{r}}_{\text{old}}(t_0)$ to $\tilde{\mathbf{r}}_{\text{new}}(t)$ (green).

client (which will arrive with the usual delay). The sender knows with certainty who receives their message; the receiver, who cannot see the trajectory of the soundwave until it arrives, will have no sense whether it "should" have hit them according to their local game state.

To further smooth over the application's disputes, the receiving client makes use of backward prediction. Since neither server nor client stores more than three of any submarine's past positions at a time, it is fortunate the above definition of $\tilde{\mathbf{r}}(t)$ can approximate the past as well as the future.[6]

Suppose a sender emits a soundwave from position $\mathbf{r}$ at time t_0, which they see reach a receiver at $t_0 + \Delta t$. On the arrival of the corresponding packet at t_1, then, the receiving client has to decide where the wave was emitted from *in its local view of the game*. The obvious choice would be the 'true origin' $\mathbf{r}$, but *Sardines* uses the backwards prediction $\tilde{\mathbf{r}}(t_1 - \Delta t)$. Conflict resolution is, fundamentally, the art of deciding which quantities are preserved across clients, and *Sardines* - a system exploring slow, real-time communications - is far less concerned with presenting a shared view of geography than it is a shared view of delay.

Testing

The following insights were made with the help of clumsy.[7] This tool can simulate many a real-world network condition, but in the interests of brevity the section will restrict its attention to two most relevant to the TCP-based game at hand: latency (where all data are delayed by a set interval) and throttle (where data over a set interval are held back, then sent all at once).

[6] *Sardines*' submarines are physics-based objects, and at one point in development, the drag they experience was factored into prediction. However, the differential equations for 2D motion with a quadratic drag were too complex to find an analytic solution - rather than being able to substitute a t-value into a given equation, the prediction would be calculated over incremental, irreversible forward time steps - so the application sacrifices this more realistic model for the ability to look backwards in time.

[7] Note that - with testing so integrated into the larger development process - the clients tested were often run within the Godot editor. With poorer performance than a standalone executable, the following results may constitute more of a worst-case scenario for *Sardines* than intended...

Latency While opinions will vary on what consistutes a 'reasonable' amount of lag, *Sardines* shows promise. The predictions made here provide a smooth enough experience at a latency of $200ms$, even $250ms$, with the game only becoming unplayable at higher thresholds. A latency of, say, $500ms$, will see the clients' worldviews slowly desynchronise, accumulating more and more error as time goes on. With no immediate solution in sight, one might be inclined to take a step back, and interrogate the network's most basic assumptions; it could well be that the theory behind, say, eschewing UDP connections has not held up in practice.

At any rate, there are more concrete fixes that can be made before worrying about such an open-ended performance issue - consider, for instance, how travel times are calculated between client and server. Currently, `delay` is averaged from 5 `syncPacket`s sent back and forth immediately on joining a lobby, which comes with a slow loading time but ultimately results in more accurate predictions. What this theory misses is `delay`s may vary over larger time scales; the experience of testing latency and seeing how travel times may spike suggests these values should be recalculated at least semi-regularly.

Figure 3: Three simultaneous, local views of the game world, at $100ms$ latency (taken during development). These would appear consistent, in spite of the delay.

Throttle On one hand, *Sardines* would appear to have a built-in threshold for throttle. Of all of variables involved in the application layer, clients only update position values continuously, and only when those position values are actively changing - and only *then* once every $0.2s$. It is therefore unsurprising that using clumsy to simulate $< 200ms$ throttle (with probability 10%) has little visible effect. However, with values just above this again giving way to desynchronisation,[8] the attached video demostrates performance at $300ms$ latency, with a 10% chance of $200ms$ throttle. This is a set of poor network conditions chosen to best challenge the application, without breaking it altogether.

Recall the interpolation technique described above. Since $T < 0.1s$, the report has assumed the submarine being interpolated starts on a predicted trajectory at time t_0 - does this assumption hold in practice? If two `PositionPacket`s have been throttled, and arrive within T seconds of each other, it follows that the submarine will not have finished its first interpolation by the time the second one begins. In this edge case of being caught mid-interpolation at some point $\mathbf{r}_0$, *Sardines* prioritises catching up; rather than using a predicted trajectory as in Figure 2, the subsequent interpolation simplifies calculations by taking $\tilde{\mathbf{r}}_{\text{old}}(t) = \mathbf{r}_0$.

Other issues remain unresolved in the submitted application, one being a vulnerability in the clients' morse code. While it's true that a TCP connection will preserve the order of dots and dashes, it will not preserve the intervals at which they were sent - if two or more throttled soundwaves end up showing at the same time, their order becomes indistinguishable! These could easily be staggered by a simple 'queuing system' on the receiver's end.

For all the optimisations and oversights discussed above, it's worth noting how fundamental testing has already been to development. With network programming being notoriously unpredictable, *Sardines* has necessarily involved a lot of quality assurance: a feature that works offline must be tested between a single client and the server; then between two clients on the same device; then across devices, or between more clients, or under poor network conditions; the list goes on. This section takes a largely qualitative approach in order to reflect the role of clumsy in this process, a means to poke and prod at the underlying network. *Sardines* is not, in the end, a perfect application - the goal of this report is not to obscure its flaws, but to better understand them.

[8] Similarly - the application can't even handle a 5% packet loss...

References

Bauer, D., Iliadis, I., Rooney, S. & Scotton, P. (2004), 'Communication architectures for massive multi-player games', *Multimedia Tools and Applications* **23**(1), 47–66.

Gambetta, G. (2010), 'Fast-Paced Multiplayer (Part I): Client-Server Game Architecture', Available at: `https://www.gabrielgambetta.com/client-server-game-architecture.html`. (Accessed: 14 January 2023).

Microsoft (n.d.), 'Learn / .NET / .NET API browser / System.Net.Sockets Namespace', Available at: `https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets?view=net-7.0`. (Accessed: 14 January 2023).

Postel, J. (1981), Transmission control protocol, RFC 793, Information Sciences Institute, University of Southern California, Marina del Rey. Available at: `http://www.rfc-editor.org/rfc/rfc793.txt` (Accessed: 14 January 2023).