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 - 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.

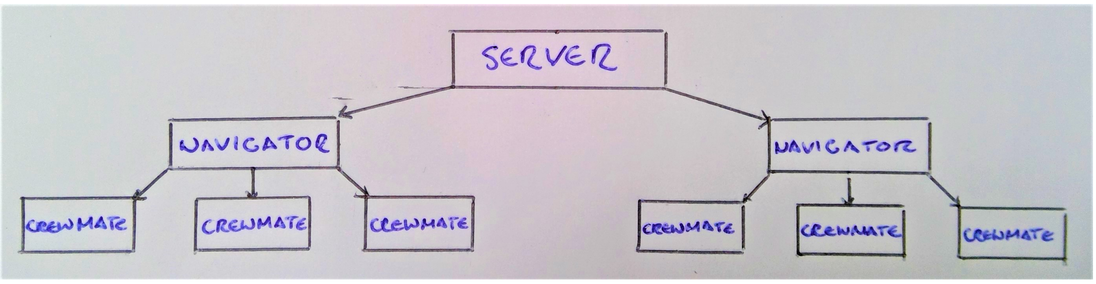

Figure 1: Potential hybrid architecture for a 32-player version of *Sardines* (simplified).

Protocols

Transport Layer In *Sardines*, client and server communicate through TCP connections, chosen for their reliability. As much is set out in RFC 798 (1981), where Postel introduces the protocol:.... Provide reliability paragraph?...

There is, of course, an argument for updating positions via UDP. Suppose that a submarine starts at position A, fails to update to position B: where the ordered nature of TCP means the application will resend and resend this update [EXPLAIN DOWNSIDE], this protocol will simply [GO TO the position C of the next successful]. While the deliberate simplicity of UDP - the [FEATURE #1], the [FEATURE #2] - lends itself to the continuous, incremental nature of movement , *Sardines* will not make use of it. Submarines in this game travel slowly, with positions needed reliably but not immediately, and with the robustness of the prediction system used (see 'Prediction'), updating position via TCP every 0.1s 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 via `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 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 standardise server and client calculations for `DateTime.UtcNow.Ticks` (which may vary from underlying OS to underlying OS, Microsoft n.d.*a*).
- `IDPacket` Contains an `int clientID`, and a `char[] clientIP` that requires, by definition, no more than 16 characters. Sent to initialise a unique identity for each client, and more generally let players track who else is playing.[1]
- `SubmarinePacket` Contains an `int submarineID`, the controller's `clientID`, and additional variables corresponding to future functionalities. 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's arc on the receiving client'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.* The client then constructs... with ID `1000`...
2. *The server receives a `SyncPacket` from the client.* Calling `Receive1000()`, the server...
3. *The client receives a `SyncPacket` from the server.*
4. *The server receives an `IDPacket` from the client.*
5. *The client receives an `IDPacket` from the server.*
6. *All clients receive (further) `IDPacket`s from the server.*

There is arguably some unnecessary back-and-forth to the above, but small packet sizes shouldn't put any meaningful strain on bandwidth. 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.

API

Sardines is built with C# in the Godot engine. It uses System.Net.Sockets to handle networking, and System.Runtime.InteropServices to serialize/deserialize packet structs. As noted in Microsoft's documentation (n.d.*b*), 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, of course, with multiple clients per submarine.

Integration

Asynchronous I/O... Connection class...

Discuss: offline vs. online updates to position!

This report would be amiss to skip over the final step:

Prediction

As discussed under 'Architecture', clients only send position updates every $0.1s$. 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 $0.1s$ intervals! Clearly, ... [introduce prediction - with reading?].

When a player chooses to move forward, they do not jump to a constant speed but continuously accelerate from zero; naturally, *Sardines* uses second-order quadratic prediction to best approximate the second-order derivative of accelaration. 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 estimates 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

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

(with subtle, case-specific considerations made given $\theta \in [0, 2\pi)$).

If prediction is the act of waiting for data, then integration is how one 'catches up' on receiving it. On receiving a new `PositionPacket` at time t_0, a programmer might be inclined to start predicting under to a 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.[3] Instead, one takes a set time T to linearly interpolate from the old trajectory to the new:

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

In *Sardines*' particular implementation, `PositionPacket`s are sent via TCP every $0.1s$; interpolation therefore takes place over a strictly shorter interval $T = 0.05s$.
To fully understand how *Sardines* uses it prediction techniques, this report must first introduce a core challenge of any networked game: conflict resolution.

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 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 of whether it "should" have hit them.

To further 'smooth over' the application's conflict resolution, 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 formulae can approximate the past as well as the future.[4]

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

[4] *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.

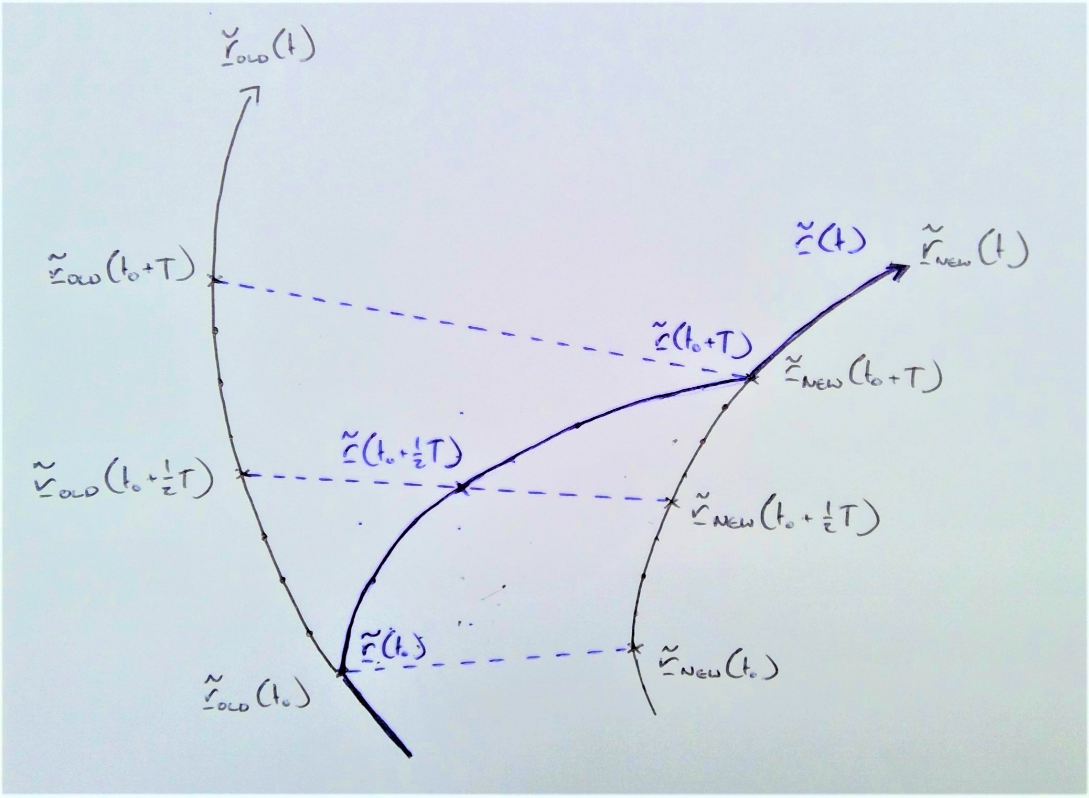

Figure 2: Interpolation from trajectory $\tilde{\mathbf{r}}_{\text{old}}(t)$ to $\tilde{\mathbf{r}}_{\text{new}}(t)$ (blue); chosen over $\tilde{\mathbf{r}}_{\text{old}}(t_0)$ to $\tilde{\mathbf{r}}_{\text{new}}(t)$ (red).

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 $\tilde{\mathbf{r}}(t_1 - \Delta t)$. As [FIGURE] puts it in [REFERENCE], [QUOTE]; conflict resolution is the art of deciding which quantities are preserved across clients, and *Sardines* - a system designed around slow, real-time communications - is far less concerned with a shared view of geography than it is a shared view of delay.

Testing

[PARAGRAPH OF METHODOLOGY] Using clumsy... Since *Sardines* only uses TCP connections, it ; this report will restrict its analysis to performance under latency and throttle

Latency Latency (or, colloquially, lag) is... For a game being released at scale, it is recommended to test latencies from $100ms$ to $1000ms$, as a bare minimum (Unity n.d.).

Figure 3: Three simultaneous, local views of the game world, at [ACCEPTABLE LAG]ms lag (above); [UNACCEPTABLE LAG]ms lag (below).

[Paragraph of Evaluation] The choice of a TCP as the application's sole transport protocol is seems to be of [little/significant] disadvantage here. [Central idea: design decisions that seem optimal in theory, but need tested in practice!]

Throttle [EXPLANATION OF THROTTLE AND TESTING]

Testing lag and throttle simultaneously yields unsurprising results. Having established acceptable network conditions - [TIME]ms lag, a [% chance of [TIME]ms throttle -

Recall the interpolation technique described above. Since $T < 0.1s$, the report has assumed the submarine being interpolated appears at position starts on a predicted trajectory at time t_0 - does this assumption hold in practice? If two `PositionPackets` 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 starts! 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 $\tilde{\mathbf{r}}_{\text{old}}(t) = \mathbf{r}_0$.

Indeed, for all the optimisations and oversights discussed in this section, its worth noting how fundamental testing has already been in development. With network programming being notoriously unpredictable, *Sardines* has necessarily involved a lot of QA: 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. While [Conclude here, or one final paragraph for more general reflection?].

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.*a*), 'Learn / .NET / .NET API browser / System / DateTime / Properties / DateTime.UtcNow Property', Available at: `https://learn.microsoft.com/en-us/dotnet/api/system.datetime.utcnow?redirectedfrom=MSDN&view=net-7.0#System_DateTime_UtcNow`. (Accessed: 14 January 2023).

Microsoft (n.d.*b*), '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).

Unity (n.d.), 'Testing Multiplayer Games with Artificial Network Conditions', Available at: https://docs-multiplayer.unity3d.com/netcode/current/tutorials/testing/testing_with_artificial_conditions/index.html. (Accessed: 15 January 2023).