COMP90020 Project Report

Jessica Pollard (1269647)
Matthew Pham (1273607)

May 2025

Abstract: This paper discusses an augmented implementation of Mattern's parallel snapshot algorithm to save the game state in an online multiplayer game. This approach allows for the application of Mattern's algorithm in a non-FIFO UDP network whilst accounting for potential packet-loss.

Keywords: distributed algorithms, snapshot algorithm, Mattern's algorithm, Chandy-Lamport algorithm, Lai-Yang algorithm, online multiplayer game

---

Contents

1 Introduction

This report discusses the analysis and implementation of a snapshot algorithm within a distributed system, specifically an online multiplayer game. A background survey and critical analysis of the Chandy-Lamport, Lai-Yang and Mattern's parallel snapshot algorithms has been completed and the implementation of an augmented Mattern's algorithm within the game is discussed.

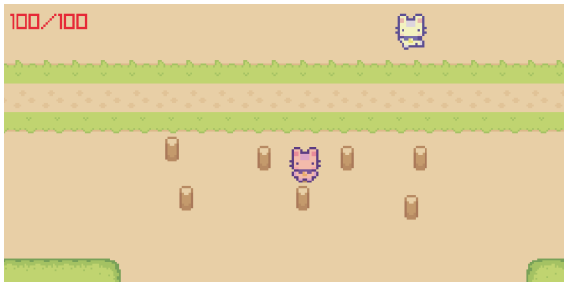
2 Background and Distributed Application

The implemented distributed system is an online multiplayer game. The game allows for two to four players per lobby, with one of the players acting as the host. Each player owns a particular quadrant of the game map and is responsible for any entities placed in that quadrant. In the context of this game, this means each player is responsible for fences placed in their section. Figure 1 demonstrates the ownership of quadrants. 1a and 1b show the perspectives of player 1 and player 3 at the same point in time where both have placed fences in their own quadrants. Figure 1c shows the perspective of player 3 once it moves to player 1's quadrant and it is now able to see all fences placed in that quadrant.

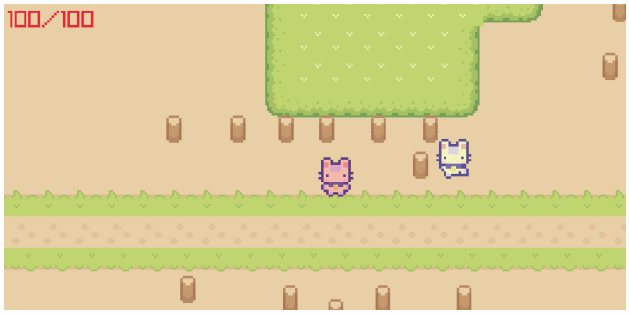
The clients within this distributed system communicate over a UDP connection and send messages for the following changes in state:

1. Player movement - message sent once per game tick
2. Player initiation and death - message sent once at the beginning of the game or when the player dies
3. Fence attacked - message sent to fence owner when fence attacked
4. Player attacked - message sent to all players when a player is attacked
5. Request fences - message sent once when a player enters another players quadrant
6. Placed fence - message sent to the quadrant owner when a fence is placed in their quadrant

(a) Perspective of player 1 with fences placed in their region



(b) Perspective of player 3 with fences placed in their region



(c) Perspective of player 3 after entering player 1's region

Figure 1: Player perspectives and fence placements

The host player is able to save the game at any point in time and restart the lobby from that save file. Due to the use of UDP for connections, careful selection of a snapshot algorithm is required in order to generate a globally consistent save state that can be serialised whilst accounting for dropped packets.

3 Survey of Related Algorithms

3.1 Chandy-Lamport

3.1.1 Overview

The Chandy-Lamport algorithm is designed to capture snapshots in a distributed system with FIFO communication channels. The algorithm defines the following key features:

- **Marker Messages:** The algorithm is based on the concept of marker messages, which are messages sent by a process in order to segregate the messages between pre- and post-snapshot states. A marker is sent along all outgoing communication channels once a process has recorded it's snapshot, but prior to sending any other messages on any channel. All processes must record their snapshots at the point in time when it receives the first marker message, no matter what channel this marker is on.

- **Marker Sending Rule:** This algorithm allows any of the i processes to initiate a snapshot. A snapshot is initiated by a process recording it's local state and then sending a marker message on all outgoing channels. This rule is described in pseudocode in figure 2.

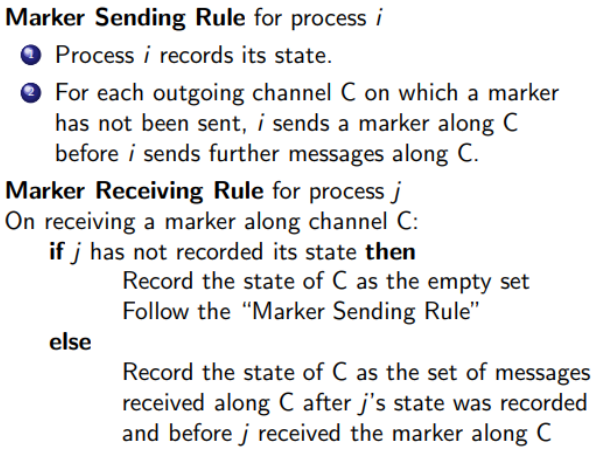


Figure 2: Chandy-Lamport Pseudocode [1]

- **Marker Receiving Rule:** When a marker message is received on any incoming channel, if the process has not yet taken a snapshot of its local state, it records the state of the channel the marker is received on and executes the marker sending rule. This rule is described in pseudocode in figure 2.
- **Termination:** The algorithm terminates on a given process once it has received a marker message on all incoming channels.

Once the algorithm has terminated, the snapshot of each process is sent to all other processes in order to determine the current global state.

The Chandy-Lamport algorithm is most commonly used in distributed systems with a clear and logical definition of message ordering. For example, the algorithm can be used in distributed databases and distributed file systems in order to checkpoint or determine a stable property, such as deadlocks and process termination [2].

3.1.2 Critical Analysis

The main requirement of the Chandy-Lamport algorithm is FIFO communication channels. This is integral to ensure the correctness and consistency of the resulting global state. If a non-FIFO channel is used, there is no way to ensure that the resulting snapshot is consistent. Our application is built on UDP connections which inherently are non-FIFO. This means that this algorithm is not applicable as there would not be a way to construct consistent snapshots of the games current state.

3.2 Lai-Yang

3.2.1 Overview

The Lai-Yang algorithm is designed to capture snapshots in a distributed system with non-FIFO communication channels. Lai-Yang builds on the concept of marker messages from Chandy-Lamport by introducing the concept of coloured messages.

- **Message Colouring:** In the initial state of all processes their messages are white. Once a process has received a red message on any channel, it becomes red and executes the marker sending rule as defined in Chandy-Lamport [3].

- **Recording Channels:** Every white process maintains a record of all white messages sent or received across each channel. These records form part of the local snapshot at the point in time when the process turns red [3].
- **Collecting Partial Snapshots:** Once a process has taken a local snapshot of its current state, it sends this, along with all recorded white messages, to the initiator process. The initiator process is then able to determine which messages were in transit for each channel, and based on this it can determine whether a particular white message belongs to the snapshot of the receiver or the channel of the sender [3].

The colouring of messages allows processes to determine the implied ordering of messages as "every message sent by a white (red) process is coloured white (red). Thus, a white (red) message is a message that was sent before (after) the sender of that message recorded its local snapshot." [3]. The usage of colours here means that an explicit marker message is no longer required, and instead a process can determine when to take a snapshot based on the colour of messages it is receiving.

Lai-Yang's algorithm is often used in distributed systems that do not have clear synchronization. Since the colour based markers reduce the interference in general message sending and the message history can be reduced to only storing the messages sent between snapshots, it is often used in distributed systems that require frequent snapshots [3].

3.2.2 Critical Analysis

Whilst Lai-Yang allows for correct and consistent snapshots to be taken in distributed systems with non-FIFO communication channels, it requires a history of all messages across all channels between snapshots to be recorded. Our application sends a significant number of messages each game tick. These include messages for player movement, placing fences and also attacking any players or fences. The sheer volume of messages per channel means that storing a history between snapshots becomes impractical and memory-consuming since snapshots are assumed to not be frequently taken in the game.

3.3 Mattern's Parallel Snapshot

3.3.1 Overview

Mattern's parallel snapshot algorithm adapts the Chandy-Lamport and Lai-Yang algorithms for systems with non-FIFO communication channels without the need to explicitly record complete message histories. It defines the following steps:

- **Pre-snapshot:** Every process begins with messages of an agreed upon colour, white. Before the recording of a snapshot, for all connected channels, each process records the number of white messages sent on every channel, as well as the number of white messages received on every channel.
- **Initiation:** A single process acts as the initiator and can commence the process as though it has received a marker message from a non-existent channel. In this case, the process turns red, records its local snapshot, begins recording messages on all of its channels and sends a marker message to all other channels [5].
- **Receipt of markers:** Upon receiving either the first marker message or the first red message on any channel, a process takes a snapshot of its local state, turns red, marks

that channel as having no more messages to record, and sends a marker message to all other channels. The process also begins recording messages on all other channels. Any markers or red messages received on any other channel will result in the state of that channel being recorded [5].

- **Piggybacking:** Instead of recording message histories like in Lai-Yang's algorithm, a red process will piggyback the number of white messages sent to a particular channel since the previous snapshot on every red message sent through that channel. This includes piggybacking on the original marker message.
- **Termination:** Once a process has received a marker message or red message on a particular channel, it will stop recording white messages on that channel when the number of white messages received by the process matches the number of white messages sent as indicated by the marker message or red message. When every channel has received a marker and fulfilled this condition, the process has successfully recorded its local snapshot. This local snapshot can then be sent to the initiating process [5].
- **Assembly of local snapshots:** Once the initiating process has received snapshots from all processes, it can assemble them to create a final snapshot of the system's global state [5].

This algorithm does not require message histories to be permanently saved, and also allows for snapshots to be taken in parallel with continued processing.

3.3.2 Critical Analysis

Mattern's algorithm has been determined to be the most suitable algorithm for the described multiplayer game application.

Since the communication channels in this game use UDP connections, there is no guarantee that packets will arrive reliably or ordered. Mattern's algorithm accounts for non-FIFO channels specifically through the piggybacking of message counts and recording of message states.

Furthermore, the volume of messages sent between clients in this game is high. This is mainly due to the movement messages sent up to 60 times per second between all clients. Due to the volume of messages, saving a message history for each communication channel is infeasible as local memory and processing memory would likely reach their maximum capacities. Since Mattern's algorithm does not require message histories to be saved, this problem is avoided.

4 Selected Algorithm - Mattern's Parallel Snapshot

Whilst Mattern's algorithm appears suitable for a multiplayer peer-to-peer network, packet loss from UDP networking channels creates a significant issue with the termination condition. The termination condition fails to cover the following scenarios:

- If it's assumed that packet loss can occur for any message, then it's likely that a message meant for a white snapshot might be dropped from the network.

- If it's assumed that messages are never resent, then the number of white messages received will never be equivalent to the number of white messages sent. Therefore, the recording of white messages cannot be guaranteed to terminate.

- If packet resending is allowed, which is common in a UDP networking protocol, then given packet loss can occur for any message, every white message sent before initiation of a snapshot must be stored by each process respectively in case one of these messages has to be resent. Having to record full message histories defeats the point of using Mattern's Algorithm over Lai-Yang's Algorithm, and therefore rendering a naive implementation of Mattern's Algorithm redundant.

Whilst the other steps of Mattern's algorithm function correctly in a packet-loss environment, the original termination condition is rendered obsolete. However, the idea that information can be piggybacked onto a message to indicate when the recording of messages should terminate is useful. If a termination condition exists that compensates for packet loss but is still able to provide an accurate snapshot of the global state at any given time, Mattern's Algorithm can be adapted to perform a global snapshot with this new termination condition instead.

In a multiplayer networking context, there indeed exists a satisfactory condition, but it is first important to analyse the fundamental ideas that allow multiplayer UDP networking protocols to function.

4.1 UDP Networking in a Multiplayer Context

The UDP netcode in this project is derived from the series of articles from Glen Fielder [4]. Instead of going through every detail of this particular networking protocol implementation, it's more useful to understand a few key concepts.

When implementing a reliable networking protocol over UDP, there is a single idea that usually differentiates a UDP based networking protocol over TCP.

- Fast-paced games usually only care about the most recent data they have received from another client. For example, if a process receives two different packets P_1 and P_2 containing a player's position and $P_1 \rightarrow P_2$, but P_2 arrives before P_1, it's preferable to ignore P_1 as P_2 provides the most up-to-date information on the current game state relative to the process receiving both messages. This type of data is known as time-critical data and occurs in a variety of contexts outside of gaming (VoIP, video streaming, etc.).

However, it's not desirable to simply accept every UDP packet and process them immediately upon arrival.

- In the real world, packet jitter and packet loss may make packet delivery uneven and unordered. For example, if a client were to process the player's position on screen based on the immediate arrival of packets, the player's position would jump back and forth as if the player is lagging, and the distance between the player's steps would be uneven.

To combat both of these issues, UDP game networking protocols process messages according to a set uniform tick rate (i.e. 30 - 60 ticks per second). Before a tick is processed, every

message received is put into a message buffer. Packets containing messages are prefixed with a header containing a sequence number for that message, indicating a total order in which messages should be processed. Upon the completion of a tick, a client processes all the unprocessed messages that currently exist, updating its local state.

The most important things to understand in the context of reordered and lost packets is the following:

- If a packet arrives out of order, i.e. the sequence number of the packet is *less than* the acknowledged sequence number of the most recently received packet, but arrives *before* the message buffer is processed, then the packet *will be reinserted* into the unprocessed message buffer.
- If a packet arrives out of order, but *after* the message buffer is processed, then the packet *will be ignored*.

Packets that are lost are equivalent to packets that did arrive to the process but were not processed by the client. It is this insight that allowed us to formulate a correct termination condition for an adjusted Mattern's Algorithm as described in the formal description.

4.2 Formal Description

4.2.1 Pre-snapshot

A single process is designated as the initiator of the snapshot, which is known to every process participating in the global snapshot. Every process is identifiable by some known, shared ID. Every process begins coloured white.

For a given process P in a complete network of N processes, each with $N-1$ bidirectional channels, P will store the following for each channel C:

- `Seq` $\leftarrow$ Last sequence number of the last message sent to C
- `Ack` $\leftarrow$ Ack of most recent sequence number received from C
- `Last` $\leftarrow$ Acknowledgement of packet processed from C
- `Msg` $\leftarrow$ Message buffer for holding unprocessed messages from C
- $\texttt{Seq}_{\texttt{white}}$ $\leftarrow$ Last sequence number of message sent to C before snapshot recording
- $\texttt{Rec}_{\texttt{white}}$ $\leftarrow$ Recording buffer for holding unprocessed white messages from C during snapshot recording
- $\texttt{Last}_{\texttt{white}}$ $\leftarrow$ Ack of last white message received from C unprocessed before snapshot recording
- $\texttt{End}_{\texttt{white}}$ $\leftarrow$ Ack of last white message that should be received from C before snapshot recording

Assumption: Buffers are 0-indexed and start empty and integer variables start at -1

Algorithm 1 On Send Message to C

```
1: Seq ← Seq +1
2: send {Message, Seq} to C
```

Algorithm 2 On Receive {Message, Rcv} from C

```
1: if Ack < Rcv then
2:     Ack ← Rcv
3:     Expand Msg to contain all messages from [Last + 1, Rcv]
4:     Msg[Rcv - (Last + 1)] ← Message
5: else if Last < Rcv < Ack then
6:     Msg[Rcv - (Last + 1)] ← Message
7: else if Rcv < Last then
8:     /* Do nothing */
9: end if
```

When P is ready to process the unprocessed messages, the following will occur:

- For every channel, P will copy all the unprocessed messages into a complete unprocessed message buffer `Complete`:

$$\texttt{Complete} \leftarrow \texttt{Complete} \cup \{m \mid m \in \texttt{Msg} \wedge m \neq \text{null}\}$$

- Then, the message buffer is cleared:

$$\texttt{Msg} \leftarrow \emptyset$$

- And `Last` is updated to the most recently processed acknowledgement:

$$\texttt{Last} \leftarrow \texttt{Ack}$$

`Complete` is then delivered to P and processed, updating the local state of P in the process. As this refers to multiplayer game play, it occurs at the end of a game tick.

The local state of P is therefore a function of both:

1. Events occurring locally at P, and
2. Events triggered by the delivery and processing of messages in `Complete`.

4.2.2 Snapshot Initiation and Detection:

When the initiating process wants to record a local snapshot, the process will act as if they received a marker from a non-existent channel. The white process will turn red, thereby colouring all its subsequent messages red. The process will record its local state immediately. Similarly, when a non-initiating process receives a marker (red) message on a channel, it will turn red, record its local state, and send out markers to all other N-1 channels.

For every channel C, a process P will record:

$$\texttt{Rec}_{\texttt{white}} \leftarrow \texttt{Rec}_{\texttt{white}} \cup \texttt{Msg}$$
$$\texttt{Last}_{\texttt{white}} \leftarrow \texttt{Last}$$
$$\texttt{Seq}_{\texttt{white}} \leftarrow \texttt{Seq}$$

P will piggyback $\texttt{Seq}_{\texttt{white}}$ to all packets sent out to channel C. Then, P will send out a marker on all channels. Finally, P will wait to receive markers on all $N-1$ of its channels.

4.2.3 Receiving Marker Messages:

When a process P receives a marker (red) message for the first time on a given channel C, it does the following:

On receipt of $\{\texttt{Message}, \texttt{Rcv}_{\texttt{white}}\}$ from channel C:

1. $\texttt{End}_{\texttt{white}} \leftarrow \texttt{Rcv}_{\texttt{white}}$
2. expand $\texttt{Rec}_{\texttt{white}}$ to accommodate for white messages $[\texttt{Last}_{\texttt{white}} + 1, \texttt{End}_{\texttt{white}}]$

P will then perform the actions described in the snapshot initiation and detection section. For any subsequent markers on other channels, there is no need for P to re-enter a recording state.

4.2.4 Snapshot Recording

During the recording of messages, processes will insert messages into each channel's unprocessed message buffer, but will also attempt to insert these messages into their recording buffer $\texttt{Rec}_{\texttt{white}}$. There are two distinct situations to account for:

- **A marker (red) message has not been received by the channel yet:** Given a message can be inserted into Msg, it should also be inserted in $\texttt{Rec}_{\texttt{white}}$. This means that the last message of Msg should always be the same as $\texttt{Rec}_{\texttt{white}}$ whilst the marker message for the channel hasn't arrived. If Msg needs to be expanded to accommodate for a newly received white packet, $\texttt{Rec}_{\texttt{white}}$ should also be expanded in the same manner.
- **A marker (red) message has already been received by the channel:** Given a message can be inserted into Msg, it should also be inserted in $\texttt{Rec}_{\texttt{white}}$ if its sequence number is less than or equal to $\texttt{End}_{\texttt{white}}$. $\texttt{End}_{\texttt{white}}$ is the sequence number of the last white message that should be received by the process, which means that any messages with a lower sequence number must also be coloured white.

Checking that the message can be inserted into Msg verifies that the message can be inserted into $\texttt{Rec}_{\texttt{white}}$ because this indicates that the message will eventually be delivered to the process. If a message cannot be inserted into Msg, it should never be inserted into $\texttt{Rec}_{\texttt{white}}$.

4.2.5 Snapshot Termination

A process knows when to terminate the recording of white messages when the following conditions are met:

1. The process has sent and received marker (red) messages on every channel.

2. For every channel C that has received its marker (red) message, C will finish recording its messages once its values of $\texttt{End}_{\texttt{white}}$ and $\texttt{Last}$ satisfy the condition:

$$\texttt{End}_{\texttt{white}} \leq \texttt{Last}$$

3. Every channel has finished recording its messages.

Once these conditions are met, the local process has finished recording its local snapshot. The process's local snapshot consists of:

1. The local state snapshot recorded by the process when it received the first marker message, and

2. Each $\texttt{Rec}_{\texttt{white}}$ buffer from all $N-1$ of its channels.

The adjusted algorithm can be considered in terms of the naive version of Mattern's Algorithm. $\texttt{End}_{\texttt{white}}$ can be thought of as the "count" of white messages that another process has sent through a channel in the snapshot, and $\texttt{Last}$ as the "count" of white messages received from a channel.

We know that this algorithm will terminate as long as a message is eventually received with a sequence number greater than $\texttt{End}_{\texttt{white}}$, because a process will eventually process all messages from the range:

$$[\texttt{Last}+1, \texttt{Ack}]$$

Leading to:

$$\texttt{Last} \leftarrow \texttt{Ack}$$

Since $\texttt{End}_{\texttt{white}} \leq \texttt{Ack}$, it follows that:

$$\texttt{End}_{\texttt{white}} \leq \texttt{Last}$$

and therefore the algorithm will terminate.

Given that every message with a sequence number greater than $\texttt{End}_{\texttt{white}}$ is a marker (red) message, it follows that if a snapshot is detected, then the snapshot recording will terminate.

Furthermore, we know that for every channel C, the recording buffer $\texttt{Rec}_{\texttt{white}}$ is an accurate representation of the white messages processed by the process. This is because a message can only be inserted into $\texttt{Rec}_{\texttt{white}}$ if it can be inserted into $\texttt{Msg}$. Since $\texttt{Msg}$ is a buffer of messages that will eventually be delivered to the process, it follows that every message in $\texttt{Rec}_{\texttt{white}}$ will also be delivered to the process.

4.2.6 Snapshot Assembly

Once a local snapshot is complete, the process sends its local snapshot to the host process directly. However, in a packet loss context, it cannot be ensured that the local snapshot reaches the host process.

To combat this, every packet from the initiating process piggybacks information indicating the processes from which it has received local snapshots. When a process receives a packet from the host process, it checks this information in the packet header. If the process

finds that the host hasn't received its local snapshot, the snapshot is resent to the host. Otherwise, the process can return to a pre-snapshot state.

Once the host has received the local snapshots of all processes, the host can compile a global state, and the global snapshot is complete.

5 Implementation Details

Most of the details of the actual implementation are covered by the formal description explained in the previous section. However, its important to highlight a few details that also contribute to the overall implementation of the algorithm in the project:

1. **Packet Resending**: The implemented UDP networking protocol permits the resending of packets throughout the length of the entire application. This is implemented using a mix of constant-size sequence buffers allocated for every out-going channel for a given process, as well as piggybacking multiple acknowledgements onto a single packet. This feature introduces issues such as the arrival of duplicate packets at a given process, which is accounted for in the project implementation.

2. **Three different snapshot colours:** In addition to the "white" and "red" colours mentioned in the previous section, a third colour "blue" is necessary in order to be able to perform multiple snapshots during game play. The recording logic however changes minimally with this addition.

3. **Reliability guarantees:** In terms of the distributed application (the multiplayer game), the implementation of the algorithm guarantees that if a request to place a fence reaches a client before a snapshot is recorded, then it will be recorded as a part of that client's local state. This allows the loading of the global snapshot back into the game to be simple to implement, as it is guaranteed that no fences are missing or misplaced.

6 Future Directions and Enhancements

The following enhancements would be considered for future work:

- **Fragmenting packets and multiple messages per packet:** Currently, the algorithm assumes that a packet can only carry a single message. Most UDP networking protocols allow a packet to carry multiple messages, as it decreases the likelihood that client messages are lost in transmission. Additionally, packet sizes are limited to the maximum UDP packet transmission size, which is around 1500 bytes. In many cases, a local snapshot can be much larger than this, which would require the algorithm to define a reliable method of packet fragmentation and reassembly.

- **Reliable distribution of the game save:** The current implementation of distributing the game save file when a lobby is started does not account for issues in reliability. This means that if a packet is lost when starting the game lobby, the impacted client will not be able to join the game. The UDP networking protocol accounts for packet-loss in other areas of the game, including the general receiving of messages. This logic was not extended to the distribution of the save due to time constraints, but it's future implementation would lead to a more complete networking protocol.

7 Conclusion

This paper presented the challenges of implementing a snapshot algorithm in an online multiplayer game with unreliable UDP communication channels. After surveying relevant algorithms, including Chandy-Lamport, Lai-Yang and Mattern's parallel snapshot, we identified an augmented version of Mattern's algorithm as the most viable. This was due to its consideration of non-FIFO communication channels and not needing to store extensive message histories.

However, we discussed the issues caused by packet-loss in the algorithms termination conditions. To handle this issue we proposed an alternative termination condition that accounts for packet-loss. The implementation demonstrates it is possible to perform a complete and consistent snapshot in a real-time distributed system over a UDP connection.

References

[1] Mukesh Singhal Ajay Kshemkalyani. Chapter 4: Global state and snapshot recording algorithms, 2008. URL: https://www.cs.uic.edu/~ajayk/Chapter4.pdf.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb 1985. doi:10.1145/214451.214456.

[3] A D Kshemkalyani et al. An introduction to snapshot algorithms in distributed computing. *IOP Science*, 2(224):1–11, 1995. doi:10.1088/0967-1846/2/4/005.

[4] Glenn Fiedler. Building a game network protocol. URL: https://gafferongames.com/categories/building-a-game-network-protocol/.

[5] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4):423–434, 1993. URL: https://www.sciencedirect.com/science/article/pii/S0743731583710750, doi:10.1006/jpdc.1993.1075.