Implementing a Simple Scalable Reliable Multicast (SRM) Framework with Adaptive Loss Recovery

COMP90020 Distributed Algorithms - Team Snorlax - Project Report

Bingzhe Jin (1080774), Kaixun Yang (1040203), Shizhan Xu (771900), Zhen Cai (1049487)

ABSTRACT

This report describes Scalable Reliable Multicast (SRM), a negative acknowledgement (NACK) -oriented reliable multicast framework (Floyd et al, 1997) [1], and its implementation. Algorithm ensures receiver-based model of reliability for asynchronous systems on the application level. By referring to term "reliable", it guarantees properties of integrity, validity and agreement, i.e., if one non-faulty process delivers a message *m*, then *m* will be eventually delivered (overall liveness) to all group members with no duplication, unless all nodes holding *m* have already crashed. However, ordering isn't fulfilled. We investigate related research on this topic and compare SRM to other approaches. The loss recovery method is adaptive to the length of repair delay and number of duplicates, making SRM efficient across a broad variety of underlying network typologies. Furthermore, report demonstrates a distributed game application that runs on top of the framework, its mechanism and what part the framework supports. The intended audience is the project marker(s) of COMP90020 at the University of Melbourne.

Keywords Reliable multicast · Adaptive methods

1. PROBLEM CONTEXT

1.1. IP Multicast

Multicasting is a method of disseminating single copy of information from one host to a set of interested recipients, where such set is called a multicast group. This one-to-many communication helps save network bandwidth by sending only one stream of data instead of multiple unicasts. Although there's another concept called application-level multicast (ALM), that will be introduced briefly in Section 1.2, our focus is on network level. IPv4 multicast reserves the special class-D address range, i.e., 224.0.0.0-239.255.255.255. According to RFC-3376, Internet Group Management Protocol (IGMP) is used by IP hosts to submit a request before it joins and leaves any group. Hosts and routers advertise their memberships to any neighbouring switches and routers. [2]

1.2. ALM

Unlike IP multicast, ALM does not require any additional protocol for routers, it uses the traditional unicast IP. One of its architectures has dedicated proxies that exchange content among themselves after which end clients can download, rather than clients share their data directly on their own. Such architecture is easier to deploy, since no change needs to be applied to existing network infrastructure; also, it supports for higher-level functionality, e.g., can utilise TCP for congestion control. Physical links, on the other hand, may have redundant traffic as trade-off if the overlay network spans considerably different from a real-world network topology. There is overhead on constructing efficient overlays. [3] As a result, we settle on old-fashioned IPv4 multicast as the base model as it's enough, and it's built-in to Java hence handy to extend.

1.3. Requirements

Because most IP multicast applications must be UDP-based, datagrams aren't guaranteed to arrive in an ordered, intact, lossless manner. Lecture covered 3 possible ordering styles, which that FIFO/causal are orthogonal to total and they can combine. [4] UDP checksum almost overcomes the integrity problem since it has only a very low chance to fail false positive. Thus the real issue is reliability, which is defined to aggregate properties of integrity, validity and agreement.

- Integrity: A correct process delivers an undamaged message at most once.
- Validity: If a correct process multicasts message *m*, then it will eventually deliver *m* to itself.
- Agreement: If a correct process delivers a message *m*, then all other correct processes in the target group of message *m* will also deliver message *m*.

That is, if one non-faulty process delivers a message *m*, then *m* needs to be eventually delivered to all group members with no duplication, as long as there are at least one multicasting processes holding *m* still alive in the system. [4]

2. BACKGROUND RESEARCH

2.1. Naïve R-multicast

Lecture mentioned a naïve R-multicast over B-multicast. [4]

```
On initialization:
    Received := {};

On receiving m at process q from p:
    if (m not in Received):
        Received += {m};
        if (q != p) then B-multicast(group, m);
        R-deliver(m);
```

The problem is self-evident. Even if no one in group g experiences any loss, q is under a compulsion to forward a message once every time it receives one, i.e., each message requires $|g|$ flooding. This is very inefficient.

Also, because the integrity of this approach relies on the underlying communication medium, where the process itself does not detect duplicates, a rare case can be a message was sent once but arrives twice; Suppose processes may only fail by crashing, agreement will be achieved only if communication is over reliable channel. That means if a temporary network partition stops a node receiving any messages from the group for a certain amount of time; when it rejoins, its state will be different to others.

In the following section, these problems will be addressed.

2.2. RMTP

As part of many communication protocols, passing acknowledgement (ACK) messages is a common technique to confirm receipt. In our scenario, a data sending source expects time-bound ACKs from target receivers, and if it does not receive replies from one data receiver, it considers that one has a high chance of missing its multicast, so then initiates a retransmission at appropriate time and awaits retransmission ACK, so on so forth. This retransmission could be either selective or to all, but clearly both improve efficiency and satisfy agreement. Furthermore, integrity can be handled by piggybacking a globally unique data identifier on each packet.

The Reliable Multicast Transport Protocol (RMTP) has a typical hierarchical structure which groups receivers into local regions or domains. The protocol dynamically assigns a designated router (DR) within each region, in order to collect others' ACKs and report to sender periodically. A DR is also responsible for retransmitting lost packets to selective nodes locally, as opposed to from the original source, therefore end-to-end latency has a significant decrease. [5] Since only one single ACK is generated per local region, implosion is less.

RMTP needs to decide a good ACK time bound B. For small B, a slow ACK leads to wasteful retransmissions. For large B, there will be a longer loss recovery period. It's suggested that variable B should reflect the observed network latency. The second difficulty is how to determine DR for a given multicast tree dynamically, which implies two algorithms: leader election and membership.

2.3. SRM

Negative acknowledgement (NACK) is another scheme for loss recovery, in which every member of group detects its individual loss actively, by detecting a gap between successively arrived sequence numbers, then announces a request.

This is where Scalable Reliable Multicast (SRM) comes in. Periodically, each member exchanges session messages to share knowledge about the highest sequence numbers for all active sources, let others know if they've received the newest packet, and what was lost. A session also contains a timestamp used to estimate the one-way distance between nodes, in a way similar to the NTP synchronization algorithm [6] but much simplified (described in Section 3.2).

A request can be repaired by any server that possesses a copy of the requested data. Requests differ slightly from traditional NACKs by sharing the same idea of XTP design [7]. Before sending a request for data from S, process P first delays a uniformly-distributed random time function of $d_{S,P}$, where this function is adaptive to repair delay and count of request duplicates (discussed in Section 3.4). If A receives a duplicate request from B before its timer expires, then B suppresses A and A postpones its timer. The concept of repair timer is similar, and a repair is to everyone as well, thus a crash at a single node will have little impact on the entire system. (both described in Section 3.3)

SRM does not distinguish a network partition and member's normal exit from the multicast session. Members keep sending data within the connected components during a partition as if partition is not present. After merge, all missed data will eventually be recovered across all partitions.

2.4. SRM vs. RMTP

RMTP can impose a "natural" total data ordering. In fact, RMTP scales better than SRM because SRM's tasks are more evenly distributed so a node should have higher capacity requirements, whereas RMTP can choose DRs with the lowest computational load, making a leaf node feels better. SRM, on the other hand, avoids RMTP's election and gets rid of ACK bounds. In general, which one is better greatly depends on the use case of a distributed application and its underlying system architecture.

3. THE SRM FRAMEWORK

3.1. Assumptions

This section briefly outlines some assumptions about SRM:

1. SRM isn't paired with any cryptographic protocol by default to provide communications security.
2. Each member is identified with a unique and persistent ID; in our implementation, it's by concatenating IP address, port number and process id thus unique in LAN.
3. Non-static membership.
4. The data space is subdivided into "pages", in which sequence numbers are precise enough to never wrap.
5. Multicast is many-to-many.
6. Asynchronous system, i.e., boundless process execution time, message delivery time, clock drift rate. [6]

3.2. Session Messages

The basic idea of session messages has been illustrated in Section 2.3. For example, if process A just receives B's session that declares C's highest sequence number is known to be 10, while the state table of itself tells 8, then A can realise messages 9 and 10 from C have been lost or slow in transmission. Members can use session messages for tracking membership, although this step remains optional. The bandwidth consumed by session messages is set to 5% of the total. Once it's measured more than this percentage, the session rate drops by that much which attempts to make it meet 5% again in the next checking round. The rate min/max bound are per 10 seconds and per second, respectively.

Instead of reporting all states for everyone who has ever sent data, each member just reports for the active sources on that page it's currently viewing to prevent explosion. We define "current-viewing" to only include part of states of which highest sequence number has been updated within the last minute. One that receives a session browses over all states rather than only those recent, performs updates, and schedules requests when needed.

Assuming path symmetry, the host-to-host distances are estimated by averaging every two successive timestamps difference, in which synchronized clocks are not essential. Let A sends session packet at time t_1, B receives at t_2; and at some later time t_3, B sends session marked with $t_2 - t_1$, received by A at t_4, then the distance is $((t_4 - t_3) + (t_2 - t_1))/2$. The next section explains how these calculated distances help determine request/repair delays.

3.3. Loss Recovery

The request-repair mechanism is triggered when the process detects packet loss via data/session messages. Process includes the source address and the sequence number of a lost packet in the request message as a unique resource identifier, and schedules a repair, with a random time before it's really sent. It's crucial to note that not only data can be lost during transmission, but also requests, repairs, as well as the aforementioned session messages. Therefore, As a result, a request timer is programmed to not cancel requests after they have been sent, but to delay the next sending by double the waiting period - until the repair has been received.

The request timer is sampled from the uniform distribution $[C_1 d, (C_1 + C_2)d]$, where d is the host-to-host distance mentioned in Section 3.2. The adaptive parameters C_1 and C_2 are addressed in the following section. If the process receives the same request before its own request timer expires, the backoff mechanism is triggered, which implies the timer is resampled from the uniform distribution but multiplied by a parameter 2^i, where i represents the number of times the backoff mechanism is triggered. To avoid the backoff mechanism being triggered multiple times in a short period of time, when a backoff is triggered once, the process will set a heuristic ignore-backoff time, which is half of the gap between the current time and the request expiration time, and only re-trigger the backoff mechanism after the ignore-backoff time expires.

When the process receives the request message, if it possesses the missing data, it will schedule a repair task for a random time. When the timer expires, the repair message will be multicast. When the process multicasts the repair, the repair task will be canceled, unlike the request task, because even if the repair message is lost, the process can still accept the request again and reschedule the repair task.

The request timer is also sampled from the uniform distribution $[D_1 d, (D_1 + D_2)d]$, where d is again the host-to-host distance mentioned in Section 3.2. The adaptive parameters D_1 and D_2 are addressed in the following section. If the process receives the same repair before its own repair timer expires, it will cancel its own repair. Because there may be duplicated requests, the process will ignore the request message for data X within $3 * d$ unit of time after multicasting the repair message for data X.

3.4. Adaptive Methods

To deal with different network typologies[1], session memberships, and loss patterns, SRM employs adaptive parameters. The request timer settings C_1 and C_2 take into account the amount of duplicate packets and the repair delay in order to achieve a trade-off. SRM keeps track of how many duplicate request messages it gets before canceling each request task, then adjusts ave_dup_req with the following formula:

$$ave\_dup\_req \leftarrow (1 - \alpha)\ ave\_dup\_req + \alpha\ dup\_req$$

It also keeps track of how many times each request is multicast until it is cancelled, as well as the total time, for calculating ave_req_delay using the formula below:

$$ave\_req\_delay \leftarrow \\ (1 - \alpha)\ ave\_req\_delay + \alpha\ (req\_delay/n\_send)$$

where α is 0.25 in our application.

[1]Such as chains, stars, and trees.

C_1 and C_2 are adjusted according to the following:

```
On initialization:
    C1 = 2; C2 = 2;
    MinC1 = 0.5; MaxC1 = 2;
    MinC2 = 1; MaxC2 = G;
    AveDups = 1; AveDelay = 1;
    epsilon = 0.1;

After multicasting a request:
    C1 -= 0.1;

After a request task is cancelled:
    update ave_req_delay;
    update ave_dup_req;
    if (Process has the smallest on-way-distance
        to the missing data source) C2 -= 0.1;
    else if (ave_dup_req >= AveDups) {
        C1 += 0.1;
        C2 += 0.5;
    }
    else if (ave_dup_req <= AveDups - epsilon) {
        if (ave_req_delay > AveDelay) C2 -= 0.1;
        if (ave_req_delay < 0.25) C1 -= 0.05;
    }
    else C1 += 0.05;
```

where G is the group size. After each repair message is multicast, SRM will update D1 and D2 to $\log_{10} G$.

4. IMPLEMENTATION

SRM's ReliableMulticastSocket extends Java's built-in MulticastSocket class. We apply the facade pattern for encapsulation, hide system's complexity, and override methods resulting in an easy-to-use "interface". As a result, a user would experience few difference from the net.MulticastSocket class, only extra step for the addition of the word "Reliable" prior to the original class name.

4.1. Message Types

Messages in SRM are classified into four types: DATA, SESSION, REQUEST, and REPAIR. Application developers just need to care about sending and receiving DATA messages, all other types of messages are managed by SRM automatically.

DATA message's body only contains the payload that the application needs to send. DATA message is a type of message that is used to communicate between applications.

SESSION message's body contains a timestamp and part of StateTable, which holds the highest sequence numbers and previous distance estimates for active sources in the last minute. The exchange of these allows state updates.

REQUEST message's body contains the source address and sequence number for the missing data, as well as the one-way distance from itself to the source. This enables nodes to request repairs for the missing data.

REPAIR message's body contains the source address and sequence number for the missing data, as well as the missing data payload. This enables nodes to repair requests.

4.2. Dispatcher and Data Cache

A background thread continuously receives messages, and dispatch tasks to different components based on their message type. We implement a message cache and a blocking queue holding unconsumed messages to feed data payload[2] to users. The handling of requests and repairs is covered in the following section.

4.3. Request Repair Pool

ThreadPoolExecutor is used as a thread pool for processing repair and request tasks. SRM assigns each request and repair task with a thread.

SRM creates the RepairTask and RequestTask classes, which both implement the Runnable interface and override the run() method to send messages and update adaptive parameters. An infinite *while* loop in the run() method of RequestTask ensures that the request can continue to be sent until the task is cancelled.

SRM keeps track of each thread's Future object in a ConcurrentHashMap so that the task can be aborted. SRM uses Thread.sleep() to accomplish the delay, and catches InterruptedException to accomplish the task cancellation or postponement.

5. APPLICATION

5.1. Game Process

Based on the SRM algorithm, we developed a game called Draw and Guess. A few players join a room via joining the same multicast group, in order to constantly multicast their status and receive others' status to push forward the game progress. In the game, each player chooses a word to start with and repeat the process of drawing and guessing until everyone has drawn and guessed enough turns. During the draw turn, players are given either a starting word or the word guessed by the previous player to draw. During the guess turn, players look at the drawing drawn by the previous player and try to guess the word. After submitting drawing or guessing each turn, players update their personal status by adding the new drawing or guessing, and multicast the new state to all other players. After receiving the updated status of all players, the next turn begins. After reaching the predefined turns, the current round terminates and an animated reply of all drawings and guessing from all player is displayed. New

[2]Can be from REPAIR.

round will begin afterwards, in which all players will be given new starting words. Finally, players will stop multicasting and exit the game when the predefined number of rounds has been completed.

A major difference of this game from akin games is how we utilize multicast to decentralize the communication structure. Players themselves monitor and control the flow of the game rather than a central server. However, the notorious drawback of ordinary multicast algorithm threatens this decentralized game such that one packet loss may cause some players to be inconsistent. To begin with, a player can never end the game properly if the last packet from another player is lost. SRM is the crucial key to prevent such happening.

5.2. System Design

As described above, our system consists of two multicast groups: the lobby and the room. The host of the room is responsible for multicasting the room information to both the lobby and within the room itself.

There are a few interesting design decisions worth noting in this section. Firstly, because of multicast's nature of having no knowledge of any member in the group, it is hard to determine whether a member has left or just lagging. Usually, systems would adopt a timeout mechanism to determine. In response to this problem, we designed two ways of inactive member detection that utilized timeout: in the lobby, received rooms are kept in the memory and there is a refresh button to manually load these rooms and remove those timeout ones at that point of time; while in the room, a thread waits on the currentRoom object and gets notified whenever it is updated by newly received multicast messages. The former only suits in situations where consistency is not strictly required, while the latter's time complexity is as high as number of members times update operation.

Secondly, our design inherits Java's existing multicast socket class, therefore inherits some undefined behaviours as well. Throughout the development, we have discovered that the port number of sockets must be the same as the group to multicast, and multiple multicast sockets are allowed to be initiated with the same port number as a result. Nevertheless, having multiple sockets with the same port number joining the same group will result in only the first socket being able to send any message to the group. We are unsure if this behaviour is intended in original Java, but our design was definitely influenced.

5.3. Embedded Election Algorithm

Although not the main focus of this project, since the host is essential for advertising the room information to the lobby and other members in the room, we needed an election algorithm for selecting a new host when the original host of a room disconnects.

The username in this game is assumed to be unique by taking user input combined with randomly generated four digit number. As a result, users can be sorted according to the alphabetic order of their usernames. Given that in the room, all players keep real-time data of all other players by monitoring their multicast messages, each process knows and can communicate with all other processes are satisfied. Furthermore, our SRM algorithm guarantees reliable communication channel. As our game runs on the same LAN, the system can be assumed as synchronized. Conclusively, all the requirements of the Bully algorithm for election are met.

Unsurprisingly, our election is achieved via an alteration of the Bully algorithm. As our players are ordered by their names and all players share a real-time updating list of all players, the largest existing player always wins the election and start sending room information.

In our system, each player's periodical multicast messages serve the purpose of all the election related messages. Only the host would multicast room information, therefore room information messages are treated as coordination message. There is no need for election message, for the absence of room information message would trigger all active players to enter an election. All players' periodical self-advertise messages serve as the answer message. Consequently, we achieve an election mechanism similar to the Bully algorithm without using any additional bandwidth.

6. FUTURE WORKS

6.1. Scalable Session Messages

In some cases, certain members may not be able to reach a proportion of others when the system grows large. To improve this, one suggestion is to elect one representative in each local area, and representatives connect under a global context. All other members would have their representative as a proxy, but still need to ensure limited scope sufficient to reach them.

6.2. Congestion Control

We mentioned congestion control in Section 1.2; and briefly in Section 3.2 when discussing the relation between session rate and bandwidth consumption. But this problem for SRM remains not completely solved, as no mechanism yet has been implemented to control congestion for normal data transfer.

DeLucia and Obraczka published a conference paper on multicast congestion control techniques, in 1998. The algorithm dynamically selects a small number of representatives to provide immediate feedback on behalf of the multicast group member, allowing the source to adjust based on the current state of the network. A probabilistic feedback suppression also decreases the quantity of feedback generated while eliminating the need for precise timer settings based on RTT to the source. [8] Recall in the previous section, representatives were elected, we feel these two can share the same set of those. Furthermore, the Real-time Transport Protocol (RTP) is another way to help control session rate.

6.3. Local Recovery

Sessions with persistent losses to a small neighborhood of members is an example of scenarios that could benefit from local recovery. A few ideas were suggested in the SRM paper [1], but we did not really implement this part in our project. How to define a node's local area, or what scope to use, is a question worth thinking about.

6.4. Security Concerns

Java's built-in MulticastSocket does not have SSL/TLS in it, so not as well our implementation of SRM. To ensure data security, encryption is needed. The issue is the handshake that we can't just make it loss-free because handshake happens before recovery starts. Is it truly safe to share handshake keys to the whole group given the recovery is in place? [3] Even if the answer is yes (definitely no), then the encryption is no longer needed. Here we believe that some sort of process should be there in order to verify a member's access permission before it joins the group. The implementation is left open for future works.

7. CONCLUSION

This paper has described in detail the SRM framework and certain design choices for developing a distributed application that runs on top of it. SRM isn't a perfect solution; some alternatives, such as NACK-Oriented Reliable Multicast (NORM) Transport Protocol [9], are better than it, while SRM is enough for us to experiment on multicast reliability. Future works on scalable session messages, congestion control, local recovery, and security concerns have been discussed.

8. REFERENCES

[1] Floyd, S., Jacobson, V., Liu, C.-G., McCanne, S., & Zhang, L. (1997). A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Transactions on Networking, 5*(6), 784–803. https://doi.org/10.1109/90.650139

[2] IETF Datatracker. (2002, October). *RFC 3376 - Internet group management protocol, version 3.* https://datatracker.ietf.org/doc/html/rfc3376

[3] Siegenthaler, M. (2006). *Advanced Course in Computer Systems CS614 Application-Level Multicast Routing - 2006* [Lecture notes]. http://www.cs.cornell.edu/courses/cs614/2006fa/Slides/overcast.pdf

[4] Islam, T., Kulik, L., & Rodriguez, M. R. (2022) *Distributed Algorithms COMP90020 Lecture 6 Coordination and Agreement: Multicast - 2022* [Lecture notes]. https://canvas.lms.unimelb.edu.au/courses/124747/files/10963178?module_item_id=3733142

[5] Paul, S., Sabnani, K. K., Lin, J. C.-H., & Bhattacharyya, S. (1997). Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications, 15*(3), 407–421. https://doi.org/10.1109/49.564138

[6] Islam, T., Kulik, L., & Rodriguez, M. R. (2022) *Distributed Algorithms COMP90020 Lecture 1 Time & Global States: Introduction and Clock Synchronization - 2022* [Lecture notes]. https://canvas.lms.unimelb.edu.au/courses/124747/files/10664984?module_item_id=3654587

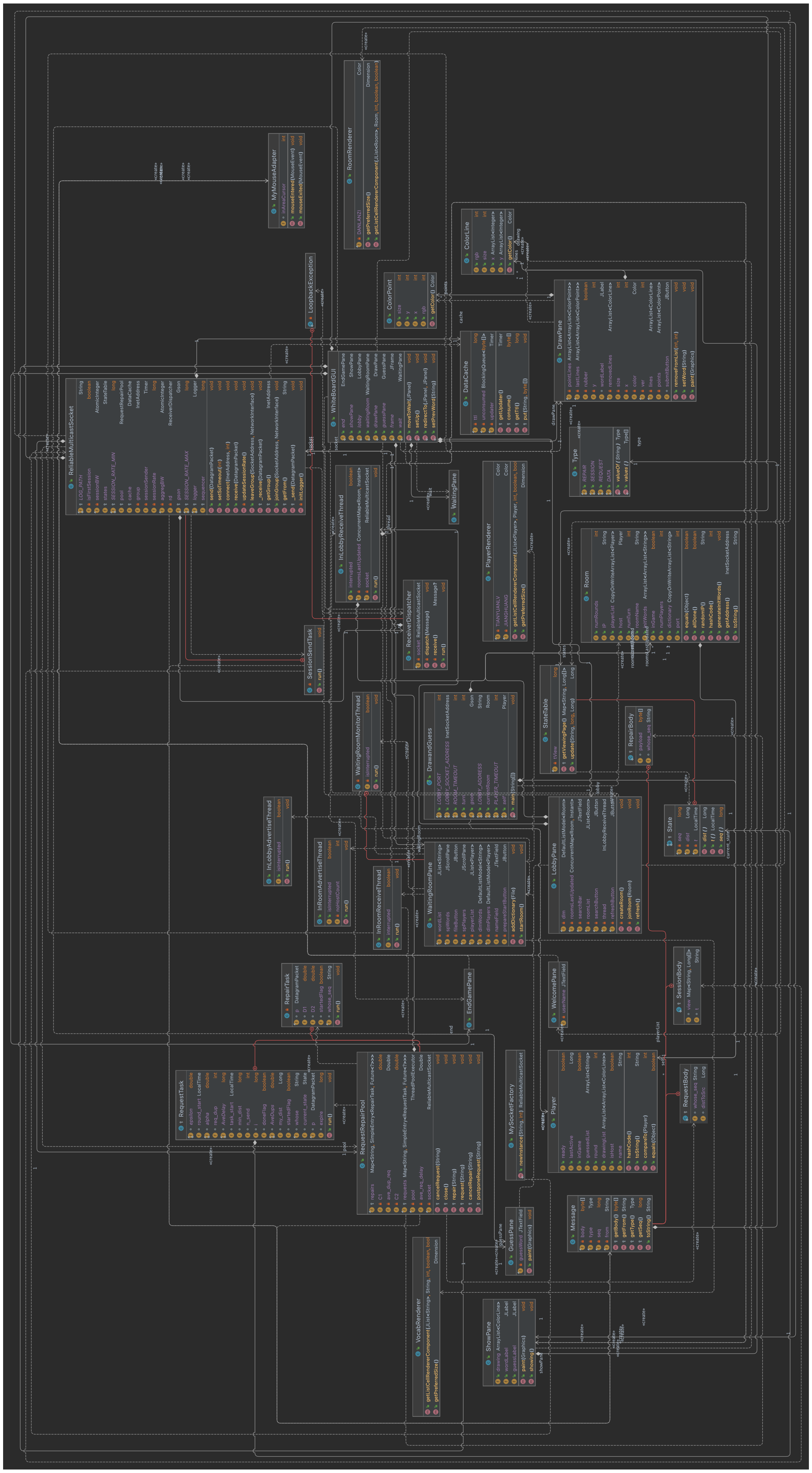
[7] Strayer, W. T., Dempsey, B. J., & Weaver, A. C. (1992). *XTP: The Xpress transfer protocol.* Addison-Wesley Publishing Co., Inc., Redwood City, CA.

[8] DeLucia, D., & Obraczka, K. (1998). A multicast congestion control mechanism for reliable multicast. *Proceedings Third IEEE Symposium on Computers and Communications. ISCC'98. (Cat. No.98EX166), Computers and Communications, 1998. ISCC '98. Proceedings. Third IEEE Symposium On*, 142–146.

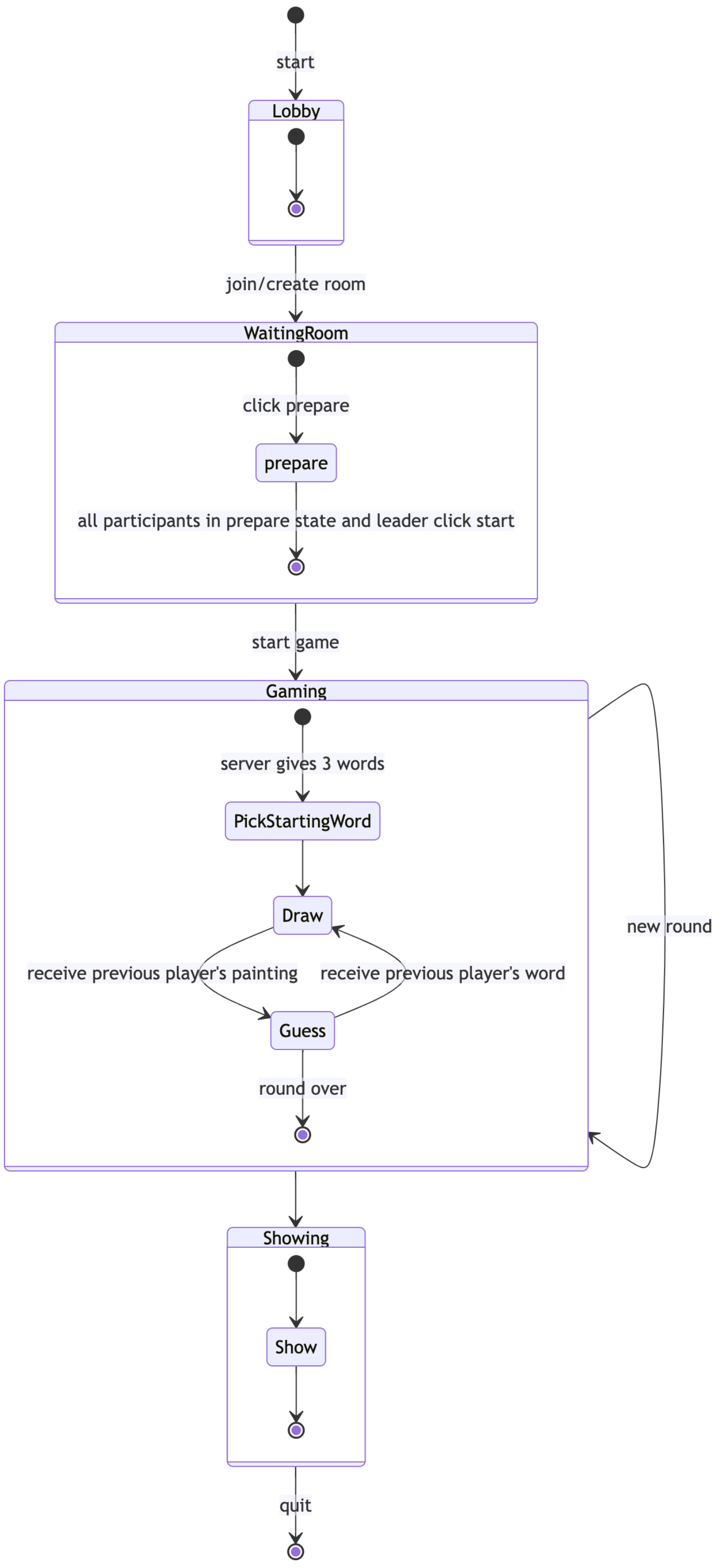
[9] Adamson, B., Bormann, C., Handley, M., & Macker, J. (2009, November). *NACK-Oriented Reliable Multicast (NORM) Transport Protocol*, RFC 5740. https://www.rfc-editor.org/info/rfc5740

[3] Think about the Man-in-the-middle attack.

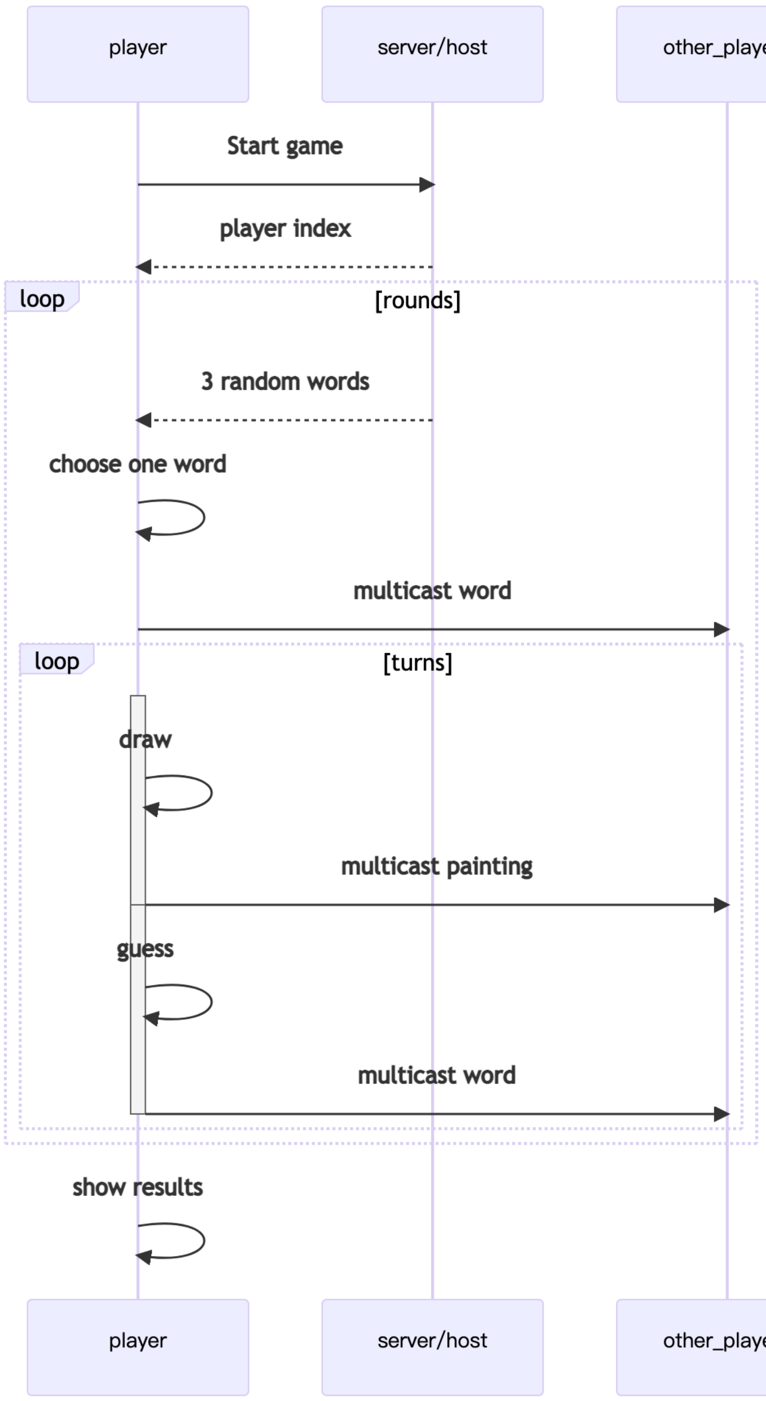
APPENDIX 1: Design Class Diagram



APPENDIX 2: State Machine



APPENDIX 3: Sequence Diagram


player
server/host
other_players
Start game
player index
loop
[rounds]
3 random words
choose one word
multicast word
loop
[turns]
draw
multicast painting
guess
multicast word
show results
player
server/host
other_players