Questions and Answers

Mansoor Malik

1. Evaluation of Starting, Maintaining, and Exiting a Game

In our design, a player joins a game by sending a single JoinGameRequest message that is multicast to all nodes. Each node already playing in a game responds by sending a JoinGameResponse message to a multicast address.

The first strength of our design is that nodes are discovered dynamically. No node acts as a registry for players. Thus there are no single points of failures. Another strength is that all communication takes place by multicasting messages over UDP. No additional sockets are needed for point-to-point communication between nodes. Using a single multicast address and port simplifies the implementation. It also minimizes the number of messages that a node must send.

A weakness of this approach is that UDP is unreliable and therefore additional mechanisms are needed to deal with lost or delayed packets or packets that are delivered out of order. Our design relies on sequence numbers, retransmission, and requests and responses to address these shortcomings. These mechanisms are discussed in more detail later. Also, using requests and responses requires that an addressing scheme be added on top of the UDP protocol. In our protocol, a 4-byte player ID is used to identify players. There is also a reserved address when a node wants to broadcast a message to all other players.

A Mazewar game supports a maximum number of players. When this limit is reached, additional players must be prevented from joining. Our design supports this basic requirement by having nodes respond with a RejectMaxPlayerLimitReached code.

When attempting to join a game, each player picks an initial random 4 byte identifier (ID). This mechanism has strengths and weaknesses. The weakness is that a player may pick an ID that is already in use. If this occurs, the JoinGameResponse messages will include a RetryDuplicatePlayerId code. The node that is sending a JoinGameRequest must then retry with a different ID. The probability of this event occurring is low since the sample space consists of (2^32)-1 possible entries whereas the maximum number of players is limited to 8. We selected this scheme because the other alternative was for the existing nodes playing in a game to propose an ID that was currently unused or to elect a single leader that was responsible for issuing IDs to new players. We did not like the first approach because it required that all existing nodes reach consensus on a common ID and we did not like the second approach because electing a leader would complicate the implementation.

A node that sends a JoinGameRequest may not receive a response to its first request. However, the requestor does not know if this happened because no other node is present or because a packet containing a request or response was dropped. To allow for transient network congestion, a node must retransmit two additional JoinGameRequests and must increase its timeout using exponential delays of 1, 2, and 4 seconds. The

strength of this approach is that multiple retry attempts should decrease the probability that requests or responses are missed because of lost of delayed packets. The weakness is that it increases the time required to start the first game. For example, the first player joining a game will have to wait 7 seconds before its game will begin. On the other hand, since there is no one else to play with, this may not matter to the player.

There is also a possibility that network congestion can occur for periods lasting more than 7 seconds. In this period, a large number of packets may be dropped. In that case, it is possible for a player to start a game assuming that it is the first player and subsequently start getting updates from other players. Our design handles this possibility by having a node send a ForceStopGame message to another node that it does not recognize. The strength of this approach is that it avoids the possibility of having independently running nodes. The weakness of this approach is that it kills a game in progress. In this case however, the first choice appears to be the lesser of the two evils.

To join a game, a node must first receive a JoinGameResponse with an accept code from all other nodes. The JoinGameResponse includes a field that lists the count of total players. This allows the requestor to ensure that the number of responses matches the number of players. If the count does not match, the new node will resend a JoinGameRequest message up to two additional times. If the count still does not match then a node must give up and retry again later. The strength of this approach is that it will deal with packets that may be lost or delayed and it will avoid games where nodes have not yet discovered each other.

Two players may attempt to join a game at the same time. In this case, both players must join not only with existing players but with each other. Our design handles this by only allowing one join to be in progress. In this event, the second node will get a response with a RetryAnotherJoinInProgress code. The second node must retry after waiting for a random time interval. The time interval increases with the 2^{nd} and 3^{rd} attempt. The strength of this approach is that it handles this corner cases but at the expense of making the implementation more complex.

A node sends a LocationStatus message whenever its mouse moves or changes direction. If one second have elapsed since the last message, a node must send an update even if it's rat has not moved. A single LocationStatus message is sent to all nodes. The first strength of this mechanism is that network traffic is minimized since a single message is sent to all players. The second strength is that this scheme is idempotent. The latest LocationStatus message contain the rat location so this scheme can tolerate lost or duplicate messages. Another strength is that a LocationStatus message can be used as a heartbeat to detect nodes that have failed due to a software or hardware failure. The Ratpack protocol specifies that if 10 seconds have elapsed since the last LocationStatus message was received, a node should be considered dead and removed from the maze.

A node leaves a game by sending a LeaveGameRequest. All remaining players must acknowledge this request by sending a LeaveGameResponse. Since the packets containing the request or response may be lost or delayed, the Ratpack protocol specifies that a node must retransmit the request 2 additional times if responses are not received from the remaining nodes. The strength of this approach is that the shared state of the remaining nodes converges quickly once a rat leaves a maze. The weakness of this approach is the additional complexity in the implementation.

2. Evaluation of Performance and Scalability

Our design does not use synchronization mechanisms to control the movement of rats or missiles. Thus, we do not expect performance to degrade as a game scales to accommodate a larger number of players. In addition, once a player joins a game, the majority of network traffic will consist of LocationStatus and MissileStatus messages. Since these messages are multicast to all nodes, the number of updates sent by a single node does not grow as additional users join a game. Thus using multicast is an effective way to scale the game.

There is a computational load on each server that arises from processing incoming packets. Since all packets are addressed to a single multicast address and port, nodes must examine and discard packets that are intended for other nodes. It is unlikely that this would become a bottleneck as the number of player increases.

The game is responsive when it is played on a local area network. This is to be expected since network latency is small. However, the network latency would be much larger if the game was played over a wide-area network. Since Mazeware is a first person shooter game, the latency in getting rat and missile location updates from other players must be minimized. If it is not, then there will be a greater likelihood of that the position of rats or missiles will have a more variation between each node. Nonetheless, this will not result in global inconsistencies as far as tagging rats or scores. This is described in more detail in the next section.

The probability of network congestion decreases as the capacity of a network is increased (10 Gbps versus 1 Gbps for example). This in turn means that the probability of encountering packet losses will go down. This will improve the performance of the game since fewer retransmission attempts will be required for messages such as TagConfirmationRequest and TagConfirmationResponse.

3. Evaluation of Consistency

The shared state of Mazewar consists of players, rat positions, scores, and missiles. We look at each element in turn.

To ensure a consistent state, all nodes must have the same list of players currently playing in a game. As described in the first section on joining, maintaining, and leaving a game, a number of mechanisms exist to ensure that all nodes are aware of every other node as a game is progressing. The strengths and weaknesses of the mechanisms for ensuring consistency for players were discussed previously so we do not repeat it here.

Each node advertises the position of its rat through LocationStatus messages. Packets may be delivered out of order when using UDP. Thus, a node that receives these messages can use the sequence number in the header to discard older updates. Since each instance of a game operates in real-time it is possible for two rats to move into the same empty position simultaneously in two different games. In this scenario, the global

state is no longer consistent. But our design permits this since we are not violating Mazewar rules as far as the local state of each game. Both nodes will subsequently receive updates from each other, and to prevent local inconsistencies, a node should discard a LocationStatus message if another rat occupies this position. The weakness of this mechanism is that it allows the shared state to diverge momentarily. However, this is an acceptable trade-off because ensuring strict consistency would require that all node reach prior agreement before allowing a move to occur. One way of achieving this alternate scheme might be to have a lock server with one lock for each cell position. A rat could only move into a cell after acquiring its lock. But this alternate scheme would have a single point of failure (lock server), it would have a high implementation cost, and it would be slow. Our scheme cannot guarantee strict global consistency but its strength when compared to other alternatives is its simplicity and speed.

When a node launches a missile in its local game, it sends a MissileStatus message to all other nodes. The message contains the missile's location and direction. It also includes a unique missile identifier. The ID of the shooter combined with the ID of the missile creates a tuple that uniquely identifies each missile in a game. Since the original message may be lost, MissileStatus updates are sent every 200 ms. A node should create a missile when it receives the first update and then use its internal clock to update the missile position. It should discard all subsequent updates for the same missile. The strength and weakness of this approach is similar to the strengths and weakness of sending LocationStatus messages. We cannot guarantee strict global consistency across all nodes and there may be timing differences as each node moves a missile position. The strength of this approach is its simplicity and speed. Since missiles can hit rats we have to prevent global inconsistencies from occurring as it relates to scoring. We thus make use of requests and responses for tags. This is described next.

If a node shoots a missile and that missile tags another rat, the shooting node must send a TagConfirmationRequest to the victim node. The victim node responds with a TagConfirmationResponse message in which they either accept or reject the tag. To allow for lost packets, the shooting node is allowed to send up to 3 requests. Because of small timing differences it is possible for a rat to be tagged in one instance of a game and to escape being tagged in another instance. The shared global state of the game will not be affected because both nodes (shooter and victim) must agree on the tag. If all 3 requests are lost then the shooter will not increase its score. If all three responses are lost then only the victim will decrease its score. This is still acceptable because the global state of the game as it relates to player scores will remain consistent. However, a weakness of this approach is it can allow a malicious user to cheat by rejecting a legitimate tag request. This point is discussed in more detail in the security section.

Since each node advertises its own score, the scores will be consistent across all nodes. The strength of this approach is its simplicity. However, there is a weakness to this approach in that it allows players to cheat. This point is discussed in more detail in the security section.

4. Evaluation of Security

Security was not an explicit goal when designing the Ratpack protocol. The assumption was made that games would be played in a trusted environment in which the implementation would adhere to the Ratpack protocol and no player would try to cheat.

Given the initial assumptions, the security weaknesses and some possible remedies are discussed below.

First of all it is possible for an attacker to mount a denial of service attack by sending ForceStopGame messages to nodes. This would prevent games from being played. The protocol would have to be redesigned to eliminate this message if we wanted to eliminate this weakness.

Another weakness is that players are not authenticated. Therefore, a malicious user can learn the identity of users by snooping network traffic. The malicious user can then send incorrect location and missile updates while masquerading as the true user.

Since all players advertise their own score, it is possible for a malicious user to design an implementation that increases their score without tagging another rat. However, in this case it is possible to detect this type of cheating. For example, all nodes can keep track of TagConfirmationResponse messages that are transmitted. A node that cheated would have increased its score without receiving a TagConfirmationResponse from another node. If a node sees this event from occurring, they can transmit a new message that we'll refer to as TagCheat. If all nodes (excluding the node that increased its score) send a TagCheat message then there is consensus that a node falsely increased its score.

The Ratpack protocol specifies that a player cannot launch a new missile until an existing one has hit a player or a wall. However, the existing Ratpack protocol has no mechanism for enforcing this constraint. A malicious implementation could allow a user to shoot multiple missiles. It is possible to remedy this flaw using a similar approach to the one described in the previous paragraph. If a node receives a MissileStatus message for a new missile while an existing one is still in flight, they could send a MissileCheat alert. If all nodes (excluding the one that launched the missile) send out this message then they have a consensus about a node that is possibly cheating. To allow for timing differences, the revised protocol may set a minimum number of times that a cheating event is detected before concluding that a node is indeed cheating.

The Mazewar game specifies that rats cannot go through walls, that rats cannot move more than one cell at a time, and that rats cannot go through other rats. The Ratpack protocol assumes that all implementations will follow these rules. Each node accepts a LocationStatus update from another node as if it is valid. The protocol would have to be redesigned if one of the objectives was to detect players making illegal moves. Similar to the discussion in the previous two paragraphs, all players could monitor the moves of all other players. If a possible breach of the rules was detected, all nodes (excluding the one that committed the breach) could broadcast a message flagging the possible breach. If a consensus were achieved, and this event occurred a number of times, then the nodes would know that one of the nodes was violating the rules.