Multiplayer Mazewar Protocol Specification v1.0

Gobind Johar <gobind@stanford.edu> , Ivan Marcin <ivmarcin@stanford.edu>

Introduction

This spec documents the protocol specification for a multiplayer version of Mazewar - a first person shooter first implemented by Steve Colley in 1974. The protocol is meant to extend mazewar into a networked multiplayer game. The game implementation is a dungeon crawler type game where player controlled "rats" walk through a maze firing missiles. In this multiplayer version -- players can see each other walking through the maze, and shoot others gaining points for killing while losing points for getting killed. The objective of the game is to score as many points as possible.

Table of contents

Protocol Definition and Message Specification

Objective

This protocol is based on a multicast paradigm with the objective of making Mazewar a multiplayer game. Clients connect to each other via a UDP multicast port and start exchanging their game states.

***Note**: All prescribed network communication that follows is in network byte order, i.e. big endian.*

General Fields

These are the general fields that appear throughout the specification and in different messages sent across the network. These form the basic components to represent game data being shared.

Field Name	Bytes	Description
Player ID	2	Unsigned integer representing player identification number
Action ID	1	Describes the type of packet (score, sync_req, sync_ack, etc.)
Position/Direction Vector	6	Combines the x, y position and facing of a player, 2 bytes per vector field
Score Token	2	Player score middle shifted to handle negative scores (i.e. a score of 0 is sent as 32768)
Player Name	15	String representation of a player name
Sequence Number	4	Monotonically increasing number to send packets with

Protocol Header

Each message should be sent as a UDP datagram, which must contain a header and a payload. The header must conform to the format specified below.

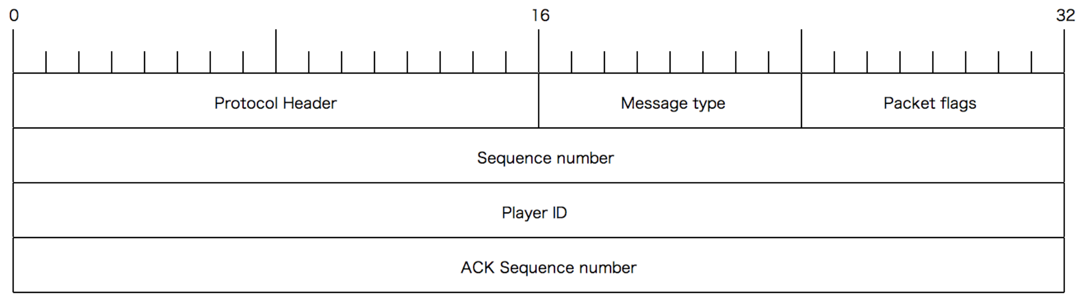


Fig. Message Header

Protocol Identifier: Datagram identification of length 2 bytes. The beginning of a Mazewar packet should always set this field to the binary value of **0xCAFE**. This will allow implementations to distinguish Mazewar packets from other packet types.

Message Type: 1 byte integer that indicates the message type, and how the packet should be parsed

Numeric ID	Action type
0x01	Sync Request
0x02	Sync Ack
0x20	Move
0x30	Kill Req
0x31	Kill Ack
0x32	Kill Denied
0x40	Keep Alive
0x10	Leave

Packet Flags: Bitmask field added for extensibility, currently only used to indicate if a packet needs to be transmitted reliably. Packets with the least significant bit set need to be retransmitted periodically until a matching Ack is received. This is covered in more detail later.

Flag	Description
0x0001	Reliability - Keep track of seq number and retransmit if ack is not received.

Sequence Number: Packet sequence number is a monotonically incrementing number used to identify the ordering sequence of messages sent by a client. Actions sent unreliably with a sequence number lesser than the last message processed should be ignored to avoid inconsistent moves when are received out of order.

Note: *Just like TCP, this protocol allows the sequence numbers to wrap around. This allows to protocol to accommodate a network delay that is fairly large and should not be hit under normal circumstances.*

Player ID: A unique unsigned 32-bit integer to identify an individual player. Assignment should be random by the implementation. A player that wishes to join an existing game session should listen to packets on the multicast port for a time slot before starting to broadcast to ensure its player ID does not collide with existing players. If a collision is detected, the player must generate a new random ID before joining the game.

Ack Number: The last reliable packet sequence number that was acknowledged by the player. This is only used for the reliable data stream.

Messages with Unreliable Communication

Most of the gameplay can be accomplished using unreliable communication. If states are out of sync, game updates are transmitted sufficient times for players to converge back to sufficient consistency.

***Note**: The following packets must set the unreliable flag bit at 0x0001 to 1 in the packet header*

Joining a Game

The Sync packet is used by a player to synchronize game statistics by advertising his local game state and requesting remote game states. Other players should register the new player as soon as a Sync Request is received. They must also reply back to a Sync Request with a Sync Ack containing their local player state. The originator of the Sync Request may add all the players that responded with a Sync Ack to his game. The Sync mechanism may be used under the following circumstances:

1. A player wishes to join an existing game
2. A player detects network timeouts and wishes to reconcile state with other players
3. A player picks up packets from unknown players and wishes to know about them

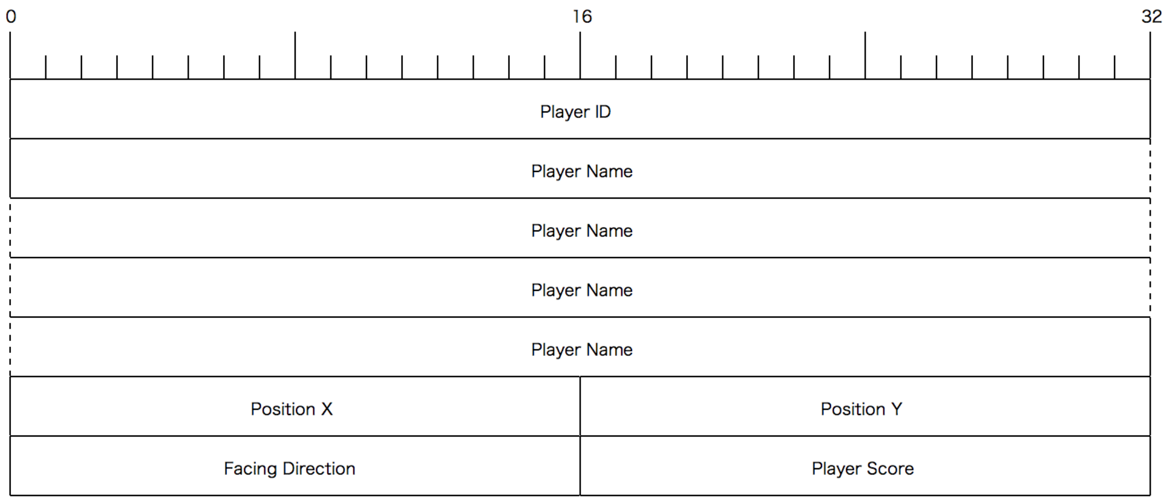


Fig. Sync Request and Sync Ack payload.

Sync Request

Type: 0x01
Acknowledgement: Sync Response
Payload:

- *Player Name:* capped to a maximum of 15 characters and null termination, i.e. 16 bytes
- *Position X*: 2 bytes for the X coordinate
- *Position Y*: 2 bytes for the Y coordinate
- *Direction*: 2 bytes for the direction
- *Score:* 2 bytes unsigned integer offset by 32768

Sync Ack

Type: 0x02
Acknowledgement: None
Payload: Same as a Sync Request, except that players shall not respond to this type of packet

Each player must respond to a Sync Request with a Sync Ack containing the necessary information. No player should send a response to a Sync Ack. This is to avoid running into Sync loops in the network.

Game Movement

Game movements may be used to update other participants about local movements, including position and direction change.

It is important to note that temporary conflicts may occur due to lack of strict consistency in some same-time move situations . Therefore, a conflict resolution strategy is used to break ties when such situations occur. For this protocol, the player with the lowest player ID shall win each tie. This is deterministic and will result in corrective action among all conflicting players. For instance, assume player A and player B move to position (1,1) locally and send updates to each other. A's player ID is lower than B. Thus, A will win the tie break and stay in (1,1). However, B must backtrack into its previous position and send another position update to everyone.

Note*: A player must not initiate a move to an already occupied position in his local maze.*

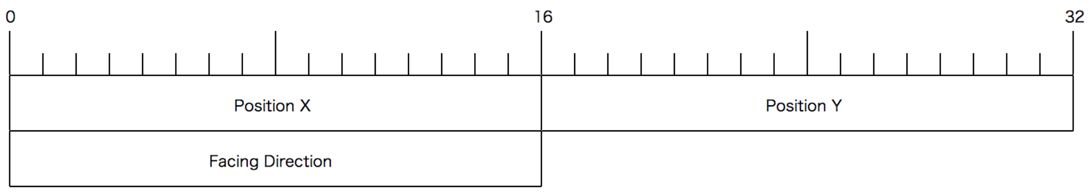


Fig. Move payload

Move

Type: 0x20
Acknowledgement: None
Payload:

- *Position X*: 2 bytes for the X coordinate
- *Position Y*: 2 bytes for the Y coordinate
- *Direction*: 2 bytes for the direction

Active Participation

Players should use a Keep Alive to indicate their participation in the game in case there is no local game activity detected and no messages are broadcasted for over 5 seconds. Each player must monitor packet activity from other players in the game. If players stop receiving any packets (including keep alive packets) from a certain player for 30 seconds or more, then they should clean up the inactive player from their game instance - a timeout.

A Keep Alive packet may also be used to send score updates to other players. When a local score change occurs, the player may inform other players about the change so they can update their scorecards as well. As mentioned before, the score should be offset by 32768 to deal with negative scoring. Note that players should not respond to the score update with their own scores.

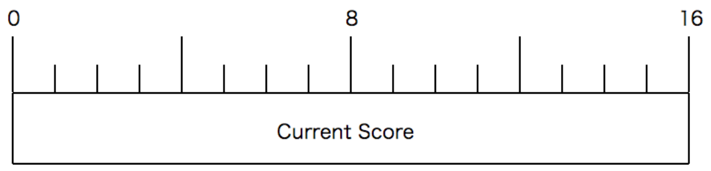


Fig. Keep alive payload

Keep Alive

Type: 0x40
Acknowledgement: None
Payload:

- *Score:* 2 bytes unsigned integer offset by 32768

Leaving a Game

When a player wishes to leave a game, he may use the Leave message. He must stop transmitting packets immediately thereafter. Following such a request, other players must remove the player wishing to leave from their game instances and ignore any further packets from this player, barring a Sync Request. In case the Leave request is dropped, the player will time out within 30 seconds.

Leave

Type: 0x10
Acknowledgement: None
Payload: None

Messages with Reliable Communication

For the game to work correctly, some critical information is required. For instance, a proposed kill must resolve and scores should be settled reliably. Thus, in addition to the unreliable data stream, the protocol also provided for a reliable data stream specifically for game critical communication. The protocol header must explicitly mark a packet as either reliable or unreliable. A reliable packet must be acknowledged by the receiving party, and until that happens, the sending party must retransmit the reliable packet periodically.

Note*: The following packets must set the reliable flag to TRUE in the packet header*

Kill Request

Once a player determines that he has shot a player, he must send a Kill Request using the reliable field in the packet header. Until the pending kill request has been acknowledged, the shooter must retry sending the kill request periodically forever. Additionally, the shooter is not allowed to send another Kill Request to the same target until the pending request has been resolved. After an acknowledgement is received, the shooter may mark this request as sent and remove it from its retransmission queue. Note that the scores should not be modified at this point.

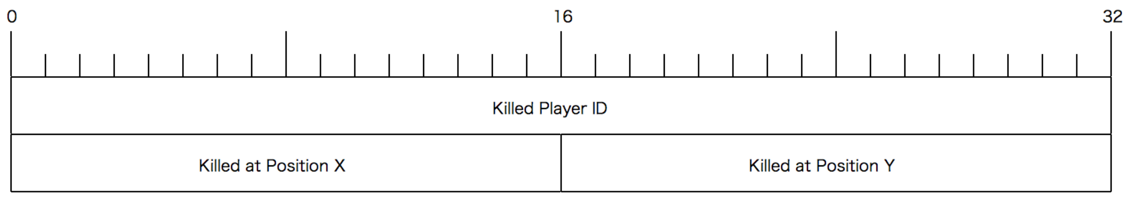


Fig. Kill Request Payload

Type: 0x30
Acknowledgement: Sequence Ack
Payload:

- *Player ID:* 4 bytes containing the target player ID
- *Location X*: 2 bytes for the X coordinate of the proposed kill location
- *Location Y*: 2 bytes for the Y coordinate of the proposed kill location

Kill Accepted

Upon receiving a Kill Request and having acknowledged it, a target must then proceed to evaluate whether the request resulted in a successful kill. If so, it must send a Kill Accept packet reliably and wait for the shooter to acknowledge. Again, this packet must be retransmitted periodically until an acknowledgement is received from the shooter.

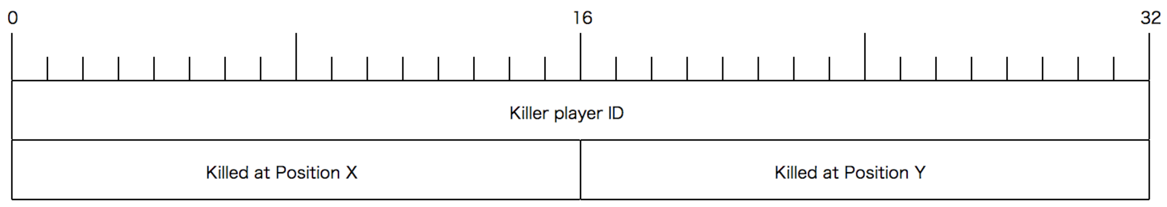


Fig. Payload of Kill Granted

Type: 0x31
Acknowledgement: None
Payload:

- *Player ID:* 4 bytes containing the shooter player ID
- *Location X*: 2 bytes for the X coordinate of the kill location
- *Location Y*: 2 bytes for the Y coordinate of the kill location

Kill Denied

This packet is the same as Kill Accept, but is used to refuse a kill proposed by a shooter.

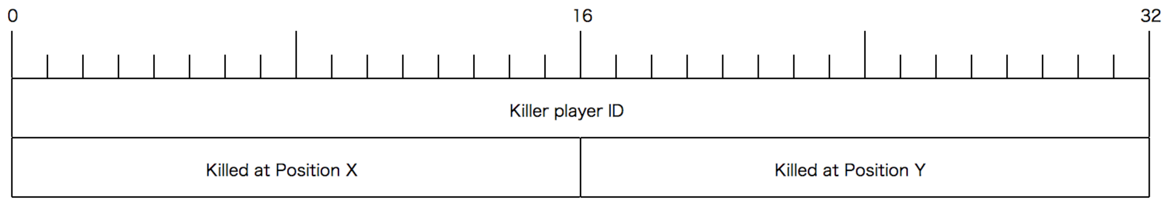


Fig. Payload of Kill Denied

Type: 0x32
Acknowledgement: None
Payload: Same as Kill Granted

Based on the Kill Accept packet, a shooter can then update its local score and acknowledge the packet. This acknowledgement will enable the target player to update its local score as well.

Assumptions on Game Mechanics

This section describes the assumptions made to design this protocol and determine what sufficient consistency means for the purposes of this assignment.

Shooting

A player should locally display a fired bullet, and calculate the impact of the bullet based on the intersection of the bullet's location and other players' location on the map. The player may then send a request to the target player along with the proposed kill coordinates to confirm the kill. A bullet hits a player if the player's most recent local position matches the proposed location of the kill.
Shooting updates are treated as critical data and are thus transmitted reliably to ensure consistent game state.

Movement

Position updates are treated as non-critical data. That is, no acknowledgements are required for such updates. To maintain the real-time nature of this game, if such packets are lost, players may simply rely on the next series of packets to refresh game state. This may cause some apparent player teleportation in bad networks, in exchange of a more responsive behavior.

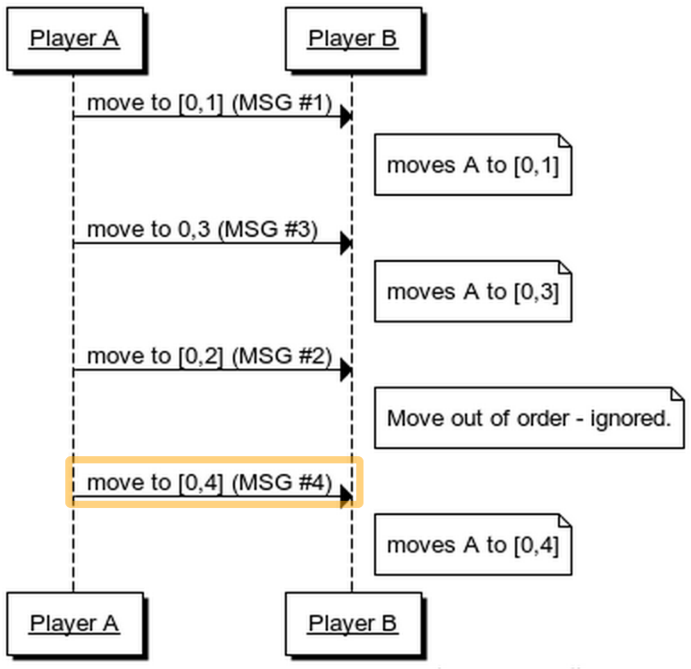


Fig. Sequence of out of order move processing (teleport behavior)

Scoring

Scoring is important, but gameplay is not severely impacted if there is a temporary inconsistency in player scores. Eventually, score updates will get through and converge players to the correct scores.

Thus, scoring is not classified as critical to the gameplay and therefore is not sent using the reliable channel.

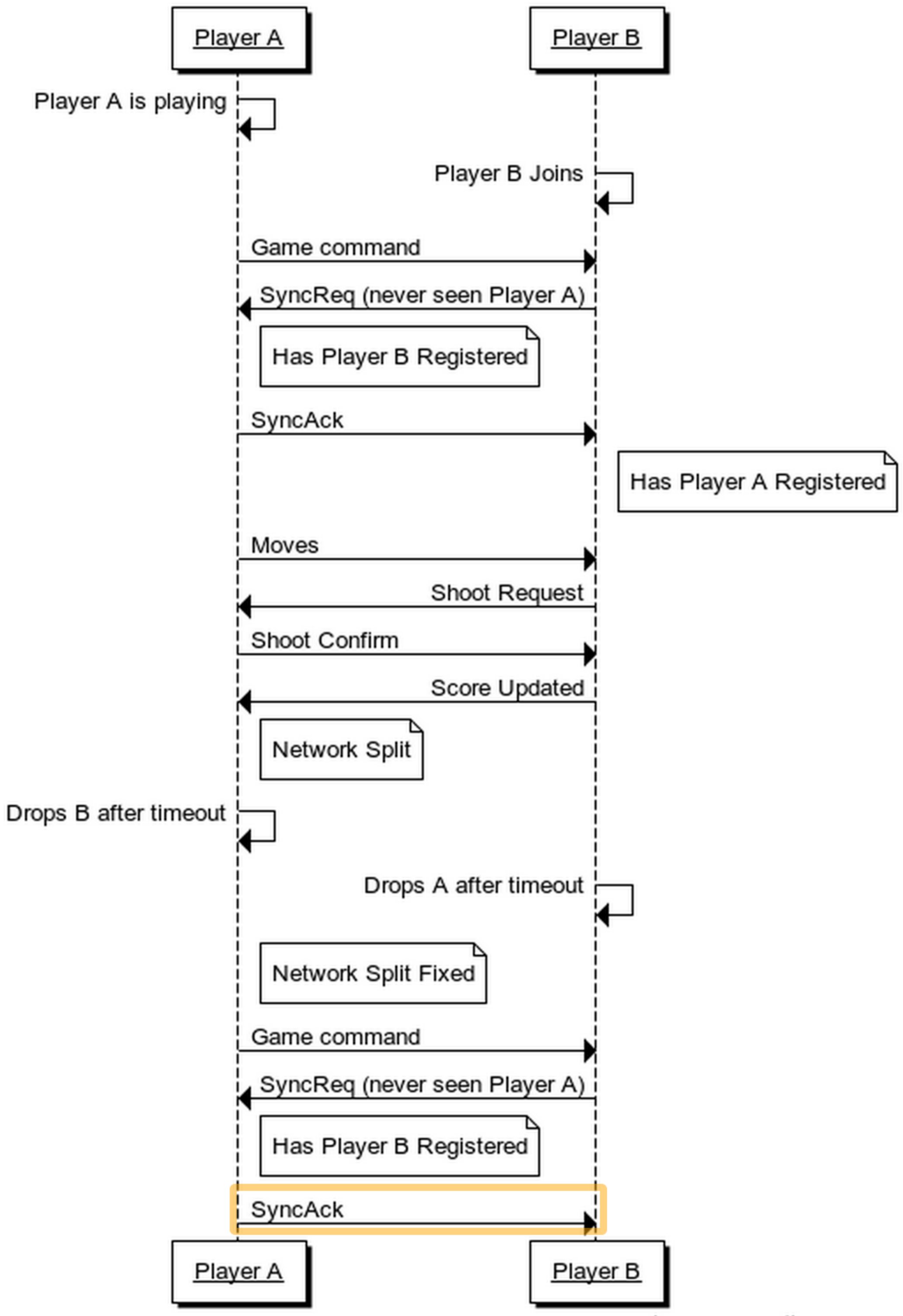
Other Assumptions

1. Players are sole owners of their local game state and are playing fair. This enables other players to reconcile differences in state by accepting newly sent state.
2. There is no attempt to inject malicious data into the game by any player.
3. No discovery mechanism is required as the players known which UDP port to connect to in order to join the game.

Appendix A- Selected Test Scenarios

Case	Description
Player joins existing game	Check a Sync Request is sent - new player is registered in the game
Player collision tests	Send Move to a player with a colliding position - clients should resolve the conflict and move to non-conflicting positions
Bullet hit test	Send Kill Request for valid, invalid positions, try to send multiple kill requests consecutively at the same position and time - only first kill is processed, rest are denied
Score synchronization	Check players send a score update packet as soon as score changes.
Packet delay	Insert delay in packets - check reliable and unreliable packets are processed correctly
Out of order packet delivery	Insert randomization of packet sequence numbers - check reliable and unreliable packets are processed correctly
Corrupted packets	Insert corruption - check the packets fail to reach app logic
Duplicate messages	Insert duplication - check duplicate messages are ignored
Network partition	Block connectivity for minutes on both clients - check sync loop is retriggered and game resumes
Firewall misconfiguration	Configure a client to drop any outgoing packets - one client should play alone with Sync Requests not being answered, both players come back online once firewall is fixed

Appendix B - Network Error Recovery and Synchronization


Player A
Player B
Player A is playing
Player B Joins
Game command
SyncReq (never seen Player A)
Has Player B Registered
SyncAck
Has Player A Registered
Moves
Shoot Request
Shoot Confirm
Score Updated
Network Split
Drops B after timeout
Drops A after timeout
Network Split Fixed
Game command
SyncReq (never seen Player A)
Has Player B Registered
SyncAck
Player A
Player B