Mazewar Questions

Sining Ma SUNedID: sma87

1. Evaluate the portion of your design that deals with starting, maintaining, and exiting a game - what are its strengths and weaknesses?

   Strengths:

   Each player will detect other players joining the game. Join phase last for 3 seconds, and join message is sent continuously. Once a join message is received, join response message is produced. The possibility of all join messages lost is very low. The shared state can be updated and maintained continuously, and even some keepalive messages are lost, the shared state is eventual consistency. One player exists the game is guaranteed to be detected no matter leave message is lost or not. This is because of keepalive timeout. We are using boost UUID to generate a global identifier for each user. It is difficult to see identical user ids.

   Weakness:

   Every message is sent to the entire multicast group. There are a lot of messages that need to receive and parse. The performance of this program is weak if there are a lot of players willing to join this game.

   If leave message is lost, the player who is still in the game needs to wait till keepalive timeout happens. This takes 10 seconds, and during this time player in the game still think the left player is in the game and can see him and shoot him. This is a local consistency.

2. Evaluate your design with respect to its performance on its current platform (i.e. a small LAN linked by ethernet). How does it scale for an increased number of players? What if it is played across a WAN? Or if played on a network with different capacities?

   On the current platform, the performance of this design is okay. This program has only one thread, and it only allows 8 players to play at the same time. But all the messages are multicast to a single group that contains all these 8 players. If there are increased numbers of players for this game, one thread needs to handle all the messages from all players. The design will become the bottleneck.

   If this game is played access a WAN or a network with different capacities, network receiving packets system call may block for some time in order to receive all the bytes of one packet. Therefore, any one user who takes longer than usual time to receive packets will drag the system performance down.

   If we want to scale many players or access a WAN, this program must be changed to be a multi-thread program. For each player, there is a receiving message thread to receive

all messages for this specific player. There is another one sending thread to send all messages. The player who wants to send messages just put the sending messages in the same message queue. Then sending messages thread will send messages from message queue through the network. This improves the performance.

In our design, all the messages are sent to the same multicast group. This means all users are supposed to receive all messages that sent by himself and some messages that not supposed to send to him. The program receives and handles many trash messages. A better solution of the design is to optimize the multicast group capability. Some players are in one group, and others are in another group. The messages are sent to the small group that supposed to include the target player.

3. Evaluate your design for consistency. What local or global inconsistencies can occur? How are they dealt with?

   If hit response message is lost for some period of time, the score will be inconsistent. The player who sends hit response message gains point, but the player who is supposed to receive hit response message does not deduct his point. This is a global inconsistency. If hit response message is lost without any method to handle packet lost, this global consistency cannot be solved. In our design, there is a hit phase and one player will stay in this phase for 2 seconds and contiguously send hit messages to shooter player. He will not be out of hit phase until he receive hit response message he wants. This victim player is supposed to receive at lease one hit response message. If he cannot receive any hit response message, the global consistency cannot maintain. This approach reduces the possibility of losing hit response messages. These ideas are also leveraged for join and join response messages.

   Another inconsistency is that if some keepalive messages are lost shared state should be inconsistent. But in the design, this is handled by sending keepalive messages once 200ms. The benefit of this is that even if some keepalive messages are lost in case network problem. The shared state is able to be in eventual consistency, and all game application logic on top of the shared state is maintained.

   The local consistency is that if leave message is lost, all other players needs 10 seconds to timeout keepalive messages, and figure out one player has left the game. We can decrease keepalive timeout to a smaller amount of time like 3 seconds. Another approach is adding a check message to our design. Check message is used to check if the other player is still in the network once we cannot receive any keepalive messages. If check response message can be received, this indicates that the other player is still in the game but not able to send keepalive messages now. But no check response message timtout should tell confirm left or dead of one player. This is faster than waiting for keepalive timeout.

   Another local consistency is related to the missile. When I manage my missile, I can find that the missile hits the wall and clear my missile state. But if there is another player who stands right next to the wall and on the moving way of my missile, this player will claim to be hit by this missile. Hit message receives after the player thinks

the missile hits the wall. This case is handled in the implementation. When one player receives a hit message, the player will send a hit response message. But updating missile related variables should be careful. Only update missile state variable when missile exists.

4. Evaluate your design for security. What happens if there are malicious users?

   I am using boost library UUID generator, so the possibility of duplicate UUID is very low. But all users can receive all sending messages. One malicious user A can pretend to be another user B, because this user A knows B's UUID. He can extract A's UUID from the packet and send messages with A's UUID to distract the shared state. This design is multicast group vulnerable. But out of this multicast group, no player can receive any packet.

   The design is multicast group trustworthy. This means that we trust all users in the same group should not hack network packets, and always try their best to maintain a correct shared state.

   The second security issue is that all the messages are sent without encryption. If there is some user who hacks the sending packet, he is able to pretend the UUID, and sending some invalid shared state data. This will break the game. Therefore, we need to encrypt at least UUID in the message packet.