Mingjian Liu Maliu2@stanford.edu

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

The starting stage of the game is: the game broadcast a request to acquire all states of other player's information, every other game received the request will respond with its own states. The joined game wait for a short period, collect all information from other, and generate its own local view of global state.
Strength: Giving a short period time of waiting, the late-join game can collect all respond it can get.
Weakness: This also result in delay to start the game.

In the maintaining stage, we introduced two mechanisms, one is dead reckoning, the other is re-transmitting with packet ID. The first one's purpose is to save received events, and process them in batch at next time slot. The second make sure that even packet may reached out of order, the unique ID of events in one player can still be sorted. To resolve conflict, if in one timeslot, two actions from different player are stepping in the same area, only one is permitted.
Strength: It gives game some graceful time for packet delivery, while the event order identifier is still there. Conflict is resolved by the game's ID. So packet loss issue is addressed, and consistency is provided for each time slot.
Weakness: It's not easy to set how long the graceful time period is. If it's too long, then the player need some time to see the action is actually performed, which gives bad game experience. If it's too short, it's possible some packets are still re-sending. So temporary inconsistency might occur. What's more, the conflict resolve by ID is not fair.

The exiting is just silently quit the game. If it doesn't respond to other's heartbeat for a whole time slot, then others would think it's already dead.
Strength: This can reduce traffic. And also make implementation easier.
Weakness: If the game just loses network connection for a time slot, others would think it's already dead. Temporary inconsistency occurs.

By the way, ID might be in conflict is also a weakness. However, it can be resolved if the late-join player saw it in packet State Inquire Respond and then change it's own ID to born.

Mingjian Liu Maliu2@stanford.edu

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?

Answer: From the design view, right now the players broadcast all information, but there are eight players at most, and delay is within 10x ms, performance is good with one thread.
If there are a lot more players joined, each of them will received more information, which will impact on performance (more I/O interrupt, more process time). One thread is not enough.
If the game is played in WAN, the RTT will vary a lot. Our design is based on the assumption that packet drop can be recovered by resend, which only take a few milliseconds in LAN, however, this is not true in WAN. If packet delay is more than a time slot, some temporary inconsistency occur. Also, we use IP as the game's ID, for WAN, we might need to use UUID to distinguish players.
If the network has different capacity, the game might be divided into different parts. In each part, games can communicate smoothly, but out of them, the packet loss will increase substantial. The result of it is, when they merged together again, to maintain consistency, one part need to rollback a lot, which makes change abruptly, and downgrade player's game experience.

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

There are three kinds of inconsistency can occur by packet loss.
Two players are stepping in the same space, but the winner's packet is lost. Then some would think the loser stepped in that space.
One player is projecting a missile, and it's lost. Then if someone should be hit didn't receive that packet, it will not reborn.
One player is cloaked but the packet is lost, then that player is still visible to some others.

This is dealt with retransmission with ACK, sending absolute information in each event packet. If some packets is lost, the sender will get notified when timer runs out and resend them. Even some packet is totally lost, the absolute information can correct it.
For example, if one missile hit someone and should be disappeared, but the one get hit just lost the packet, for one timeslot, if the connection is still not covered, then

others will think it quit. If the connection is back, then it will get notified by the missile owner's absolute information and update others.

Evaluate your design for security. What happens if there are malicious users?
In our design, the games always trust other gamers about their 'absolute information', which is made for recover from temporary inconsistency. If someone cheated on it, we have no way detect it. By checking all events received, there is still no way to check someone has cheated or not, since it's not possible to check whether there are still some missing packets.
What will happen:
If one set visible to see all other cloaked rats or other's missile, then no one can detect it because it's cheating only in its local view.
If one player gets tagged doesn't broadcast it, the others will think the missile still exist.
If one set to 'fly' to some other place, others have no way to tell whether it's cheating.
If one use other's ID to send some fake event, then the global view is compromised. Even worse, the game might be inconsistent.
If one keeps sending far more packet then the game can process (like a DDOS attack), the game will have destroyed. Or just register for 7 IDs, then others can't join the game.

To prevent malicious outside, we should think of encryption. So others out of the game can't see what's happening inside.
To prevent gamer cheating on game, there might be three ways we can think of. First, add error detection and correction mechanism. Second, introduce a master who decides what's happened totally, then there is a single point of failure and the 'distributed' attribute. Last, introduce some consensus mechanism, like Paxos, let everyone vote the global decision. But it will compromise the performance and add much more complexity.