# CS244B Distributed system

# Homework 1 – Mazewar

Protocol design and specification

Suj2@stanford.edu

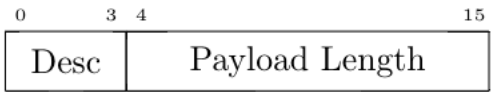
Maliu2@stanford.edu

## Protocol Definition

| Descriptor | Description |
|---|---|
| Heartbeat | Used to sync up with other players. Every node should multicast it periodically. |
| Heartbeat ACK | The response to Heartbeat. Each player should reply if it receives a heartbeat from other players. |
| Event | This message is sent once the local player is about to take an action: movement, direction change, missile projection, cloak etc. Along with the event information, it also provides the sender's local state for the sake of eliminating cumulative inconsistency. |
| Event ACK | The response to an Event. This descriptor only provides the acknowledge. |
| State Inquiry Request | The broadcast to inquire all existing players' states when a new player joined. It contains the new player's ID. |
| State Inquiry Response | Point-to-point response to a State Inquiry Request. It provides not only the replier's current absolute state, but also its uncommitted action(explained in dead-reckoning section), to maintain consistency. |
| State Inquiry ACK | Acknowledgement to State Inquiry Response. |

## Descriptor Header

A 2-byte string uniquely identifying the descriptor in network.




### Descriptor ID:

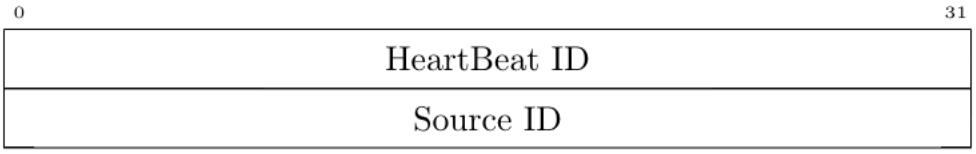
0x0 = Heartbeat
0x1 = Heartbeat ACK
0x2 = Event
0x3 = Event ACK
0x4 = State Inquiry Request
0x5 = State Inquiry Response
0x6 = State Inquiry ACK

### Payload Length (12 bits):

The length of the descriptor immediately after the header. Range from 0 – 4095 bytes.

## Heartbeat (0x0):

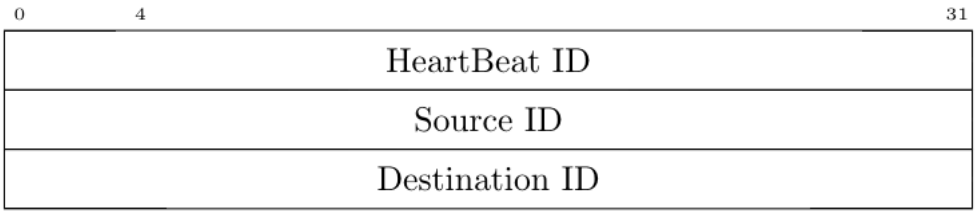
The heartbeat payload contains heartbeat ID, source ID.




**Heartbeat ID** A 32 bits field. Every time a player sends out a new heartbeat, it should be incremented by 1.

**Source ID** A 32 bits field for the ID of source player. Here we choose IP since the game is in LAN. If it's not LAN, UUID might be a better choice.

## Heartbeat ACK (0x1):



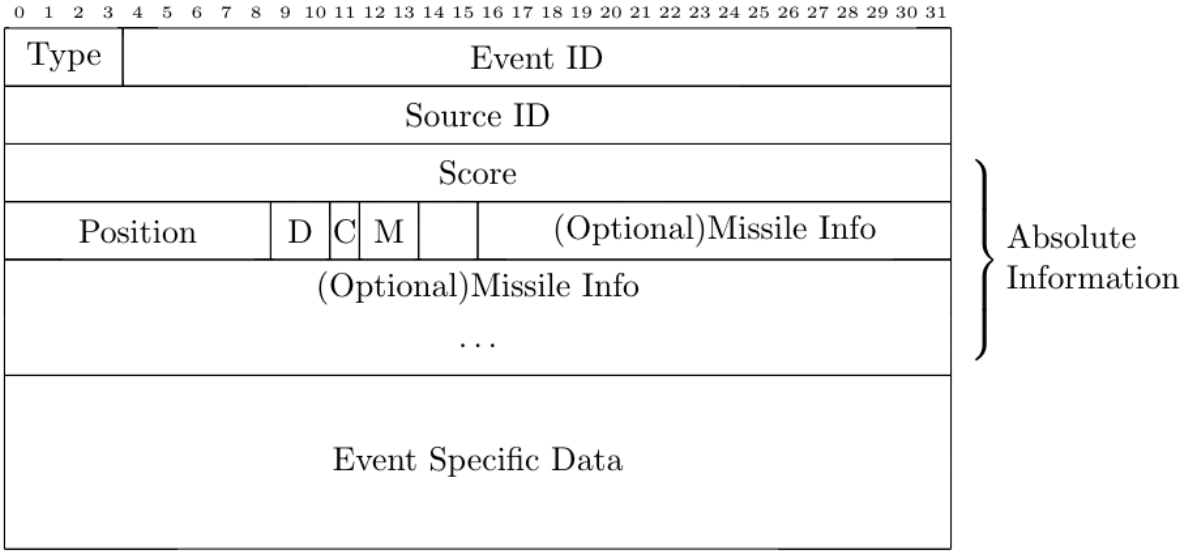

**Heartbeat ID** A 32 bits field. Every time a player sends out a new heartbeat, it should be incremented by 1.

**Source ID** A 32 bits field for the ID of source player. Here we choose IP since the game is in LAN. If it's not LAN, UUID might be a better choice.

**Destination ID** The heartbeat owner ID.

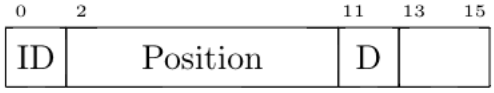
## Event (0x2):

It contains event type, event ID, player ID, the absolution information, and event specific data. Every field are explained below.




| | |
|---|---|
| **Type** | A 4 bits field indicating type of event. It can be born, movement, cloak, missile projection, missile hit, each contains its specific data. |
| **Event ID** | A 28 bits field for incrementally index identifying the sequence of event for each player. It can be used to handle disorder delivery/packet loss. |
| **Source ID** | A 32 bits field for the ID of source player. Here we choose IP since the game is in LAN. If it's not LAN, UUID might be a better choice. |
| **Score** | 32 bits field for the sender's score, can be converted to uint32. |
| **Position** | The maze's size is 32x16 = 2^9. So this field indicates player's absolution position in the maze. |
| **D** | A 2 bits field for the player's direction: 00 = up, 01 = down, 10 = left, 11 = right. |
| **C** | A 1 bit field for the player's cloak state. 0 = uncloak, 1 = cloak. |
| **M** | 2 bits field for the count of player's ongoing missile. It's followed by each missile's information. |

## Missile information:




| | |
|---|---|
| **ID** | A 2 bytes field include the missile ID (0-3, assume there can be at most 4 missile on the fly). |
| **Position** | The absolute position a missile located in the maze. |
| **D** | Flying direction: 00 = up, 01 = down, 10 = left, 11 = right. |

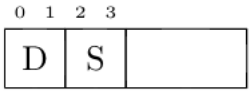
## Event Specific Data:

**Cloak** A 1 bit field, 0 = uncloak -> cloak, 1 = cloak -> uncloak.

**Movement**

It's a 1 byte information. It contains two kinds of event:

1. The player moves. In this case, S = 1 or 2.
2. The game change direction. In this case, D is different from the absolution state. S = 0.



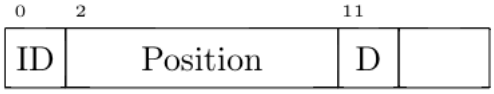

| | |
|---|---|
| ***D*** | Player's direction: 00 = up, 01 = down, 10 = left, 11 = right. |
| ***S*** | Speed: 0 = no move, 1 = move forward, 2 = move backward. |

**Born**

It's a 2 bytes field contains the new position (9 bits) and direction (2 bits).

**Missile Projection**

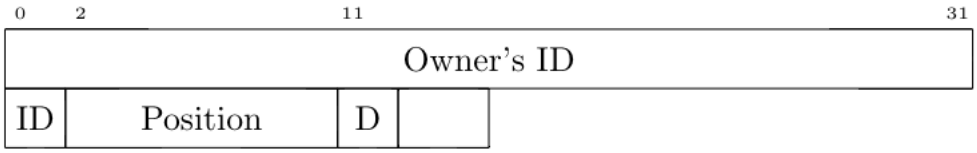
A 2 bytes field includes new missile's information.




| | |
|---|---|
| **ID** | Missile's ID(0-3) |
| **Position** | The same 9 bits absolute position. |
| **D** | The same 2 bits direction. |

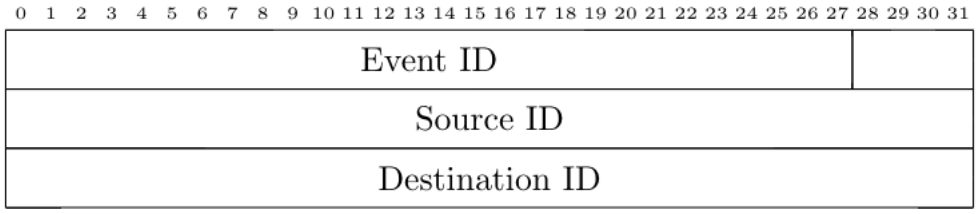
**Missile Hit**

A 6 bytes field includes the missile's owner ID, missile's information.




| | |
|---|---|
| **Owner ID** | 32 bits owner's ID |
| **ID** | 2 bits Missile's ID(0-3) |
| **Position** | The same 9 bits absolute position. |
| **D** | The same 2 bits direction. |

## Event ACK (0x3):



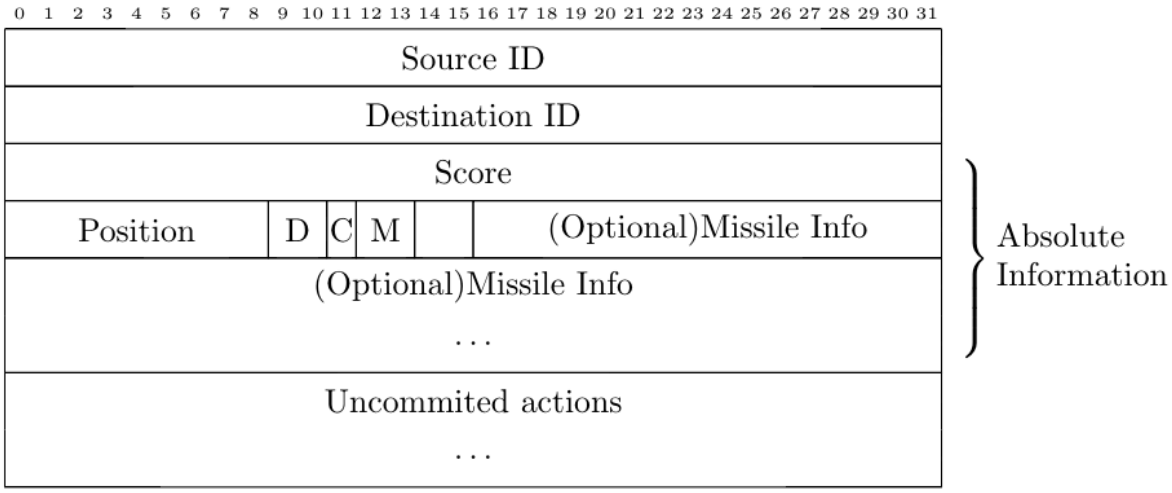

| | |
|---|---|
| **Event ID** | A 28 bits field for incrementally index indentifying the sequence of event for each player. |
| **Source ID** | The sender ID of this message. |
| **Destination ID** | The event's owner's ID. |

## State Inquiry Request (0x4):

It only contains a 32 bits field for Source ID.


## State Inquiry Request (0x5):

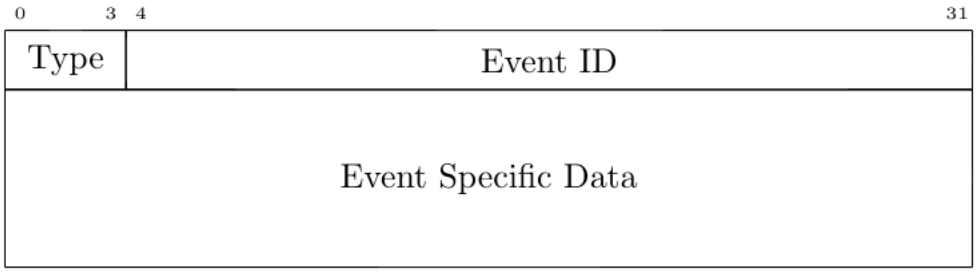
It contains source ID, destination ID, Absolute Information and also uncommitted events.



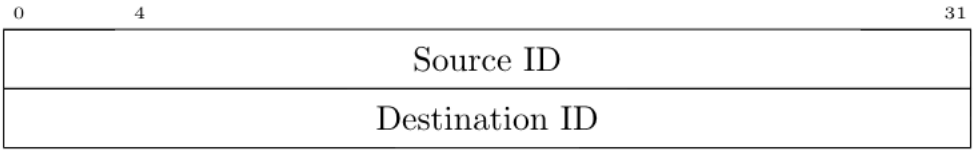

| | |
|---|---|
| **Heartbeat ID** | A 32 bits field. Every time a player send out a new heartbeat, it should be incremented by 1. |
| **Source ID** | ID of the response sender. |
| **Destination ID** | ID received from State Inquiry Request. |
| **Absolute Information** | The absolute information as explained before. |
| **Uncommitted actions** | The sent out but still uncommitted events. |

### Uncommitted actions:

All field are explained in Event payload.

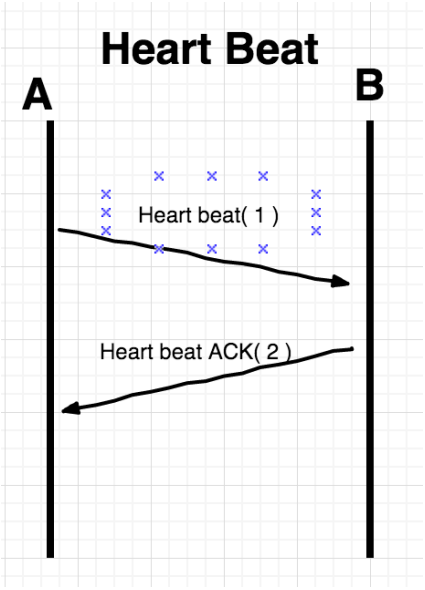



## State Inquiry ACK (0x6):




| | |
|---|---|
| **Source ID** | The sender's ID. |
| **Destination ID** | ID of State Inquire response owner. |

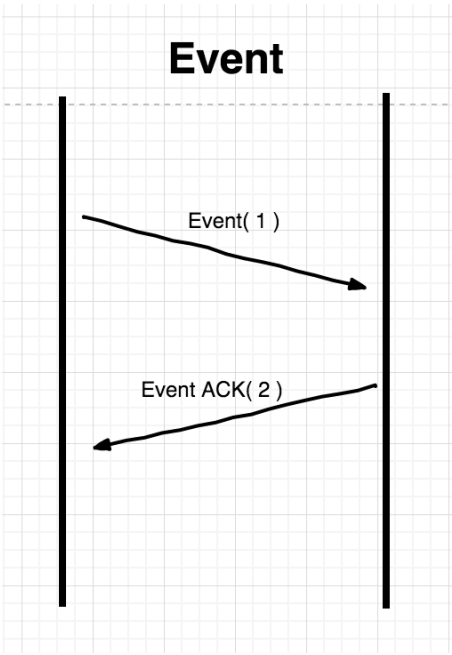
## The sequencing and semantics of the packets




### Heartbeat

Heart beat
A send heart beat message to B to ask if B is still alive.

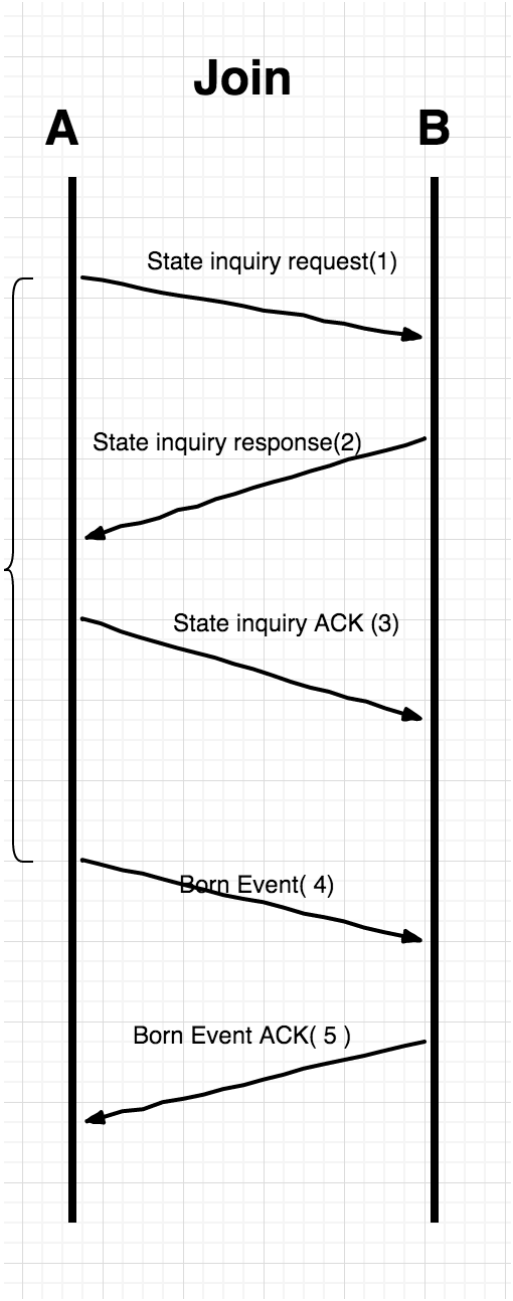
Heart beat ACK
B send ACK message to inform A that B is still alive.




### Event

Event
A send heart event message to instruct B what event have happened.

Event ACK
B send ACK message to inform A that B is have received message.




## Locate and join game

1. State Inquiry Request
When a new player A joins the game. A will multicast "State Inquiry Request" to everyone in the system to inform every player its existence and ask for others' state.

2. State Inquiry Response
Once player B receive state inquiry request from late joined player A. B will send state inquiry response back to A to notify the current state of B to A.

3. State inquiry ACK
After received State Inquiry from B, the late joined player A will send State inquiry ACK to B in order to inform B that A have already receive B's state.

4. Born Event
After some fixed interval of time, A will send a born event and share its state to everyone.

5. Born Event ACK
Other players send ACK to A and confirm born event.

## Exit game

The player just silently exits the game. With no ACK response on heartbeat for a number of times (5 we set here), other nodes will regard it has exited.

### Packet Loss/Order Handle

The Mazewar uses UDP multicast/unicast to send message. UDP doesn't provide either reliability or ordering. Therefore, it's possible packets are not delivered or they arrived in no order. Here we explain what mechanism we use to tackle these issues.

**Packet Loss:**

We use acknowledge to arrive reliability. All the ***Event***, ***Heartbeat***, and ***State Inquiry Response*** requires a relevant acknowledge(ACK) packet. The sender will keep re-sending packet until it gets the corresponding ACK packet.

Here we referenced TCP and use its "Exponential Backoff Algorithm" (https://tools.ietf.org/html/rfc2988). It triggers a timer=T once a packet is sent, if no ACK received when time out, it will re-send the packet and trigger a timer=Min (2 x T, Maxtime), where Maxtime is the upper bound for timer.

If no ACK after 5 times' re-send, assume the packet is lost and connection is broken. Each packet triggers its own mechanism to handle with it.

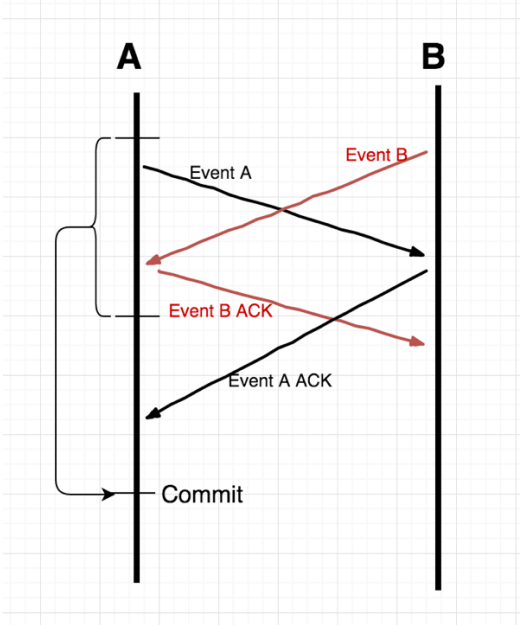
**Order**

Packet will arrive in no order with UDP. We introduce a ID field for each Heartbeat and event. If the previous packet arrives after the its later packet has already been processed, the receiver just simply throws it away. This may lead to some inconsistency, which will be addressed later in dead-reckoning.

## Timings of protocol events

### Dead-reckoning algorithm

We use dead-reckoning to handle events in batch. It's suitable for distributed system since this mechanism doesn't require a global clock. The example is descripted below.




We split local time into slots. Each slot is 200ms. For each slot, it will process the event happened one slot before (400ms to 200ms before). All disorder packet can be sorted by event ID and all conflict in the same time slot can be resolved by some fixed method. For example, if both A and B send event to move to place X, and A ID > B ID, then A move.
By this, we can achieve loosely couple, less network communication, less network delay requirement, provide all player with "sufficient consistency" without having some tightly synchronization.

## Connection lost

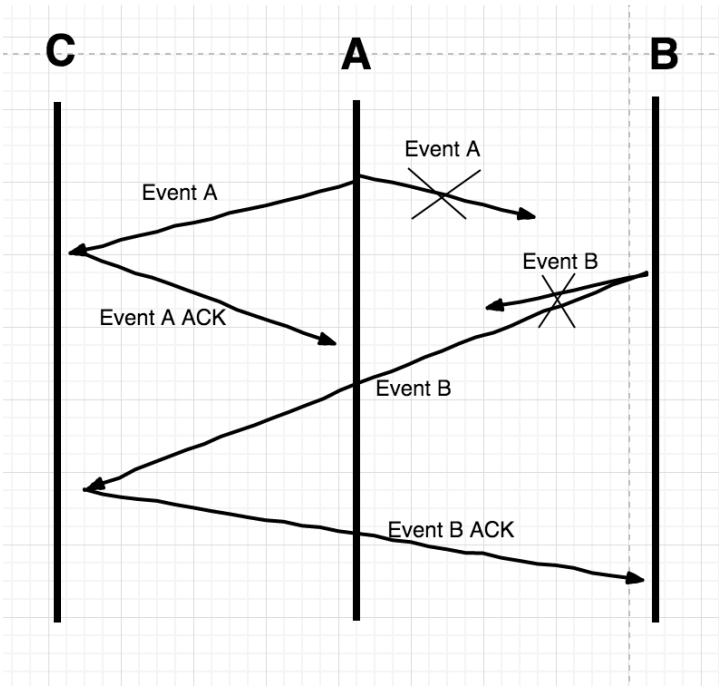
Since the Mazewar players are in the same LAN, for simplicity, we made two assumptions:

1. There won't be Byzantine condition, meaning there can't be two or more parallel ongoing game. For example, players are A, B, C, D. It's won't happen like A and B can communicate each other, so does C and D. But A & B can't reach neither C nor D, so that A & B have totally different shared state with C & D.
   In short, if player A totally had lost one connection to C, then it should also have lost all other players. Therefore, only one shared state is present, even with some degree of inconsistency.

2. The expectation of transmission delay is relative low compare to WAN, like 10x~100x milliseconds. So the inconsistency will most likely be corrected after short period of time.

By the assumption, there are two cases of connection lost:

1. Player A temporary lost connection to player B. But A can still reach other players. In this case, assume A multicast an event "A move to place X", while B also multicast another event "B move to place X". The diagram below shows this scenario.




Event “A move to place X” and “B move to place X” is in conflict. In A’s local state, it can’t see B’s event, so it simply moves A to X. So does B. This led to inconsistency.
Meanwhile, all other nodes like C are able to receive both events. Since they are in conflict, they will decide by some logic to resolve it, like if A’s ID > B’s ID, then let A move. As a result, the inconsistency will only exist in either A or B’s local state. (Fortunately, one of them still do the right thing).
This inconsistency will be resolved when connection has been recovered and either event is delivered or a new event with absolute state is sent.

2. Player A lost all connections to other players.
   If the connection can’t be recovered in short period of time, then it will also lose all heartbeat with others. In this case, player A will think itself quit the game so that its local state is discard. All other players are still sharing the same state of A until they think A has been quit. Therefore, no inconsistency occurs.
   If the connection is recovered shortly, the inconsistency will arise when A made some action locally, but these action can’t be send to all others. In this case, when connection is recovered, all event sent from A to others or from others to A will be delivered with their up-to-date absolute information. So the conflict can be later resolved locally in each player’s state.