Software Engineering of Distributed Systems
Kungliga Tekniska högskolan

Problem and Solution of Goldy

Seminar Paper of Distributed Systems

Shanbo Li
2007-10-16

Problem and Solution of Goldy

Shanbo Li **840810-A478** shanboli@Gmail.com

This is a paper for a seminar of distributed systems. There are four parts in this paper. Firstly this paper shows a distributed game named Goldy and implemented with Erlang. Then it will describe the problem of Goldy. After that it will give two solutions of the problem. At last the outlines of implements will be illustrated.

1. Goldy: a distributed game

Goldy is a distributed game which is implemented by Erlang with a very simple structure. Each player in Goldy will start the game and announce her position and also announce the positions of a set of gold nuggets. Each player can then start to move, trying to collect as many nuggets as possible. There are three types of spot on the user's window. And spot with red shows the operator's current position, greed spots are other players' position, yellow spots mean that they are gold nuggets. *Figure 1* shows a screenshot of Goldy.

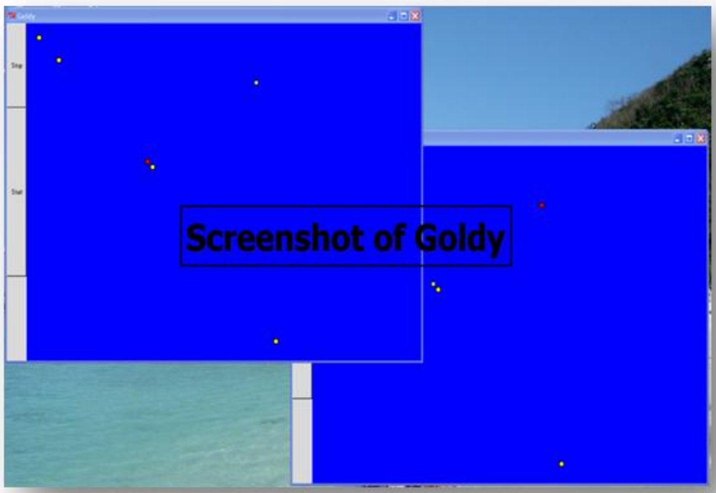
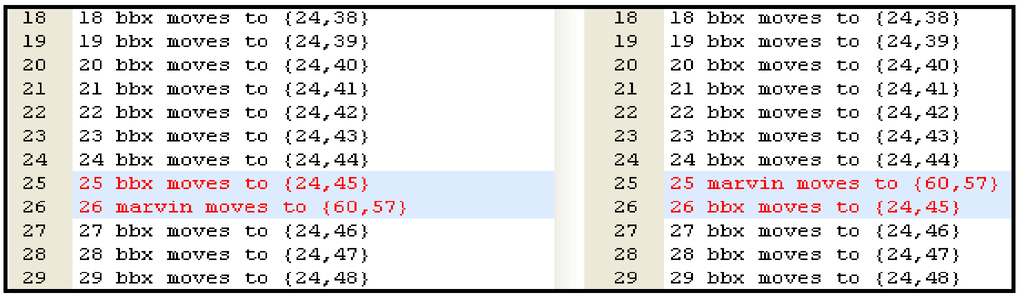


Figure 1 screenshot of Goldy

2. The problem of Goldy

It is fun to collect the gold nuggets. But some problem occurred while we played the game. What you see is not what you really got. Or one play can see that he eats the gold, but another guy announced it is himself who really gets the gold nugget. It seems that at a certain moment different players have different screens. This scenario should not happen on a distributed game.

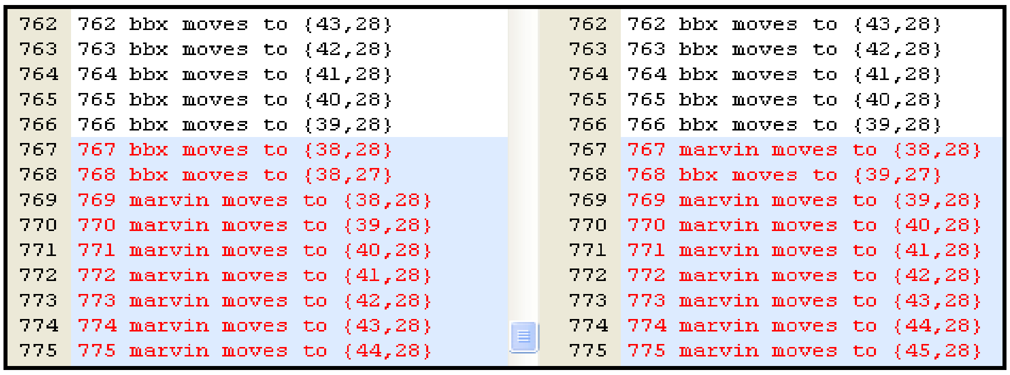
I think the latency between peers may lead to a message delay, so that different play gets messages by different order. To help me improve this and know what happened, I added a file write function to Goldy. While players enjoying the game, the recorder writes every move of every player into a local file named of the local player name. And the records in the files sorted by time order and begin with a serial number. The time order obviously is the time when the play received the message either from him or other players. *Figure 2* shows a part of records compare.



```
18 bbx moves to {24,38}          18 bbx moves to {24,38}
19 bbx moves to {24,39}          19 bbx moves to {24,39}
20 bbx moves to {24,40}          20 bbx moves to {24,40}
21 bbx moves to {24,41}          21 bbx moves to {24,41}
22 bbx moves to {24,42}          22 bbx moves to {24,42}
23 bbx moves to {24,43}          23 bbx moves to {24,43}
24 bbx moves to {24,44}          24 bbx moves to {24,44}
25 bbx moves to {24,45}          25 marvin moves to {60,57}
26 marvin moves to {60,57}       26 bbx moves to {24,45}
27 bbx moves to {24,46}          27 bbx moves to {24,46}
28 bbx moves to {24,47}          28 bbx moves to {24,47}
29 bbx moves to {24,48}          29 bbx moves to {24,48}
```

Figure 2 records compare (piece one)

The assumption is right. Delay of messages makes players see different order movement. Red font shows the different parts of records. At the 25th step, the record on the left shows player bbx moves to {24, 45} before player marvin moves to {60, 57}, however the record on the right shows player marvin moves to {60, 57} first. It may be ok when the players are not next to each other. But if the players are nearby and want to move the same place, and messages transfer happen to be delayed at that moment, what will happen? Unfortunately this happened when we played the game. On the *Figure 3* you can see what happened and how terrible it was.



```
762 bbx moves to {43,28}         762 bbx moves to {43,28}
763 bbx moves to {42,28}         763 bbx moves to {42,28}
764 bbx moves to {41,28}         764 bbx moves to {41,28}
765 bbx moves to {40,28}         765 bbx moves to {40,28}
766 bbx moves to {39,28}         766 bbx moves to {39,28}
767 bbx moves to {38,28}         767 marvin moves to {38,28}
768 bbx moves to {38,27}         768 bbx moves to {39,27}
769 marvin moves to {38,28}      769 marvin moves to {39,28}
770 marvin moves to {39,28}      770 marvin moves to {40,28}
771 marvin moves to {40,28}      771 marvin moves to {41,28}
772 marvin moves to {41,28}      772 marvin moves to {42,28}
773 marvin moves to {42,28}      773 marvin moves to {43,28}
774 marvin moves to {43,28}      774 marvin moves to {44,28}
775 marvin moves to {44,28}      775 marvin moves to {45,28}
```

Figure 3 records compare (piece two)

At 767th step, records show that player bbx and player marvin moves to {38, 28} together. And after that the movements of on the two peers are totally different. What a terrible scenario!

After compared the records and analyze what happened. I confirm that ***the problem of Goldy is that message transfer delay can make different peers receive the messages by different order and players see different screens. Moreover when players want to move to the same place, and delay occurred, one player's follow movements will be totally different on two peers.*** Next part of this article will discuss two ways to solve the problem.

3. The solutions of Goldy

To solve the problem illustrated above, I figure out two methods. One solution is using a **Center Server** to handle the global state of players. And the other solution is involving **Totally Ordered Logical Clocks** and a player makes a move by request to others and gets all acknowledgements back.

3.1 Center Server solution

As most online game did, center server can be used to handle the global state of players. At initial of the game, players have to connect with a center server and announce their locations as well as gold nuggets' locations.

When players want to make a move, he must send a request to the center server, and the center will check if the target is legal that means that no other guy stand on the target position. If it is ok to move there, the center server will tag player older position as an available place and store the player in the new position, then give an acknowledgement to the player. After that the center server will broadcast the new global state to all players include the just moving guy. After players received a new global state he will update the GUI and acknowledgement to the center server. The acknowledgement can make sure no packages will lost. A timeout function is also needed, so that if any problem occurred during message transfer, the message will be resend.

The center server will update its global state by the order it received the messages. So the terrible scenario on *Figure 3* will never happen.

The weakness of this solution is that as the player increases, the center server needs to handle more messages. And the speed to acknowledge each player may fall down.

3.2 Totally Ordered Logical Clocks solution

The problem of Goldy can also be solved by total ordered logical clocks. Lamport timestamps and priority will be used to make every move of every player have a total order. So there will not be the any problem of the order of movement. Like center server solution, before a player makes a move he has to send require to all others, and get all acknowledgements back. During wait time, if he received requirement of the same target location he will compare the Lamport timestamp with the play who sent the requirement. And the one whose timestamp is earlier will go first. If they have the same timestamp the will compare with the priority. And the one with a higher priority will go first.

This solution also has disadvantage. If two players always have same timestamp and always want to move to the same place then the guy with lower priority will be very unfortunately to sit there and look the other guy moving.

4. The implements of solutions

The implements will figure the outlines of two solutions.

4.1 The Implement of Center Server solution

On the center server the state will be saved as a list of objects where each object is represented by the structure **{Type, Id, Pos}**. Center Server initials with a state which stored all players' starting positions and gold nuggets' positions. And it will send the list to all players, until all players acknowledge that the received the state list game starts.

At the center server a function named **run_on_server** will do loop to receive messages and broadcast new messages. At the player' computer that is the client will send a request to the server with a format of **{player, Id, Dir}** by a function named **loop** and wait for the acknowledgement**.** Center server will receive the message and calculate the new position and use a function named **update** to search the state list. If the new position is available the function will update the state list and send **{got, Dir}** to the player. After that, server will broadcast the latest global state to all players. Then server starts a thread to wait for the acknowledgement. If any players' time is out it will send the latest state to the player again. The broadcast can be implemented by a module named **multicast**. If the new position is unavailable server will send **{no}** to the player who sent a requirement.

Both server and client will keep the state list. But the state list on the server is used for check or update state. And the state list on the client is just like a cache to help client to calculate if a move is legal or not. But it can be only updated by the message from server.

4.2 The implement of Totally Ordered Logical Clocks solution