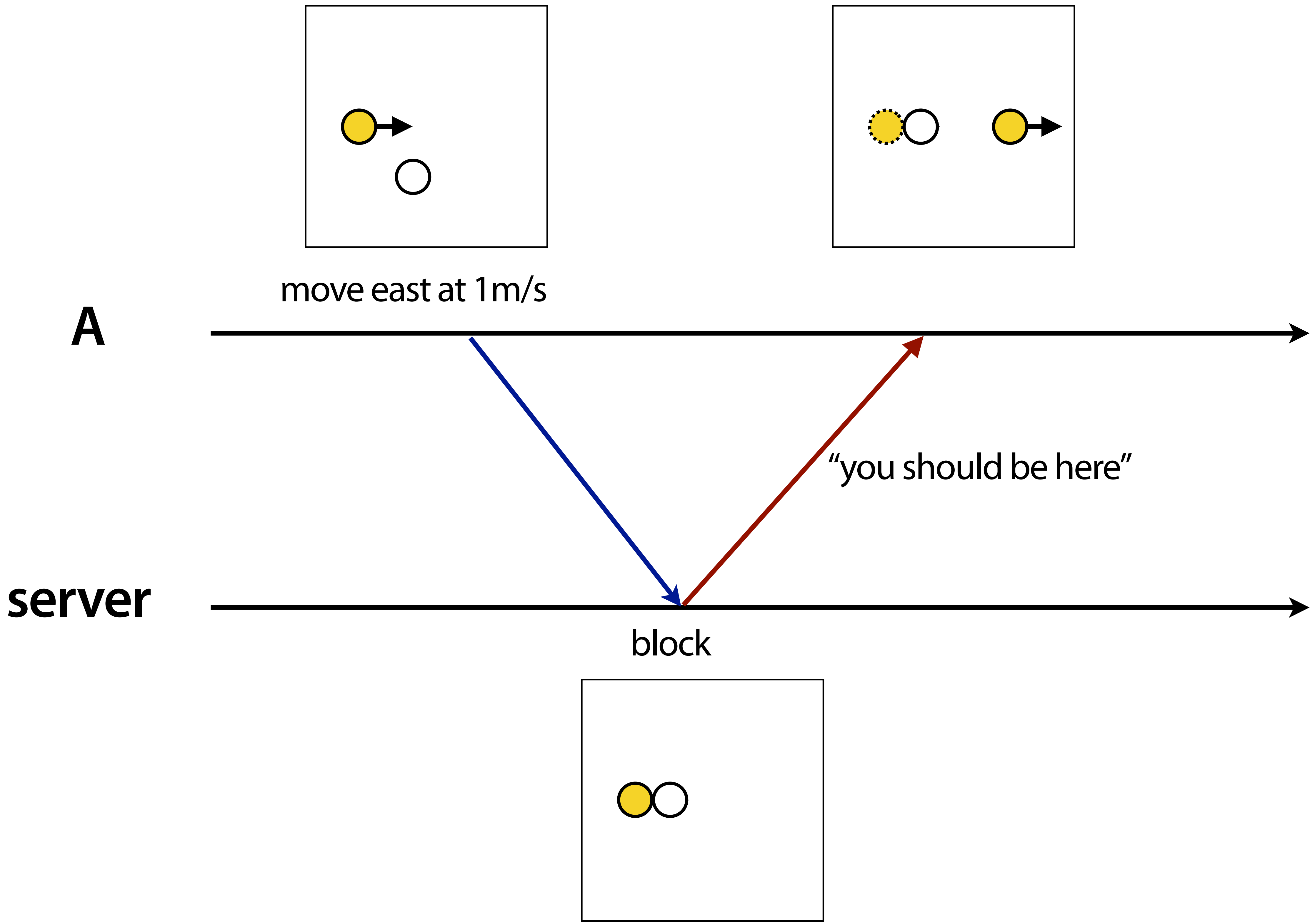
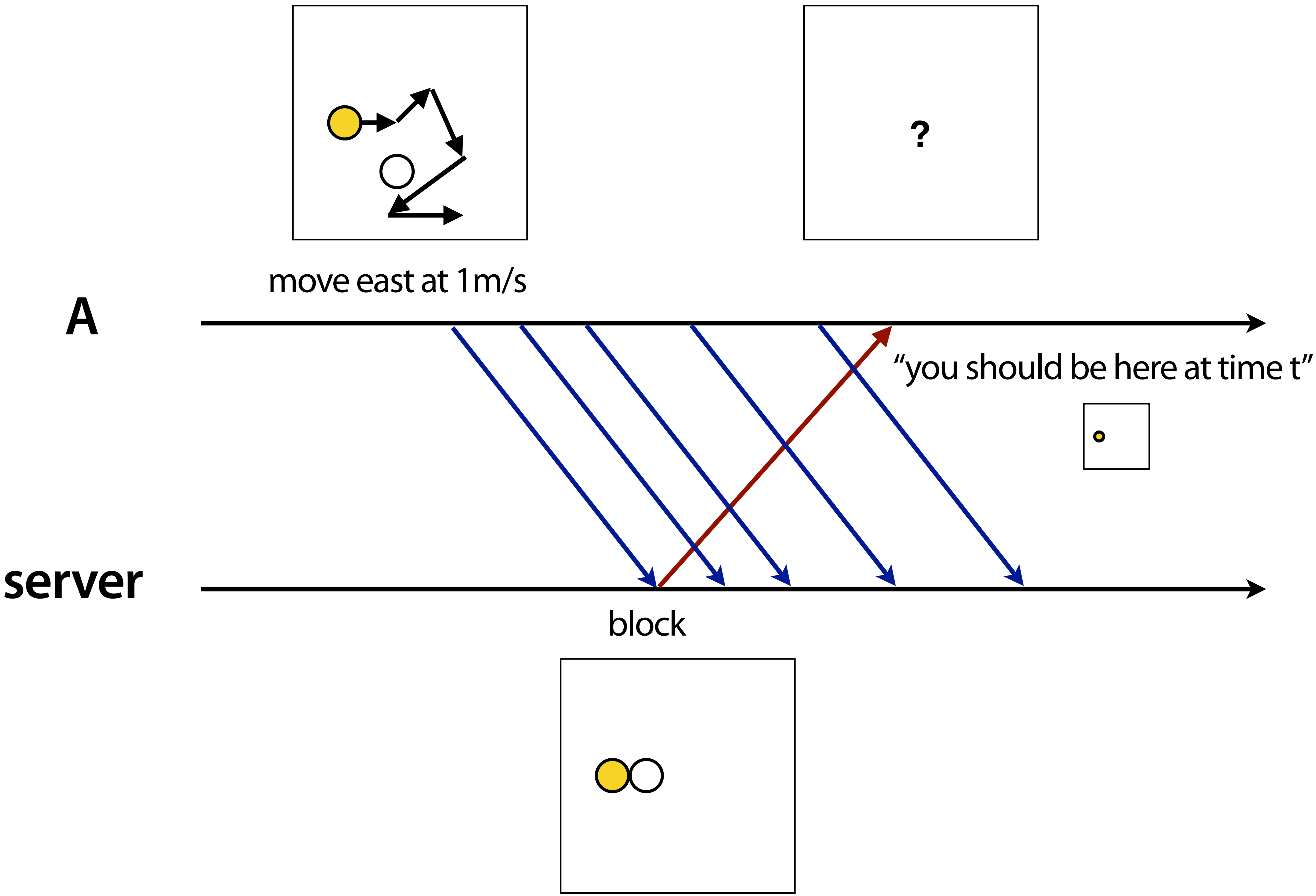
Lecture 3

Prediction and Compensation

Let's focus on inconsistency on players' position.

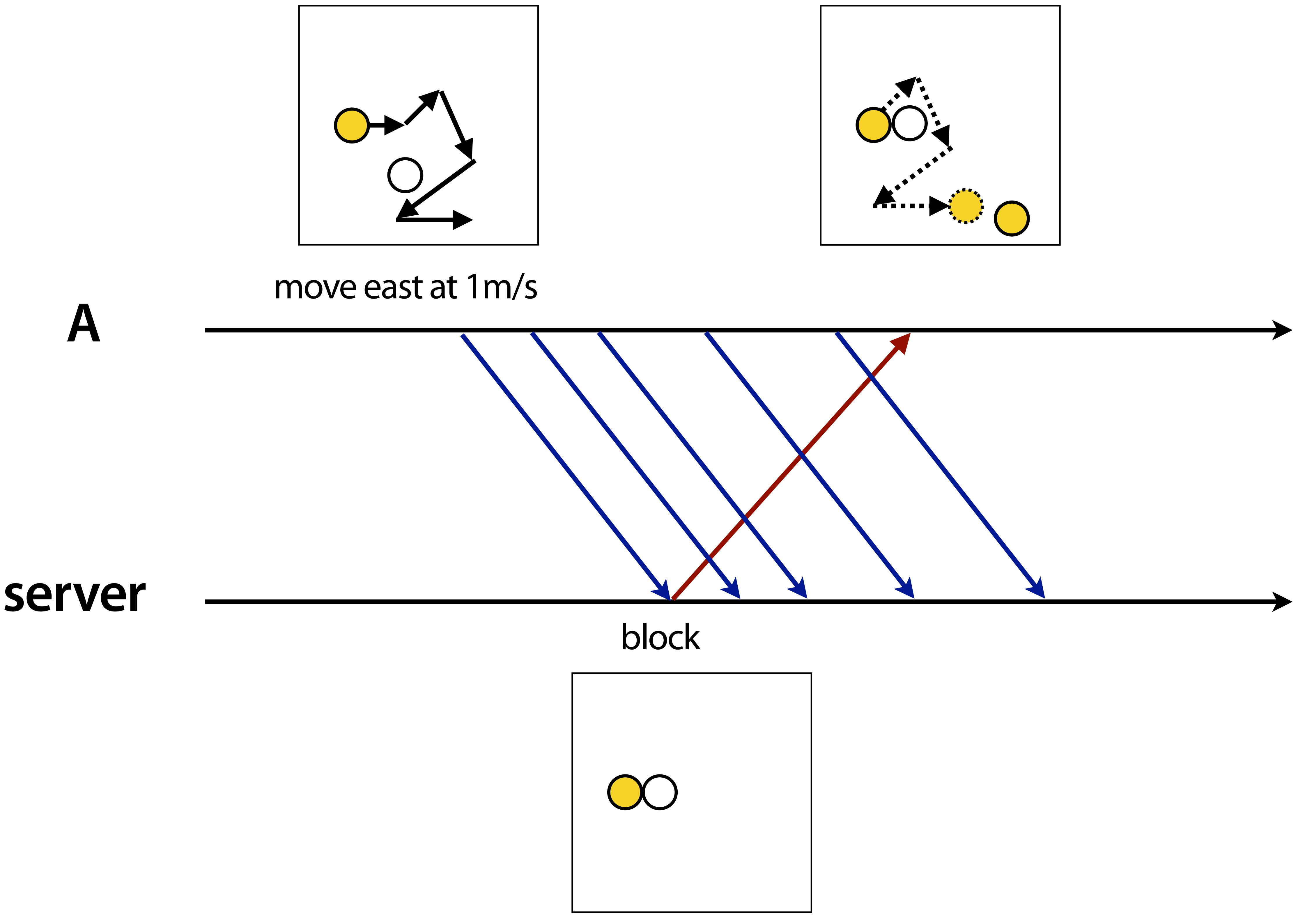


calculating the correct position gets tricky in twitch games



Unreal Tournament's lock-step predictor/corrector algorithm for player's movement

Player re-executes its move to find updated position.

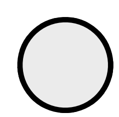


How to fix position error: Convergence

naive approach: player updates its position immediately -- teleporting to the correct position, causing visual disruption.

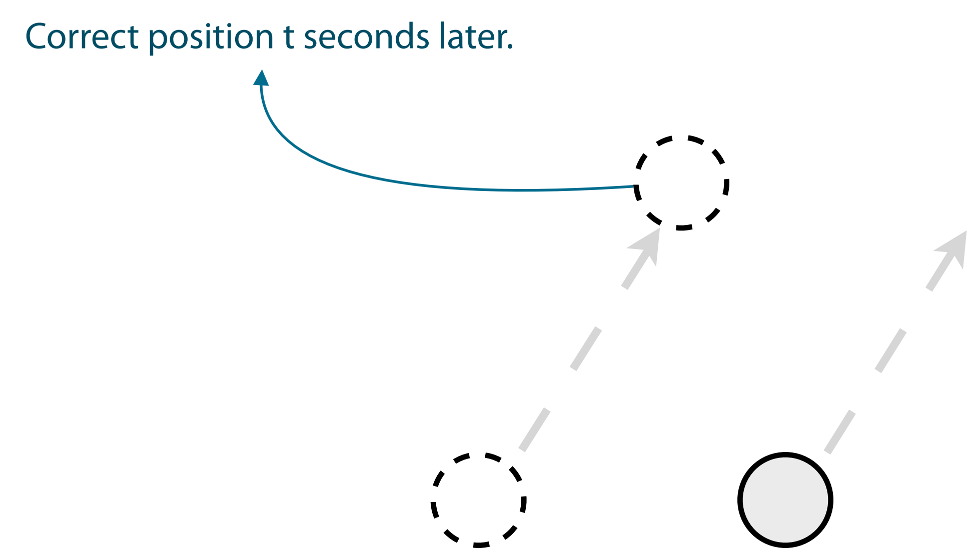
(zero order convergence)

naive approach: player updates its position immediately -- teleporting to the correct position, causing visual disruption.

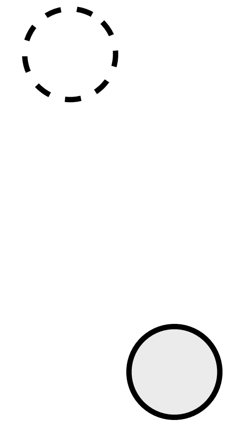


(zero order convergence)

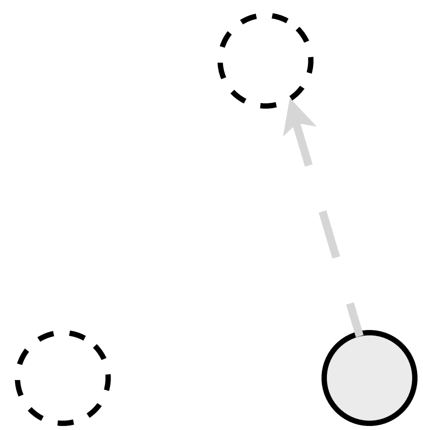
Convergence allows player to move to the correct position smoothly. First pick a convergence period *t*, and compute the correct position after time *t*.



Convergence allows player to move to the correct position smoothly. First pick a convergence period *t*, and compute the correct position after time *t*.

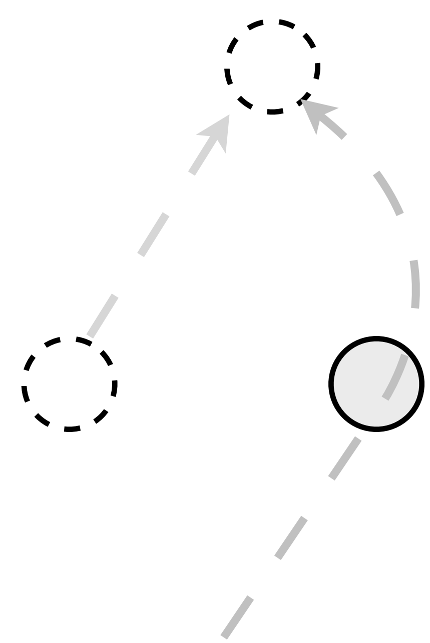


Move to that position in a straight line.

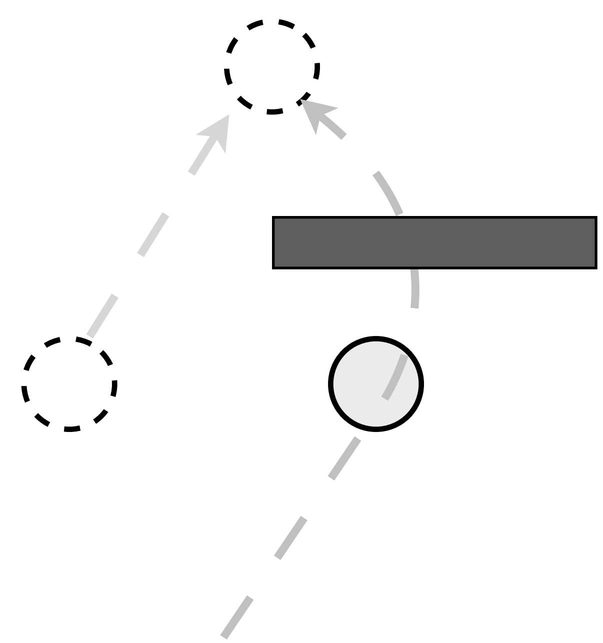


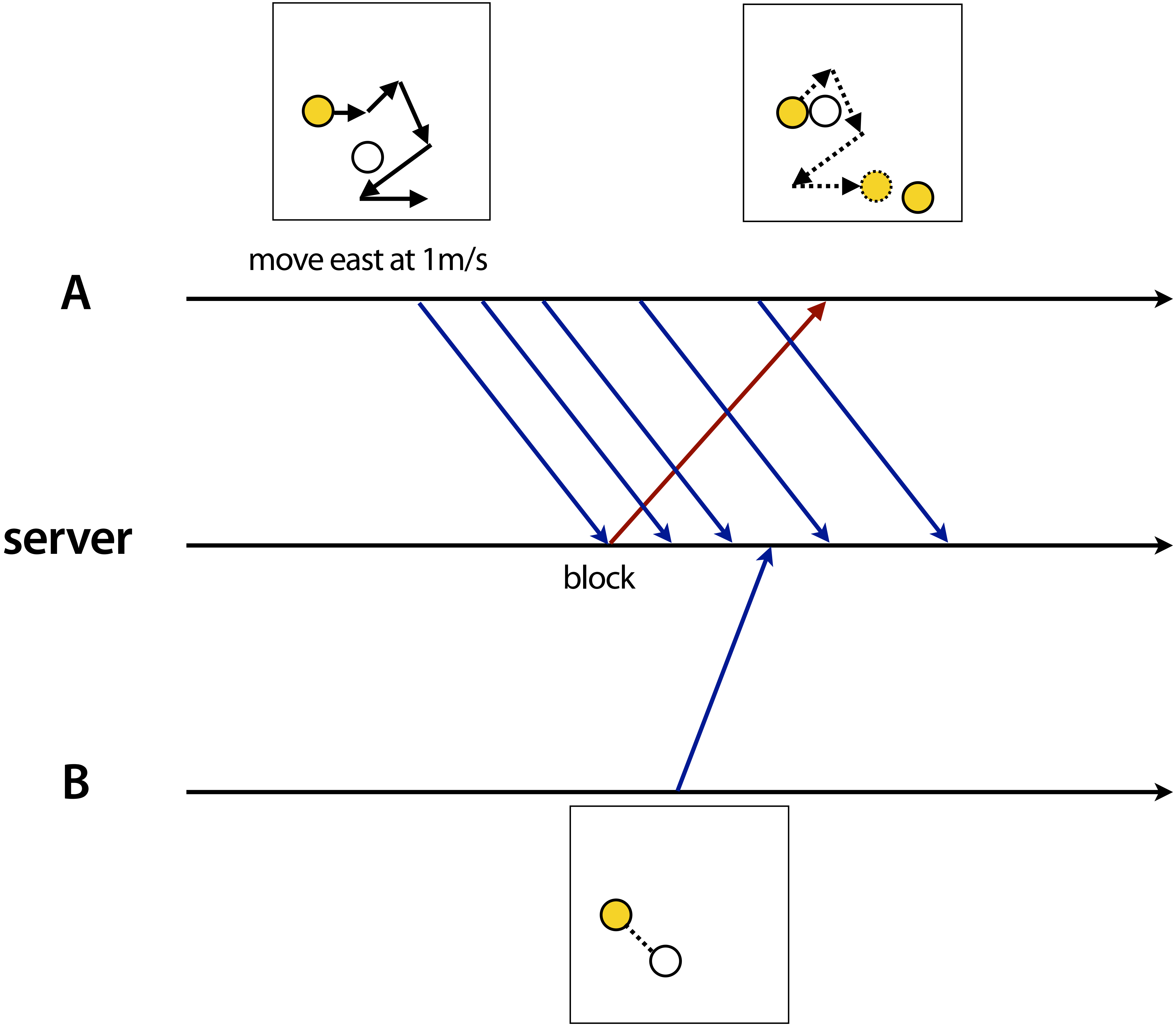
(linear convergence)

Curve fitting techniques can be used for smoother curves.



Visual disruption can still occur with convergence.




move east at 1m/s
A
server
block
B

Based on:

- A's state?
- B's state?
- server's state?

at the time when:

- B sends the message?
- server receives the message?

Easy to decide at B, but can't trust B. Have to decide at server. (permissible server architecture)

Finding B's state is harder.

Finding when B sends the message is easier.

Idea:

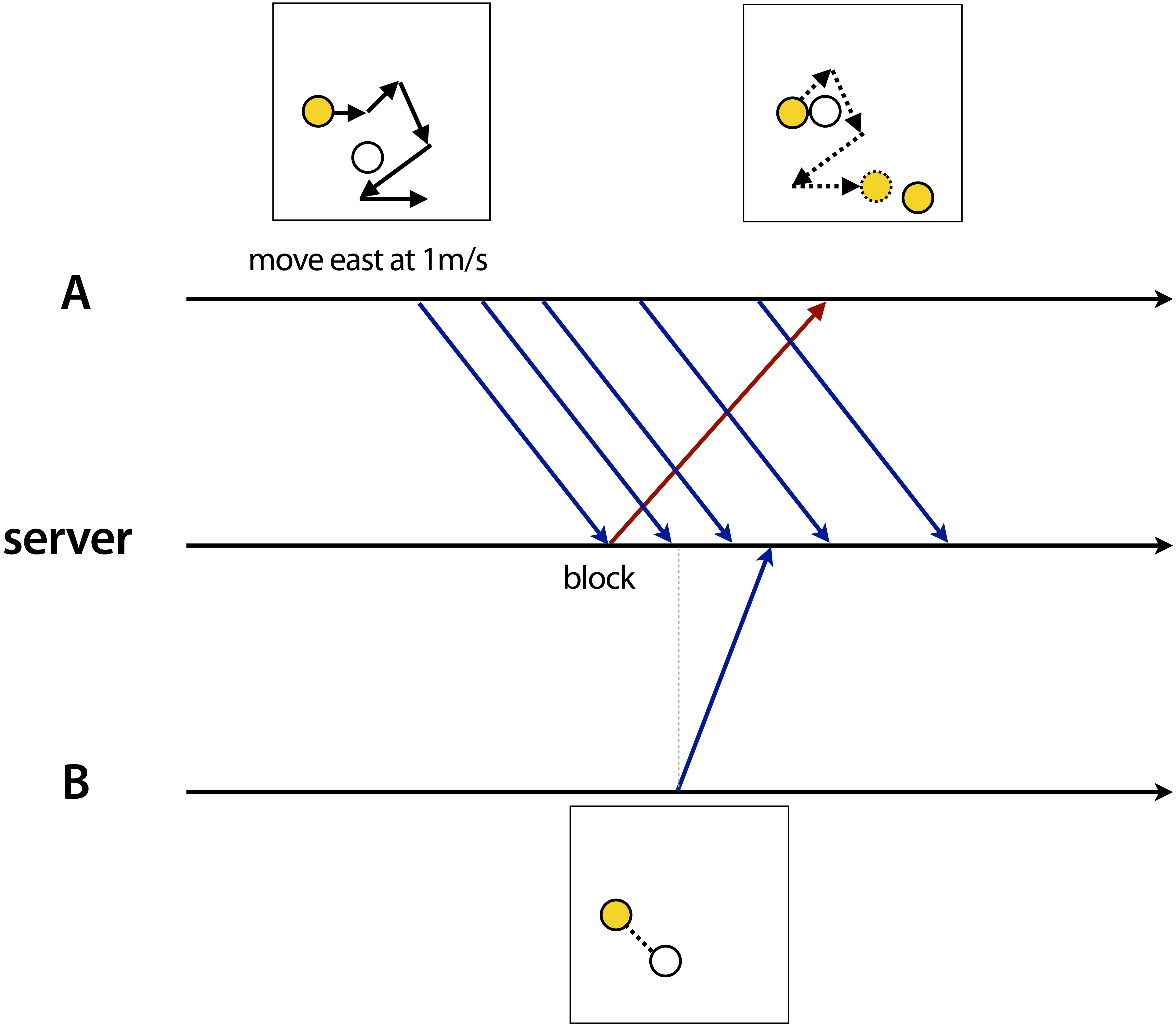
Lag Compensation

or

Time Warp

Based on server's state at the time when B sends the message

1. estimate $t = RTT/2$
2. rewind server's state to t seconds ago
3. resolve hit/miss
4. play forward to now


move east at 1m/s
A
server
block
B

Half-Life® 2: Episode One

Half-Life 2: Episode One The first in a trilogy of episodic games, Episode One reveals the aftermath of Half-Life 2 and launches a journey beyond City 17. Episode One does not require Half-Life 2 to play and also includes a first look at Episode Two.

GET HALF-LIFE 2: EPISODE ONE NOW!

Half-Life® 2

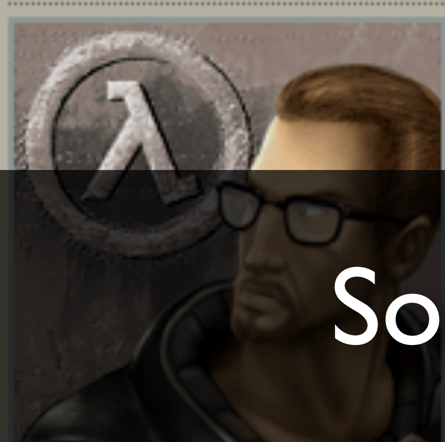
Half-Life 2 defines a new benchmark in gaming with startling realism and responsiveness. Powered by Source™ technology, Half-Life 2 features the most sophisticated in-game characters ever witnessed, advanced AI, stunning graphics and physical gameplay.

GET HALF-LIFE 2 NOW!

Counter-Strike™: Source™

Counter-Strike: Source blends Counter-Strike's award-winning teamplay action with the advanced technology of Source™ technology. Featuring state of the art graphics, all new sounds, and introducing physics, Counter-Strike: Source is a must-have for every action gamer.

GET COUNTER-STRIKE:SOURCE NOW!



Half-Life: Source

Winner of over 50 Game of the Year awards, Half-Life set new standards for action games when it was released in 1998. Half-Life: Source is a digitally remastered version of the critically acclaimed and best selling PC game, enhanced via Source technology to include physics simulation, enhanced effects, and more.

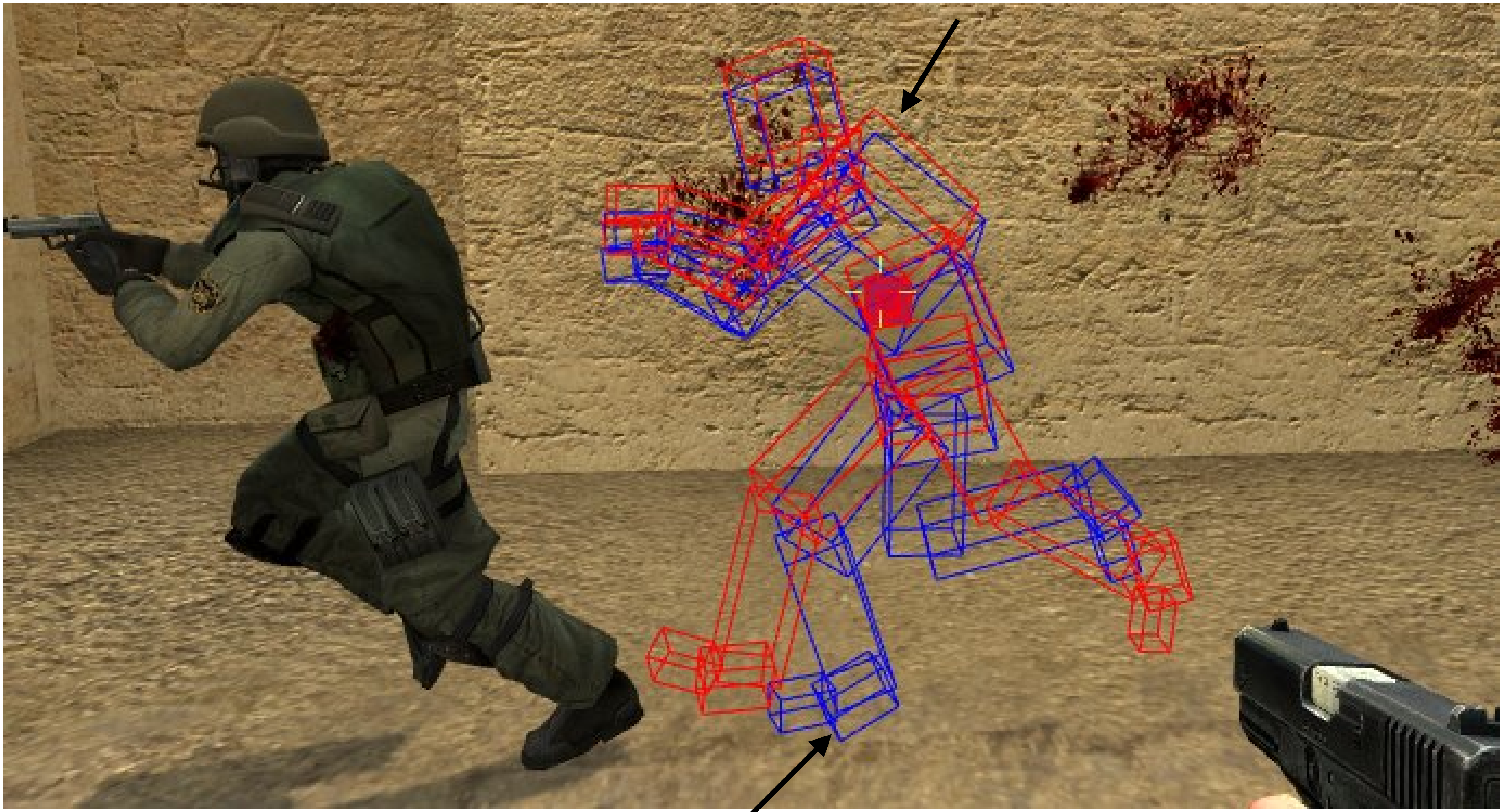
GET HALF-LIFE:SOURCE NOW!

Source Multuplayer Game Engine

RTT/2 seconds after B shoots

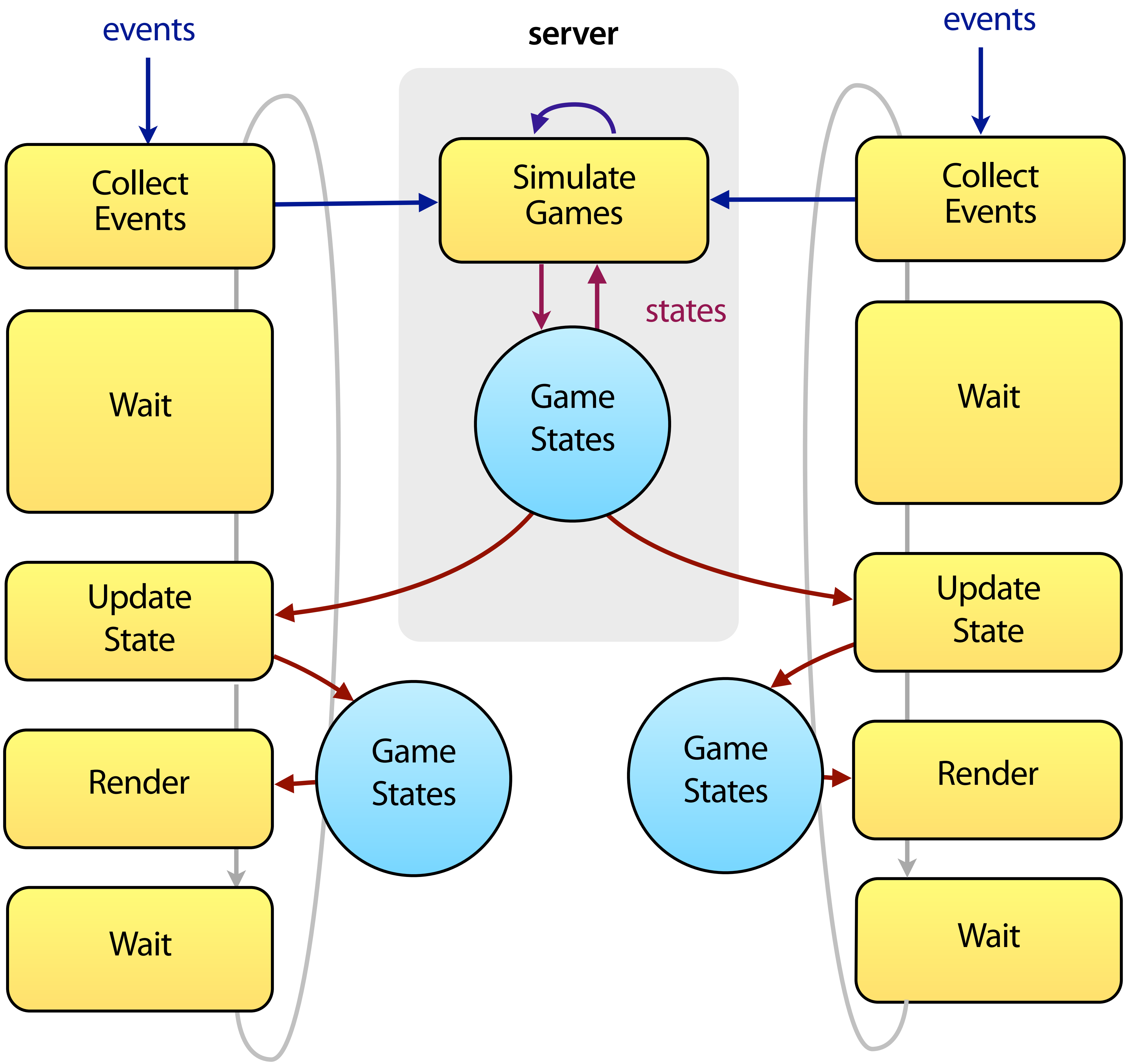
What B sees now

red: What B saw RTT/2 seconds ago



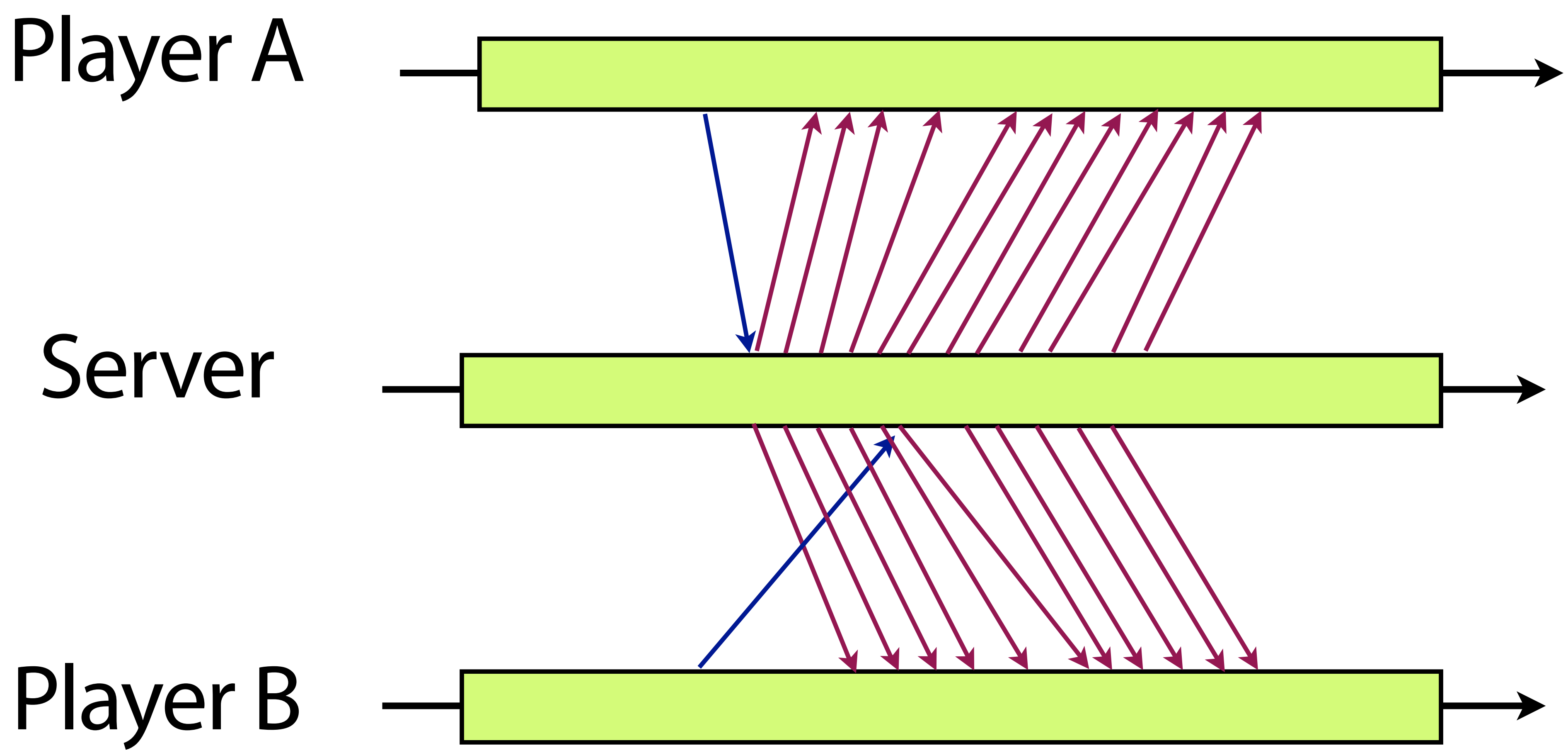
blue: What server thinks B saw RTT/2 seconds ago

http://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking



events
server
events
Collect Events
Simulate Games
Collect Events
states
Wait
Game States
Wait
Update State
Update State
Game States
Game States
Render
Render
Wait
Wait

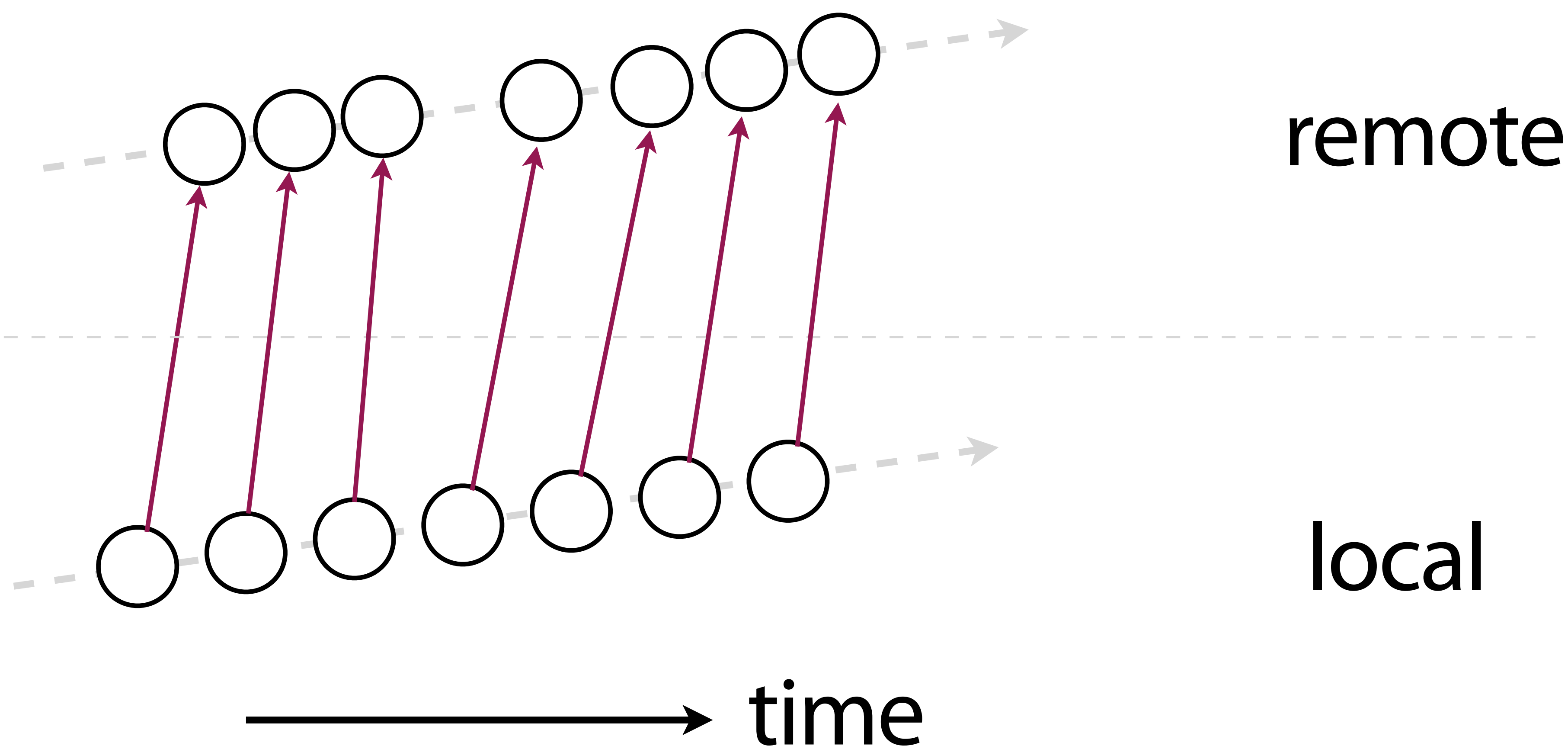
Players send move command. Server replies with new positions periodically.



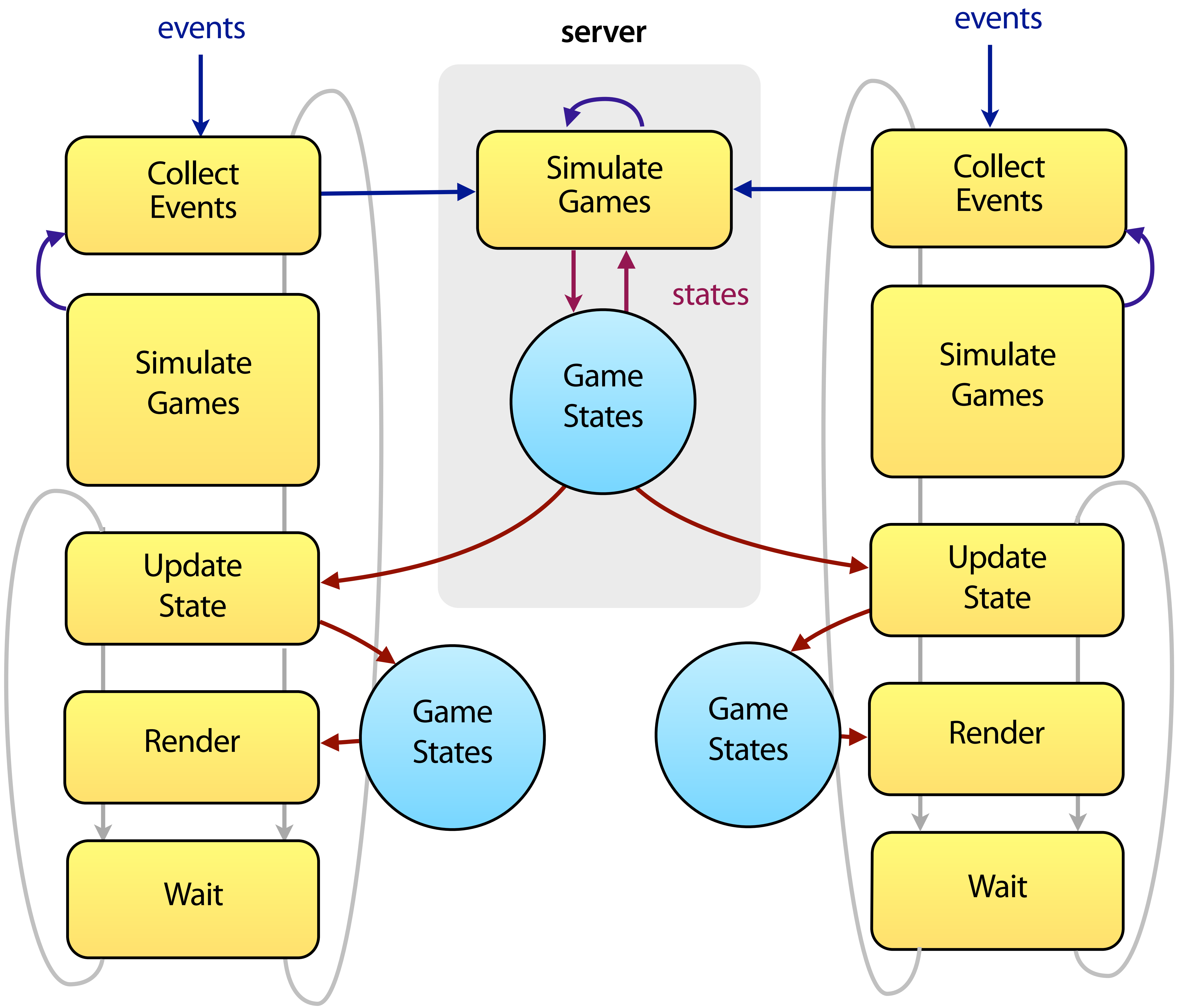
Issues:

1. Message overhead
2. Delay jitter

Delay jitter causes player's movement to appear erratic.

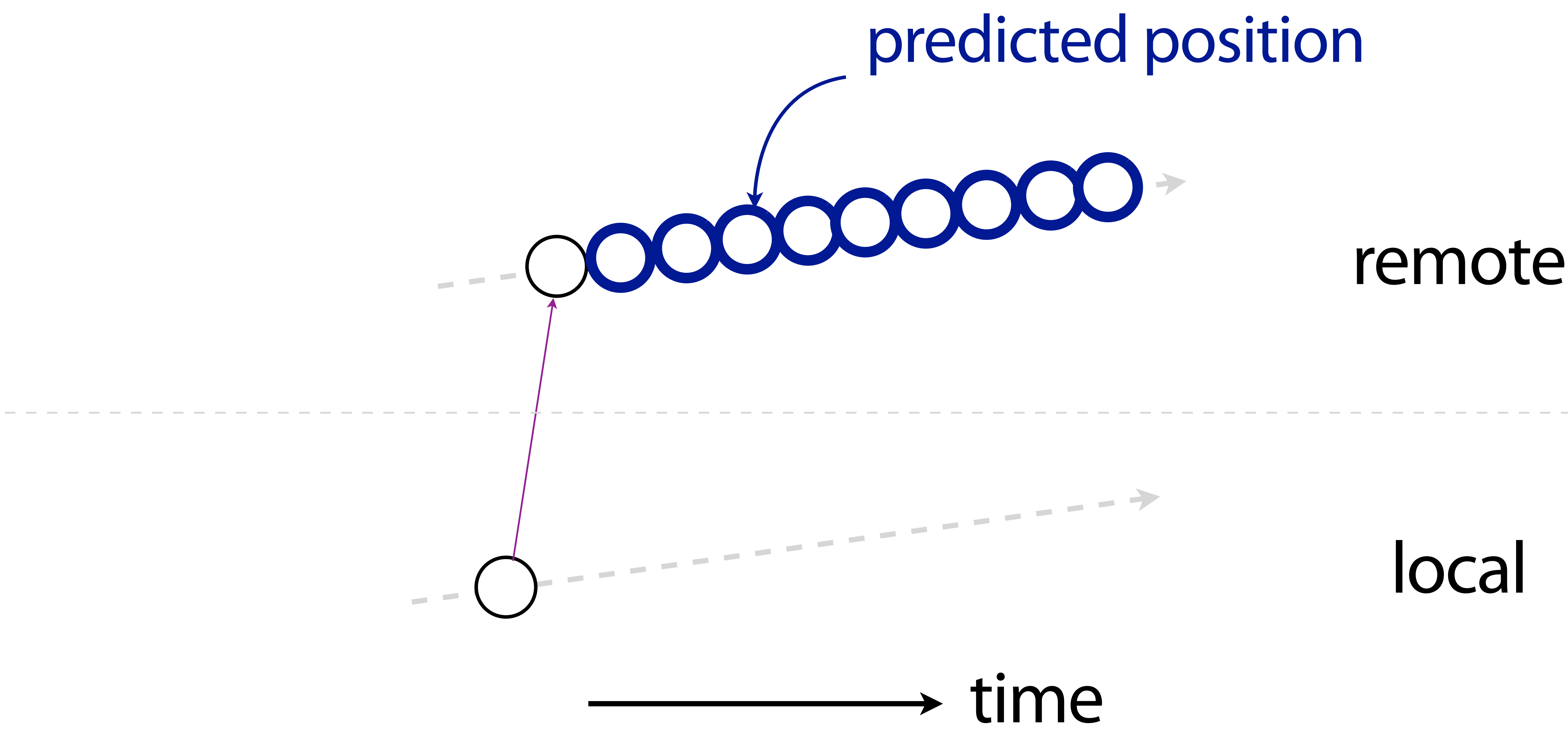


Demo:
2 Player Pong



events
server
events
Collect Events
Simulate Games
Collect Events
Simulate Games
states
Simulate Games
Game States
Simulate Games
Update State
Update State
Game States
Game States
Render
Render
Wait
Wait

Suppose the velocity remains constant, then we can predict every position at all time.

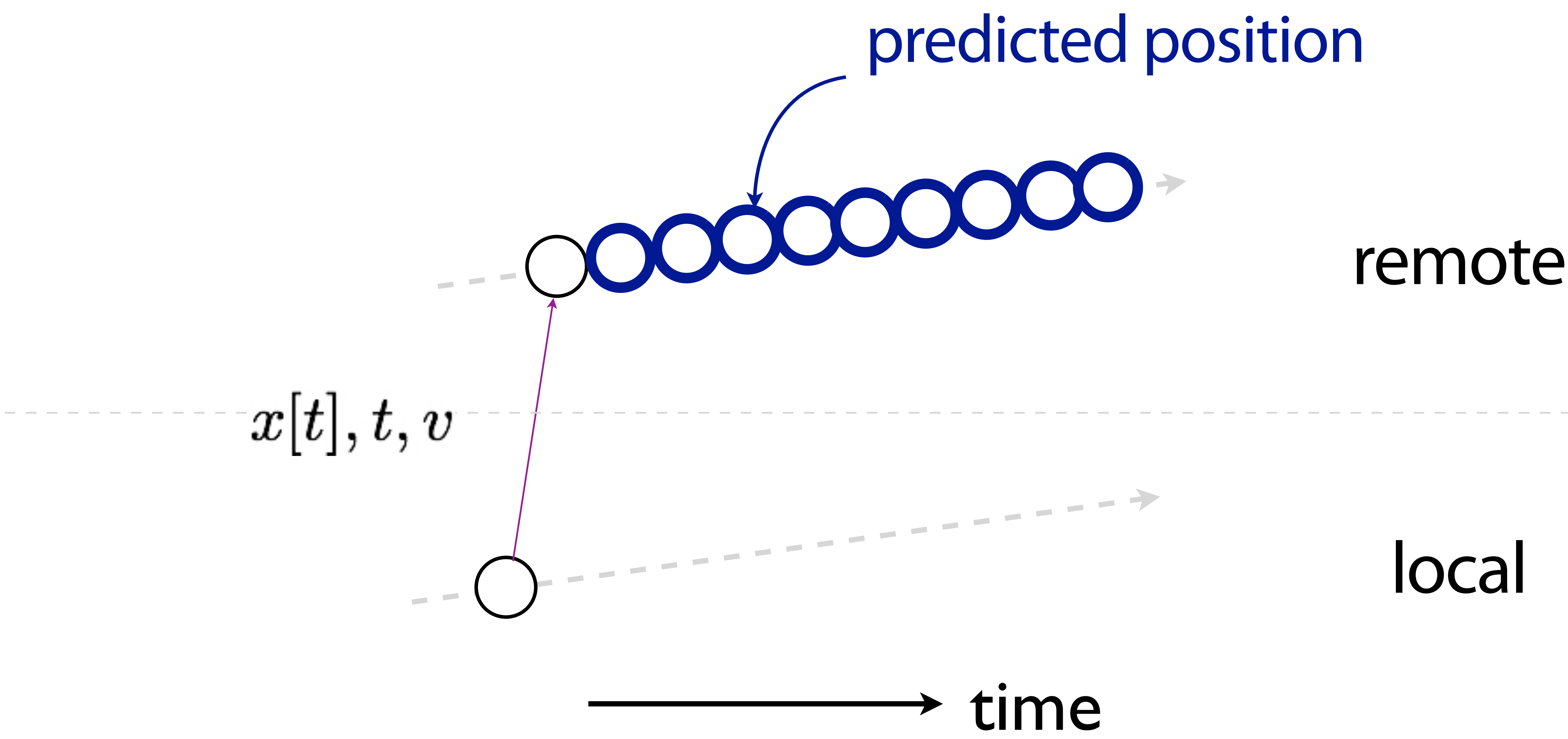


$x[t]$ position of entity at time t

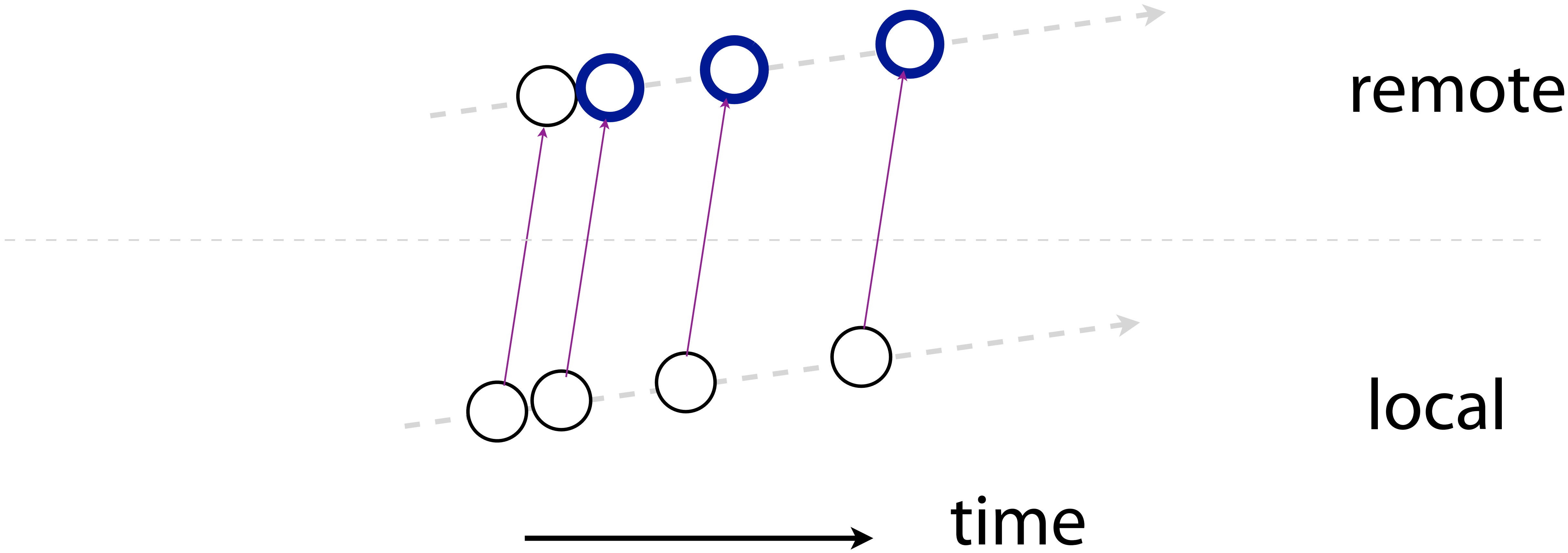
v velocity of the entity

$$x[t_i] = x[t_{i-1}] + v \times (t_i - t_{i-1})$$

We send over the initial position x[t], t, and velocity. (Why do we need to send t?)



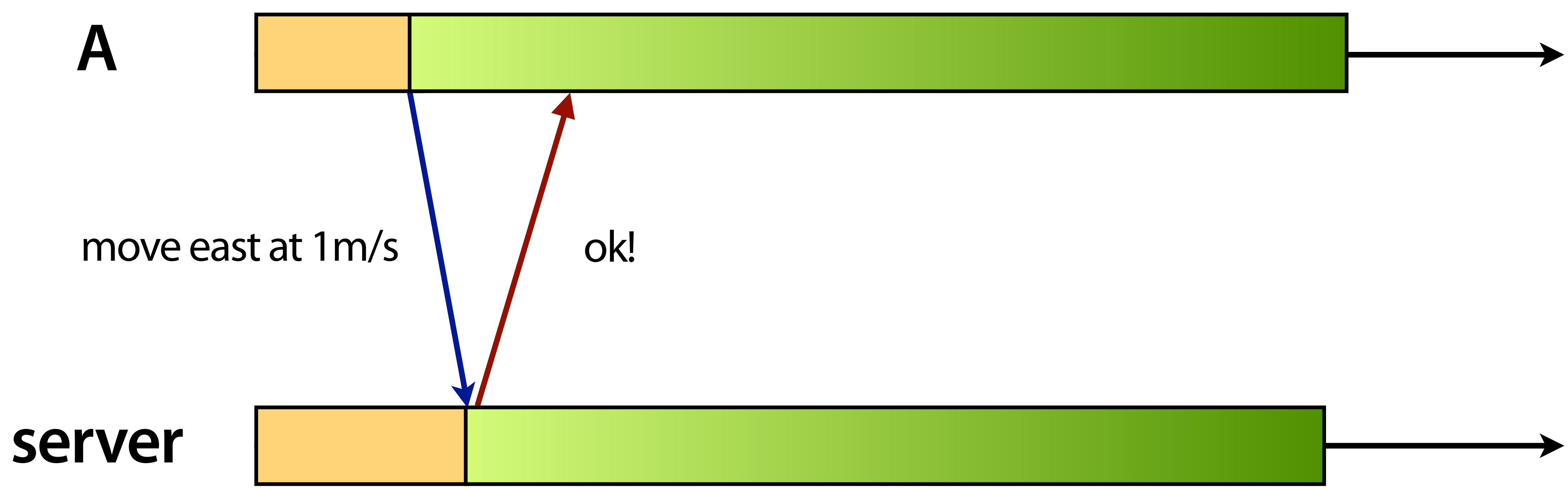
But velocity may change (e.g. a car accelerating). To counter this, we send position, velocity, and acceleration as update.



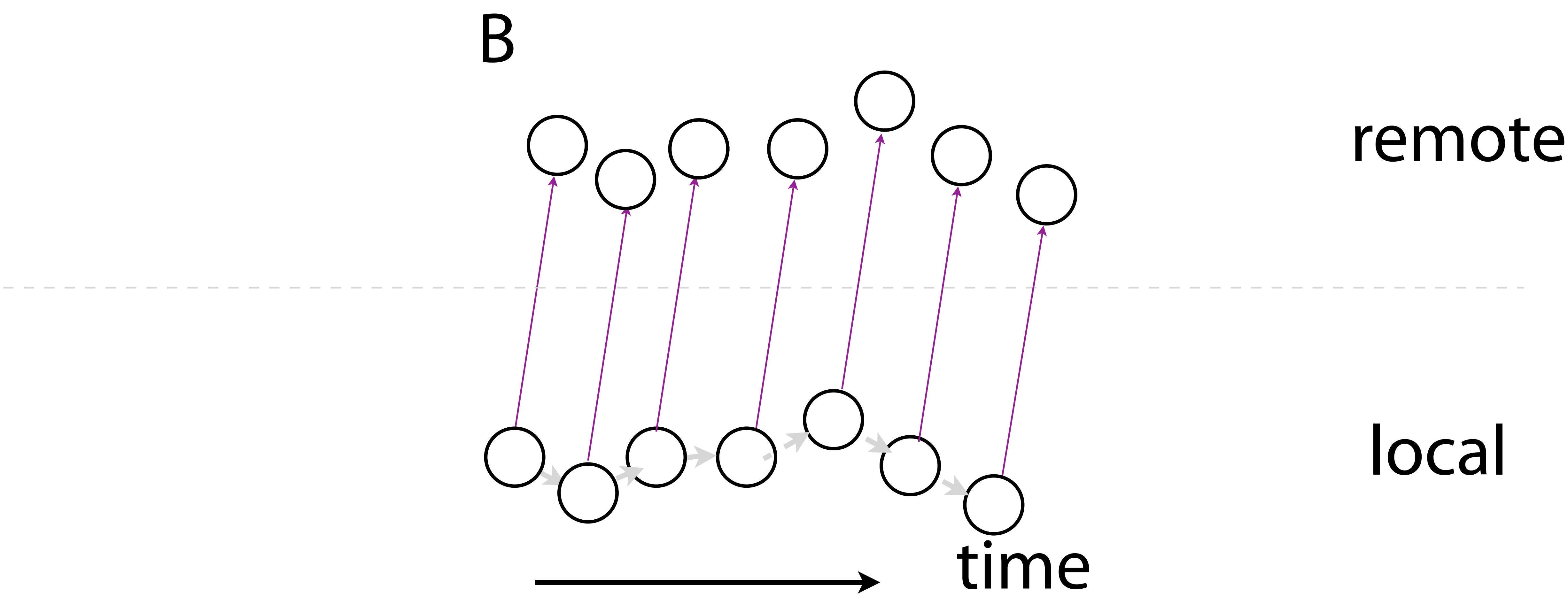
$x[t]$	position of entity at time t
v	velocity of the entity
a	acceleration of the entity

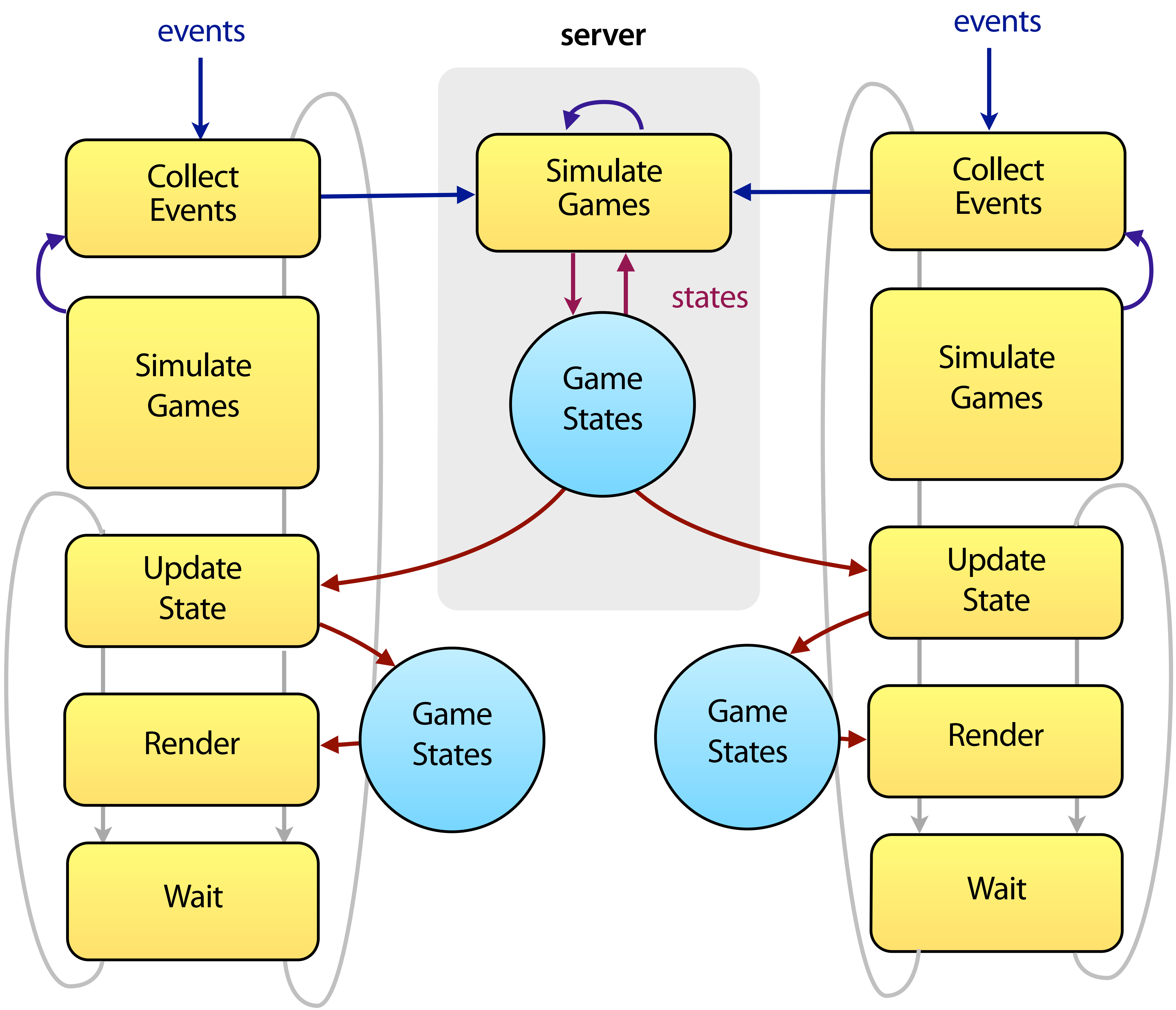
$$x[t_i] = x[t_{i-1}] + v(t_i - t_{i-1}) + \frac{1}{2}a(t_i - t_{i-1})^2$$

local states are updated continuously at player



We will still need substantial number of updates if the direction changes frequently (e.g. in a FPS game).



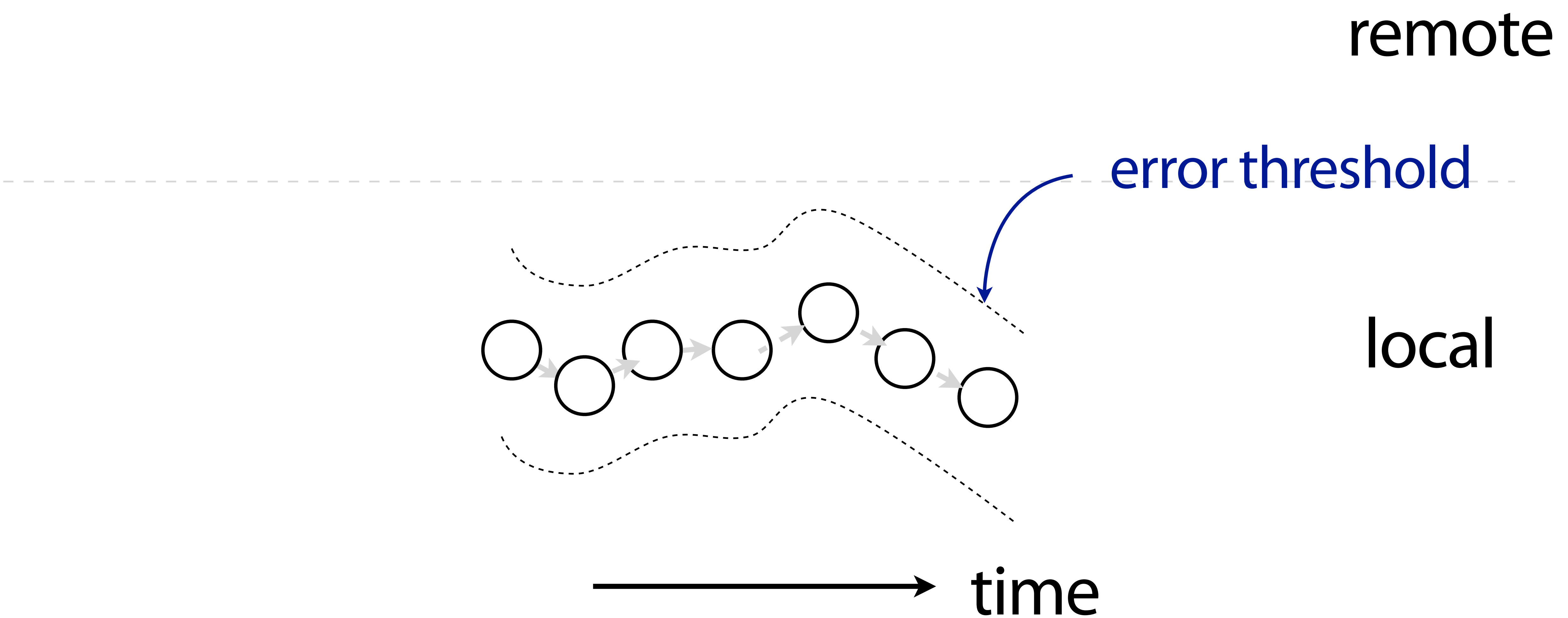


events
server
events
Collect Events
Simulate Games
Collect Events
states
Simulate Games
Game States
Simulate Games
Update State
Update State
Game States
Game States
Render
Render
Wait
Wait

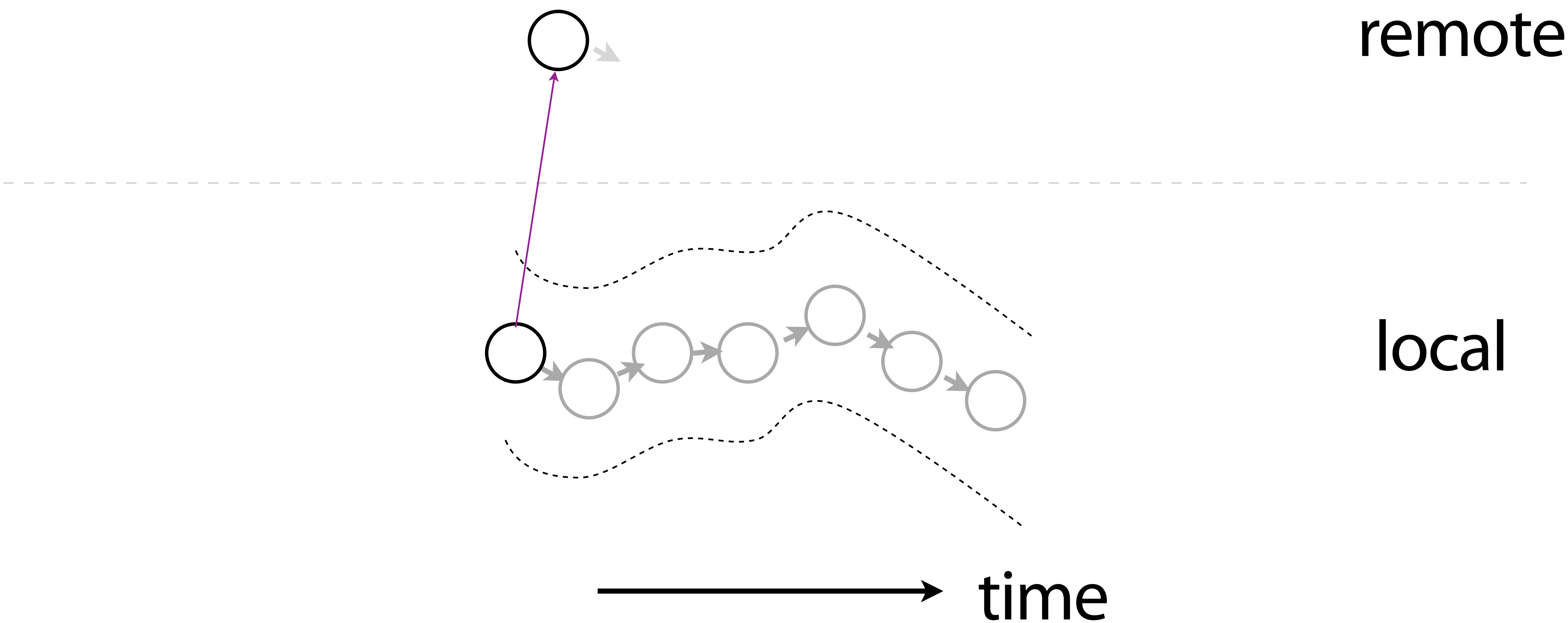
Idea:

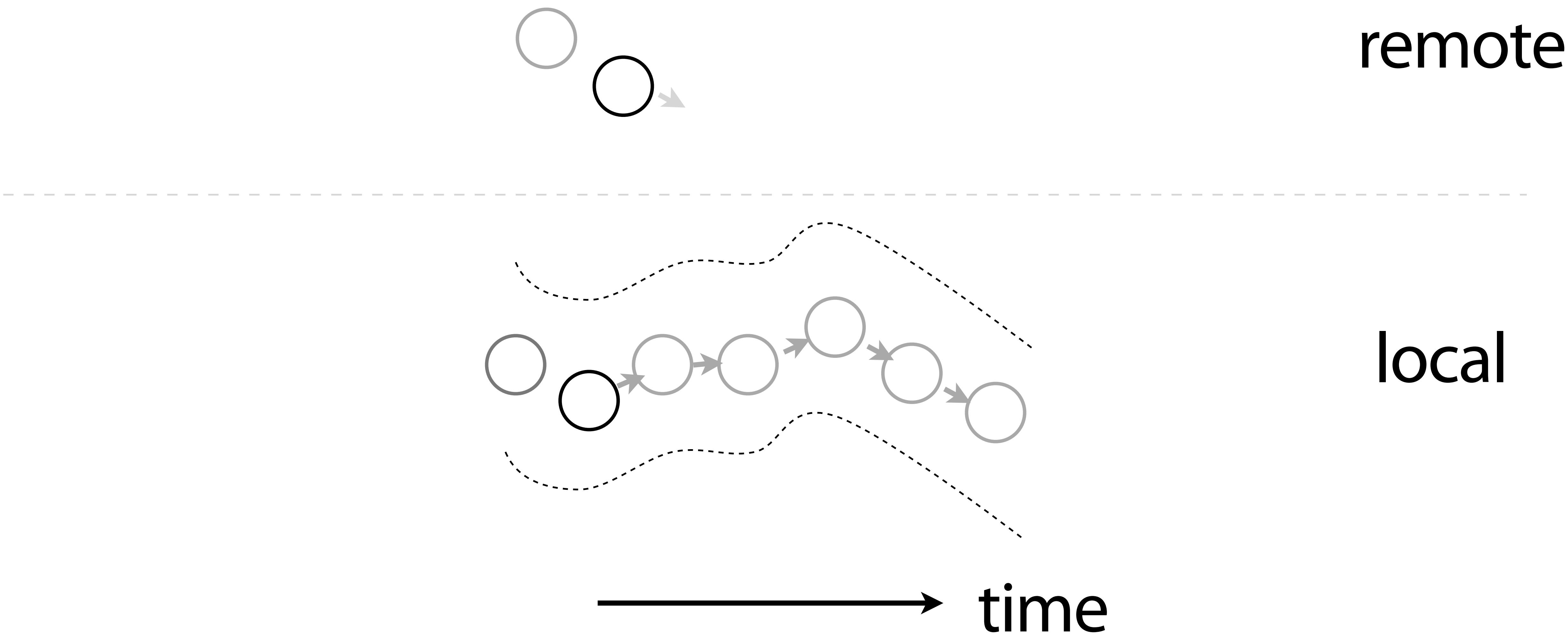
Dead Reckoning

Trade off message overhead with position accuracy -- (no update if error is small)



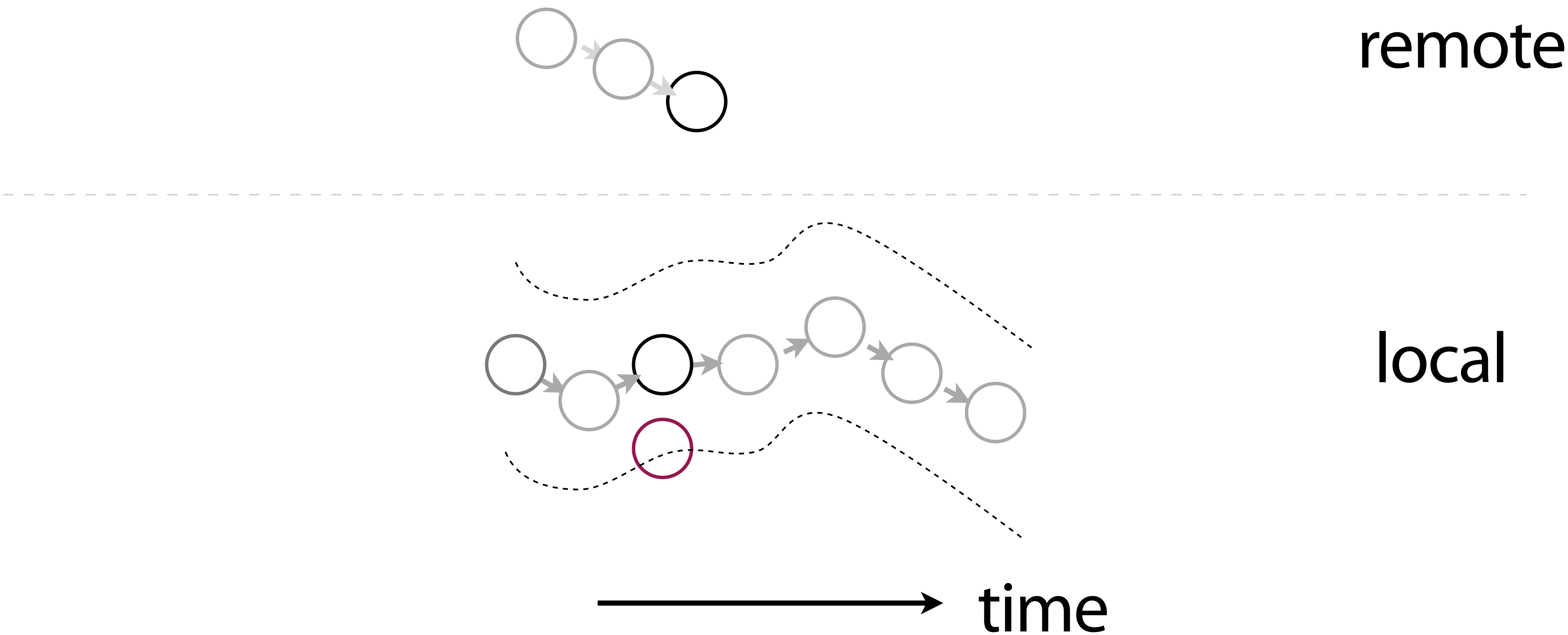
remote
error threshold
local
time


remote
local
time

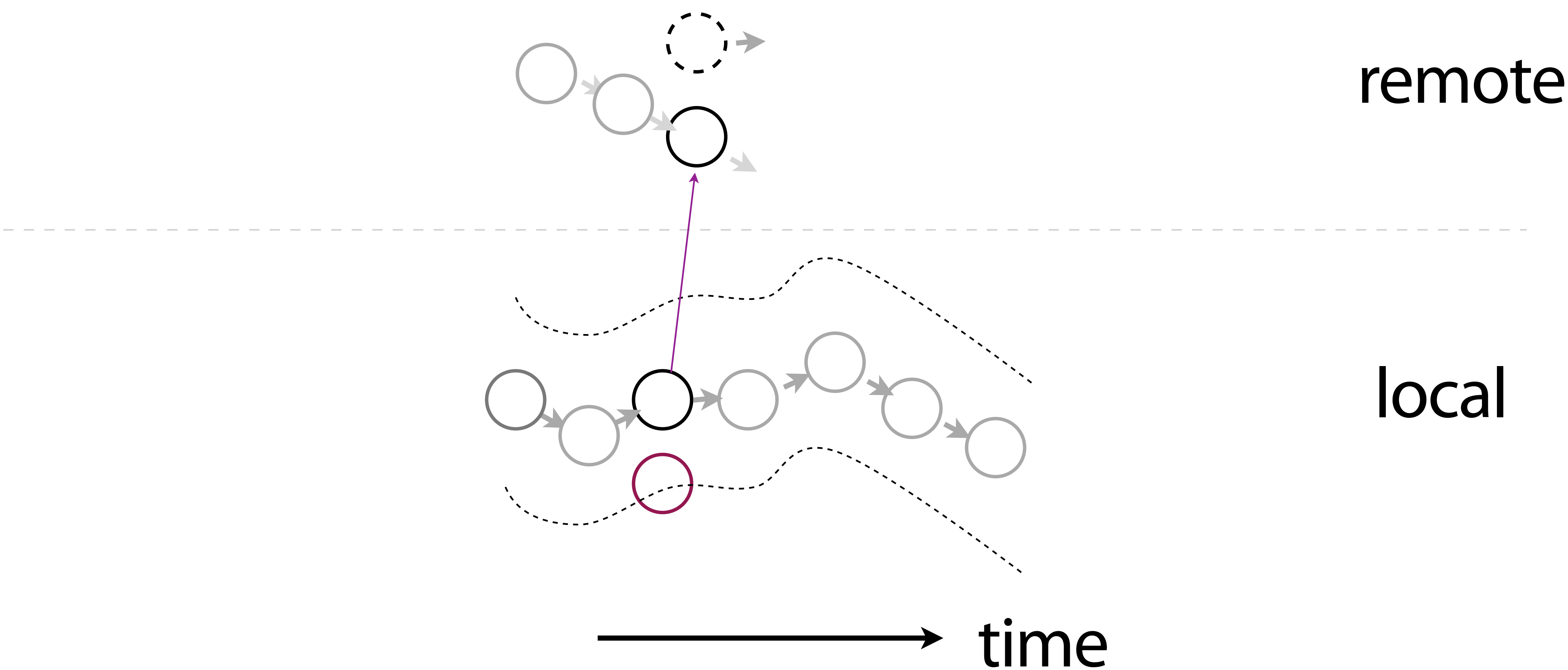

remote
local
time

predicted position

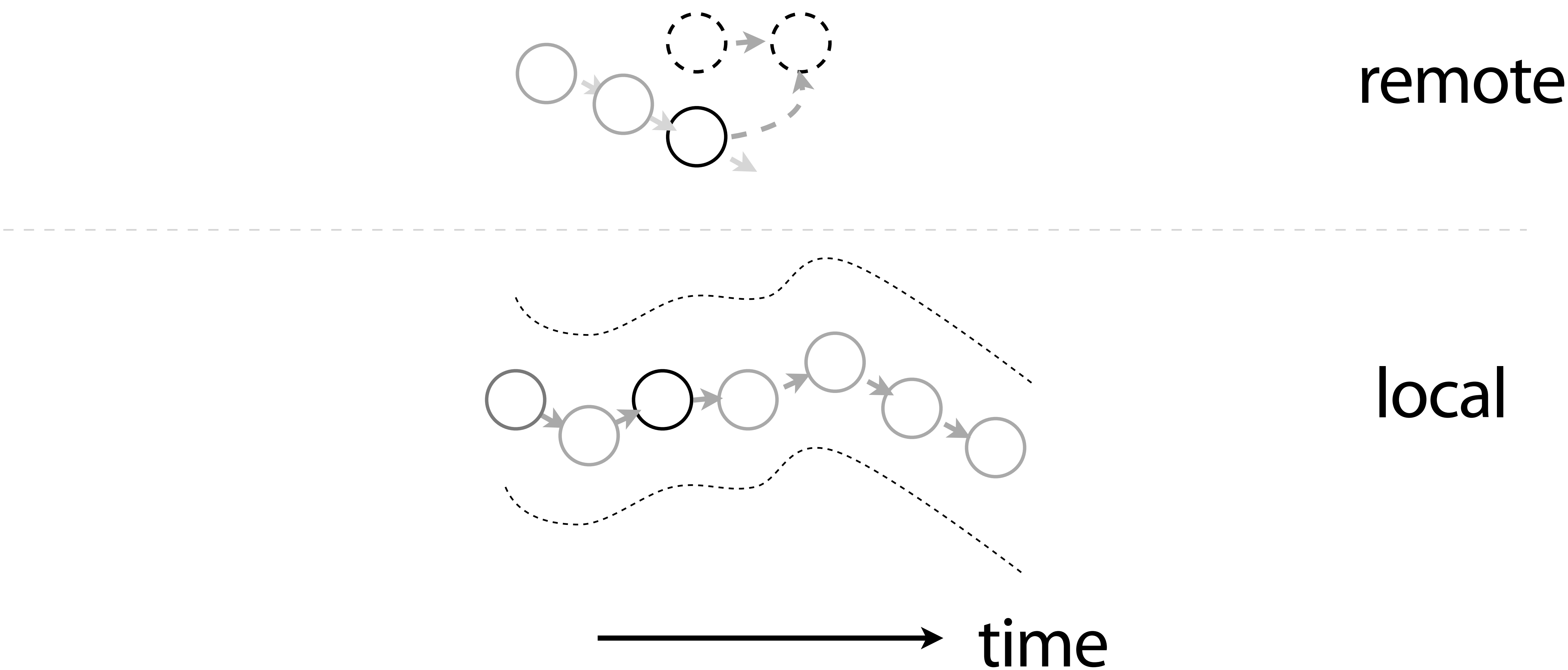
local version of the predicted position



local and predicted position are now too far apart. Update remote host with the new velocity and position.



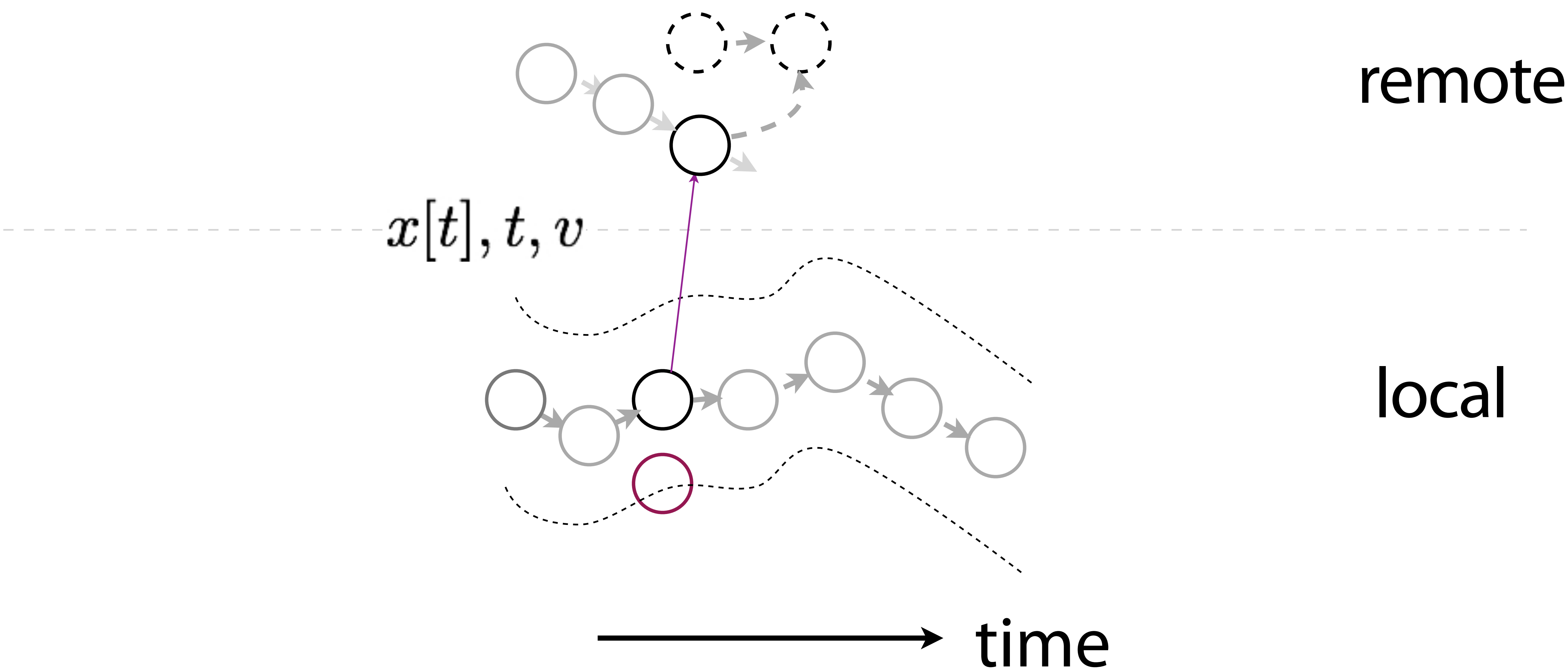
The remote host converges the entity to the correct position smoothly.



For the local to know the predicted position, it needs to simulate the remote view of the entity location.

Space inconsistency: due to error threshold and convergence

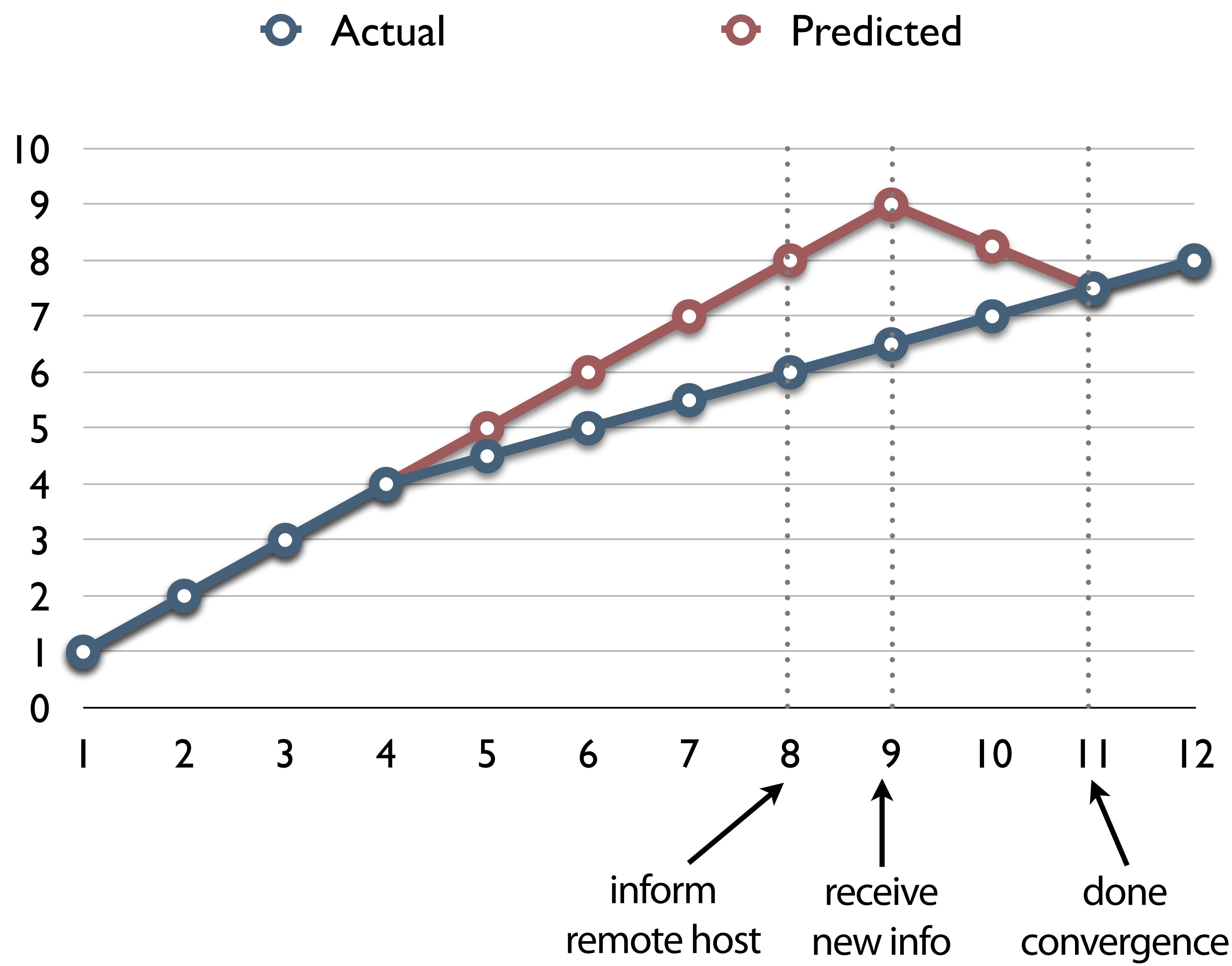
Time inconsistency: due to message delay and clock asynchrony

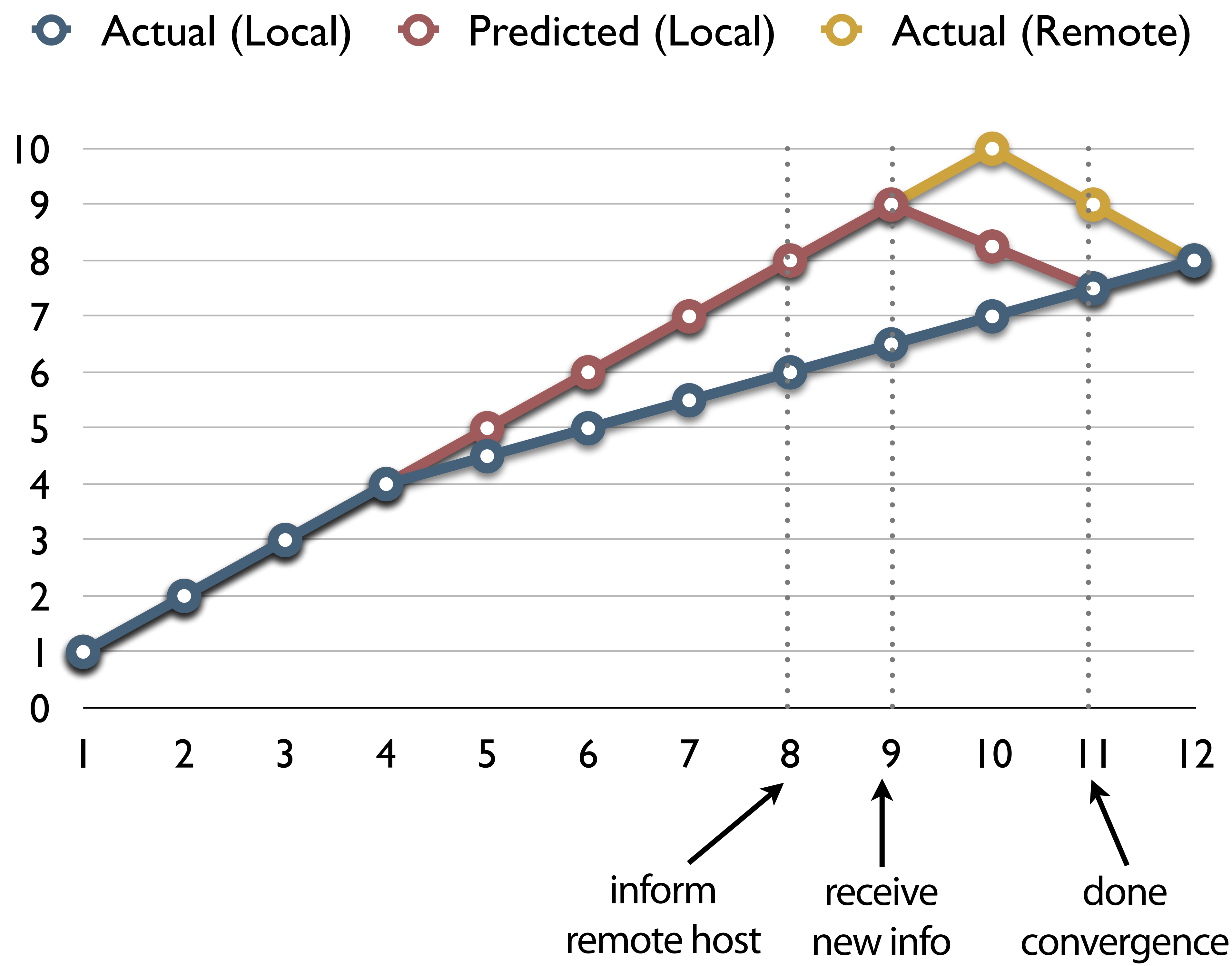


What is the difference between the actual and predicted position ?

How long does the difference last?

Dead Reckoning Error Analysis (in 1D)


Actual
Predicted
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
11
12
inform
remote host
receive
new info
done
convergence


Actual (Local)
Predicted (Local)
Actual (Remote)
10
9
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6
7
8
9
10
11
12
inform remote host
receive new info
done convergence

higher CPU cost

(needs to simulate other players)

unfair

(higher latency leads to larger error)

how to determine
the error threshold?

Demo:

2 Player Pong