§4.3 Frequent State Regeneration

- Many NVEs cannot afford the communications and processor overhead required to support absolute consistency through a centralized repository
- Many NVEs do not require high level consistency
- Limited and temporary error is allowable
- Smooth interface vs. absolute consistency

- Replace the distributed consistency protocol with a more aggressive state update notification system

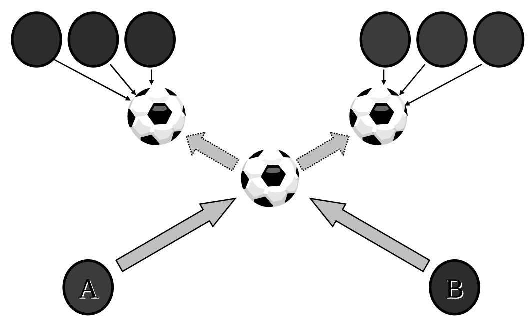
Frequent State Regeneration (cont'd)

- Source host does not care what state information is cached or available to other hosts
- Each update contains whole entity state, whether or not it has changed
- The owner of information uses *blind broadcast*
  - asynchronously and unreliably
  - at a regular interval
  - forward to all participants
- The receiver does not acknowledge packets
- Assumption: high transmission rate will make inconsistencies relatively unnoticeable
- Even with moderate packet loss, blind broadcast can typically deliver more packets than shared database due to its overhead

Entity Ownership: Background

- Blind broadcasting sacrifices absolute consistency, and reduces some flexibility that centralized repositories offer
- In a centralized repository system
  - any host can modify any entity
  - reliable and order-preserving updates
- With frequent state regeneration systems, ensure that multiple hosts do not attempt to update an entity at the same time

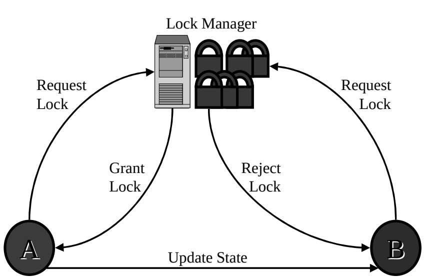
Problem: Who's Got the Ball Now? (Part II)



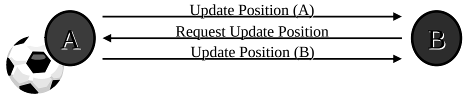
Explicit Entity Ownership

- Ensure that shared state can only be updated by one host at a time
  - exactly one host has the ownership of the state
  - the owner periodically broadcasts the value of the state
- Typically user's own representation (avatar) is owned by that user
- Locks on other entities are managed by a lock manager server
  - clients query to obtain ownership and contact to release it
  - the server ensures that each entity has only one owner
  - the server owns the entity if no one else does
  - failure recovery

Lock Manager: Example

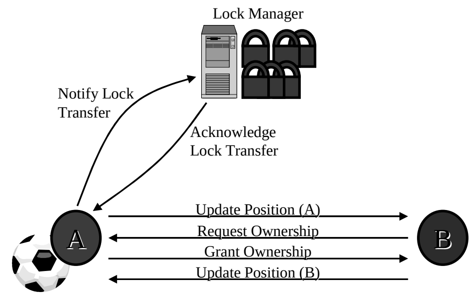


Proxy Update



- Non-owner sends an update request to the owner of the state
- The owner decides whether it accepts the update
- The owner serves as a proxy
- Generates an extra message on each non-owner update
- Suitable when non-owner updates are rare or many hosts want to update the state

Ownership Transfer

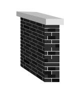


Ownership Transfer (cont'd)

- The lock manager has the lock information at all times
- If the host fails, the lock manager defines the current lock ownership state

- Lock ownership transfer incurs extra message overhead
- Suitable when a single host is going to make a series of updates and there is little contention among hosts wishing to make updates

Reducing Broadcast Scope



- In a frequent state regeneration system, each host sends updates to all participants
  - causes hosts to receive lots of extraneous information
- Multicast and area-of-interest techniques
  - filter the updates before they get sent to inappropriate recipients
- Who should do the filtering?
  - the host itself?
  - a server?
- We shall return to this in §6

Frequent State Regeneration: Advantages and Drawbacks

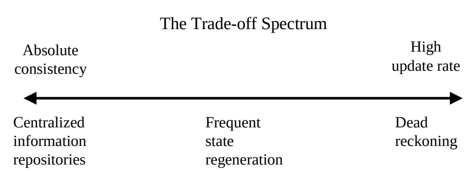
- Adds multi-user capabilities to existing single-user applications
- Blind broadcasting does not require a server, consistency protocol nor a lock manager (in most cases)
- Offers support for a large number of users
- Exhibits better interactive behaviour

- Requires considerable network bandwidth
- Susceptible to network latency
  - jitter = variation in network latency from one packet to the next
- Assumes that all hosts are broadcasting at the same rate

Flashback: Maintaining Dynamic Shared State

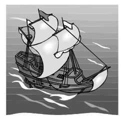
Three basic approaches to maintain dynamic shared state:

- shared repositories
- frequent broadcast
- state prediction

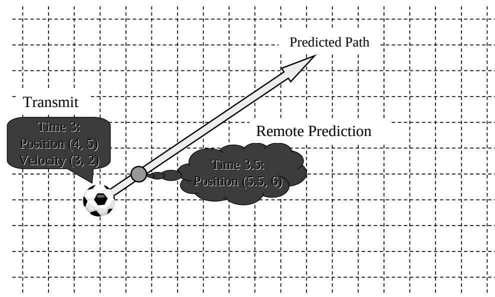


§4.4 Dead Reckoning of Shared State

- Transmit state update packets less frequently
- Use received information to *approximate* the true shared state
- In between updates, each host predicts the state of the entities



Dead Reckoning: Example

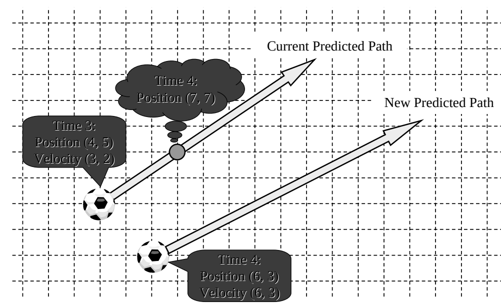


Dead Reckoning Protocol

DR protocol consists of two elements:

- prediction technique
  - how the entity's current state is computed based on previously received update packets
- convergence technique
  - how to correct the state information when an update is received

Prediction and Convergence



Prediction Using Derivative Polynomials

- The most common DR protocols use derivative polynomials
- Involves various derivatives of the entity's current position
- Derivatives of position
  1. velocity
  2. acceleration
  3. jerk

Zero-Order and First-Order Polynomials

- Zero-order polynomial
  - position p
  - the object's instantaneous position, no derivative information
  - predicted position after t seconds = p

⇒The state regeneration technique

- First-order polynomial
  - velocity v
  - predicted position after t seconds = $vt + p$
  - update packet provides current position and velocity

Second-Order Polynomials

- We can usually obtain better prediction by incorporating more derivatives
- Second-order polynomial
  - acceleration *a*
  - predicted position after *t* seconds $= \frac{1}{2}at^2 + vt + p$
  - update packet: current position, velocity, and acceleration
  - popular and widely used
  - easy to understand and implement
  - fast to compute
  - relatively good predictions of position