
Fundamentals of Parallel and Distributed Simulation

Wentong CAI

aswtcai@ntu.edu.sg

School of Computer Engineering

Nanyang Technological University

Singapore 639798

Main References

◆ PADS

- R. Fujimoto, Parallel and Distributed Simulation Systems, Wiley Interscience, 2000 (see also http://www.cc.gatech.edu/classes/AY2006/cs4230_fall/index.html)

◆ HLA

- F. Kuhl, R. Weatherly, J. Dahmann, Creating Computer Simulation Systems: An Introduction to the High Level Architecture for Simulation, Prentice Hall, 1999.
(see also <http://hla.dmsso.mil>)

◆ Acknowledgement: Many slides are adopted from

- Prof. Richard Fujimoto's PADS'04 tutorial and lecture slides

Outline

- ◆ Introduction
 - **Basic Concepts**
 - **Parallel and Distributed Simulation**
 - **Simulation Decomposition**
 - **Causality**
- ◆ Conservative Synchronization
 - **A Simple Conservative Algorithm**
 - **Null Message Algorithm**
 - **Synchronous Algorithms**
- ◆ Optimistic Synchronization
 - **Time Warp**
 - **Local Control Mechanisms**
 - **Global Control Mechanisms**
 - **Other Mechanisms**
- ◆ HLA And Distributed/Federated Simulation

Outline – Introduction

- ◆ **Basic Concepts**
 - **Modelling and Simulation**
 - **Time Advancement Mechanism**
 - **Model and Execution**
- ◆ **Parallel and Distributed Simulation**
 - **What is Parallel and Distributed Simulation?**
 - **Motivation for Parallel and Distributed Simulation**
- ◆ **Simulation Decomposition**
 - **Logical Processes**
 - **Time Stamped Messages**
- ◆ **Causality**
 - **Local Causality Constraint**
 - **Synchronization Problem**
 - **Overview of Synchronization Protocols**

Modelling and Simulation

- ◆ A system: A group of **objects** that are joined together in some regular **interaction** or interdependence toward the accomplishment of some purpose.
- ◆ Model: A representation of an **entity**, **process**, or system
- ◆ Simulation: The process of exercising a model to characterize the behavior of the modeled entity process, or system over time
- ◆ Computer simulation: a simulation where the system doing the emulating is a **computer program**

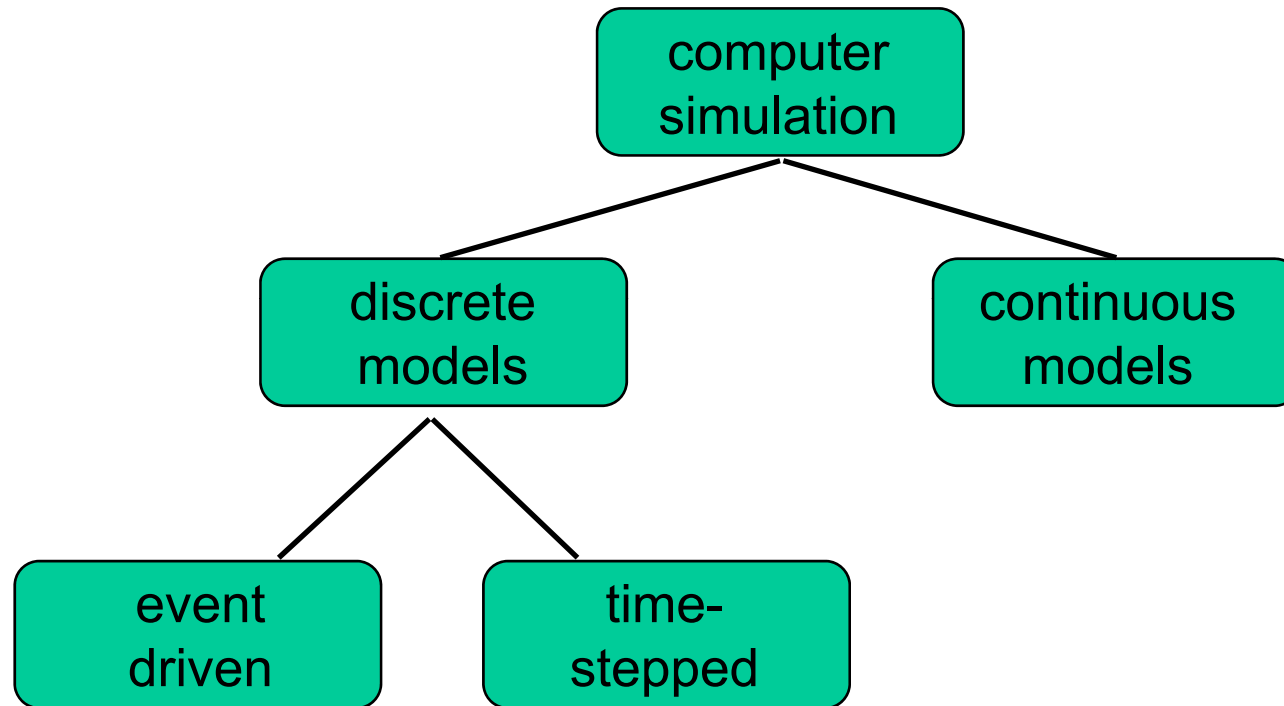
Why Use Simulations?

- ◆ It may be too difficult, hazardous, or expensive to observe a real, operational system
- ◆ Parts of the system may not be observable (e.g., internals of a silicon chip or biological system)

Uses of simulations

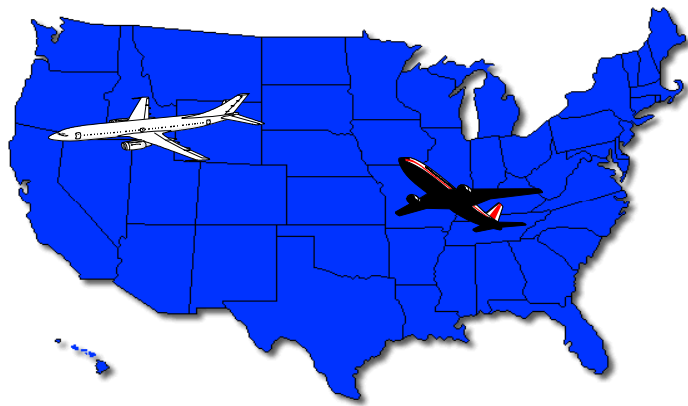
- ◆ Analyze systems before they are built
 - **Reduce number of design mistakes**
 - **Optimize design**
- ◆ Analyze operational systems
- ◆ Create virtual environments for training, entertainment

Simulation Taxonomy



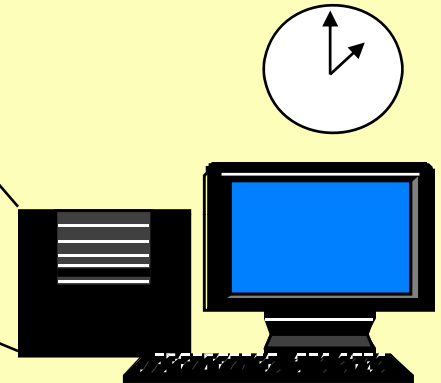
Time

- *physical system*: the actual or imagined system being modeled
- *simulation*: a system that emulates the behavior of a physical system



physical system

```
main()
{ ...
  double clock;
  ...
}
```



simulation

- ◆ *physical time*: time in the physical system
 - **Noon, December 31, 1999 to noon January 1, 2000**
- ◆ *simulation time*: representation of physical time within the simulation
 - **floating point values in interval [0.0, 24.0]**
- ◆ *wallclock time*: time during the execution of the simulation, usually output from a hardware clock
 - **9:00 to 9:15 AM on September 10, 1999**

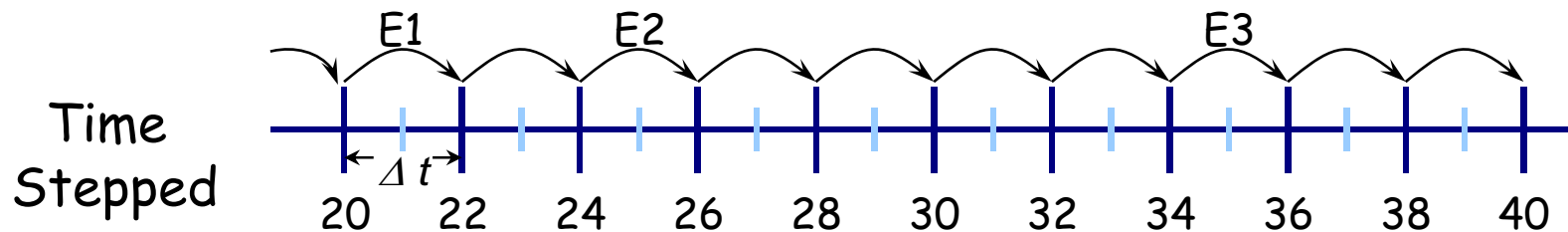
Time-advance Mechanisms

- ◆ **Simulation clock:** A variable that keeps the current value of (simulated) time in the model
 - Usually there is no relation between simulated time and (real) time needed to run a model on a computer
- ◆ Two approaches for time advancement of simulation clock
 - **Fixed-increment Time Advance (Time Stepped)**
 - **Next-event Time Advance (Event Driven)**

Time Stepped vs. Event Stepped

◆ Fixed-Increment Time Advance (time-stepped)

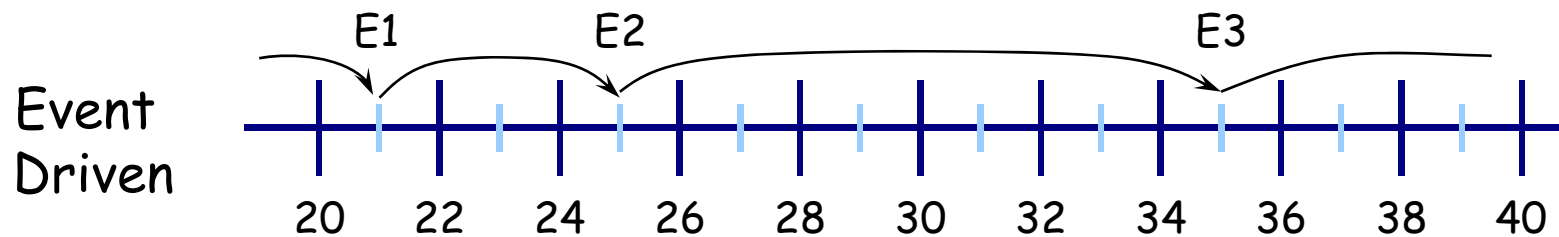
- clock is updated by the same time increment, Δt
- After each clock update, all events that were scheduled to occur during this interval are identified. Events are considered to occur at the end of the interval and the system state are updated accordingly.



Time Stepped vs. Event Stepped

◆ Event-Driven Advance

- time is advanced from the time of the current event to the time of the next scheduled event, i.e. simulation skips over periods of inactivity



Paced vs. Unpaced Execution

Modes of execution

- ◆ *As-fast-as-possible* execution (unpaced): no fixed relationship necessarily exists between advances in simulation time and advances in wallclock time
- ◆ *Real-time* execution (paced): each advance in simulation time is paced to occur in synchrony with an equivalent advance in wallclock time
- ◆ *Scaled real-time* execution (paced): each advance in simulation time is paced to occur in synchrony with S * an equivalent advance in wallclock time (e.g., 2x wallclock time)

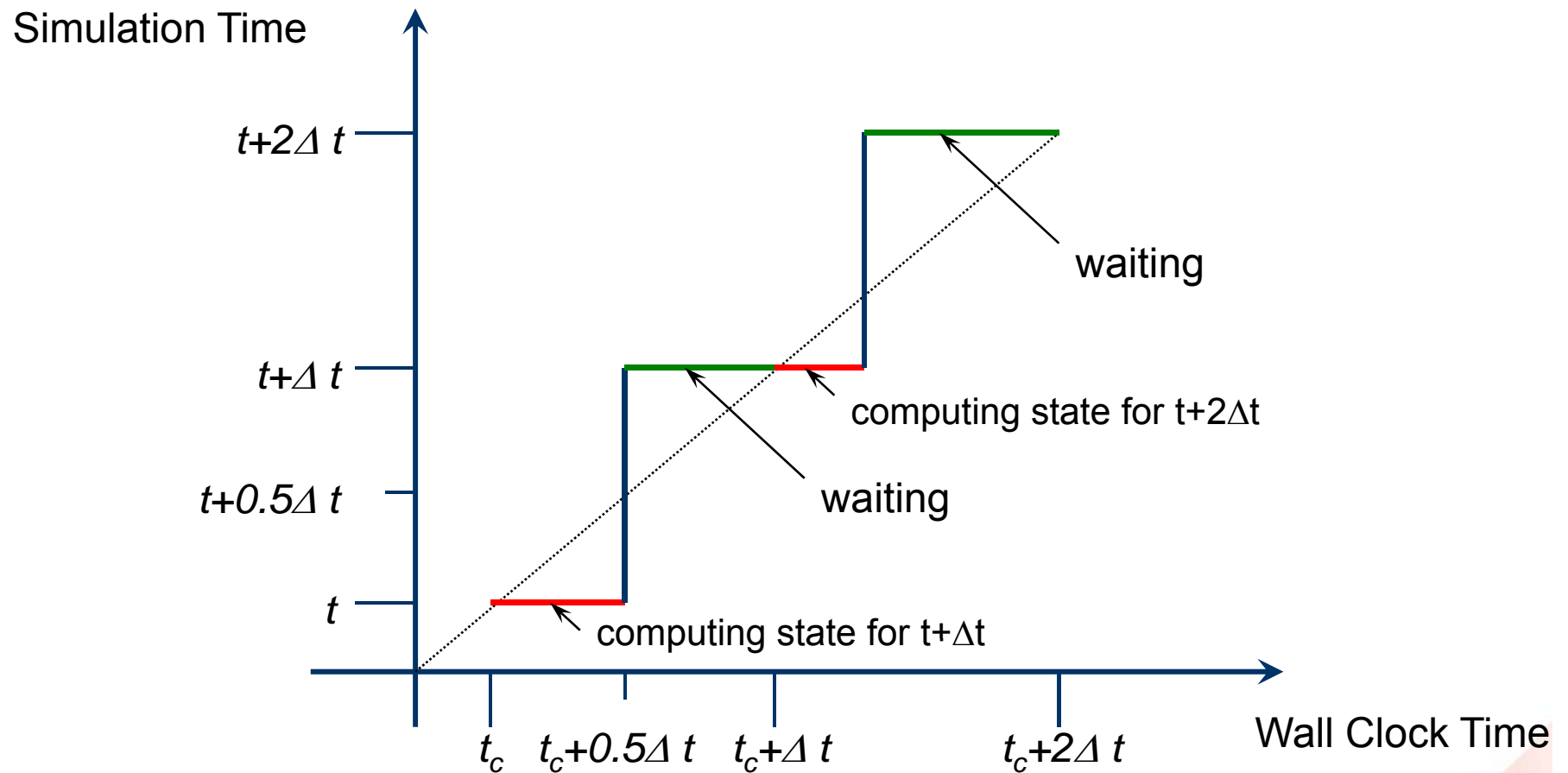
$$\text{Simulation Time} = W2S(W) = T_0 + S * (W - W_0)$$

W = wallclock time; S = scale factor

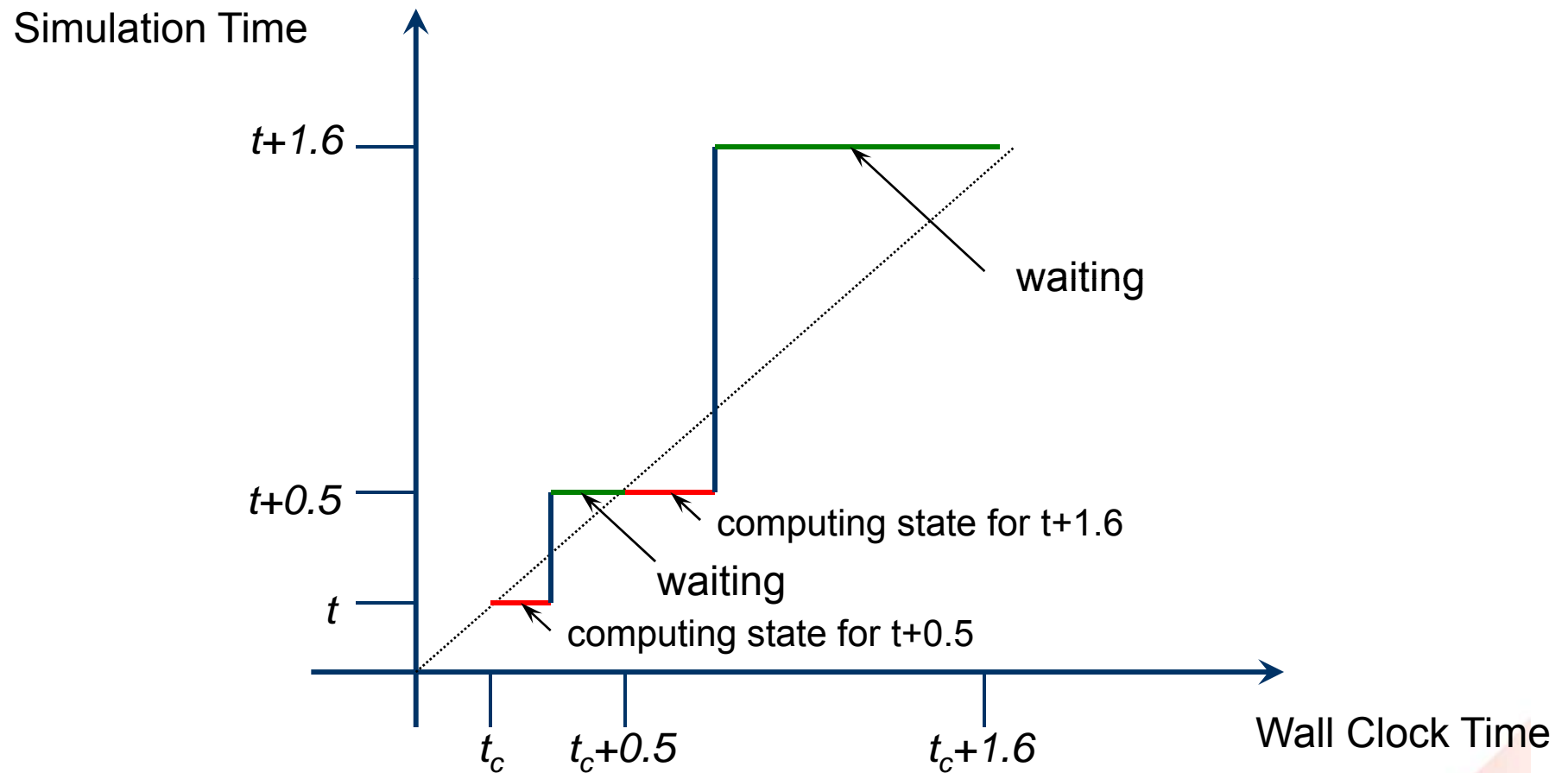
W_0 (T_0) = wallclock (simulation) time at start of simulation

(assume simulation and wallclock time use same time units)

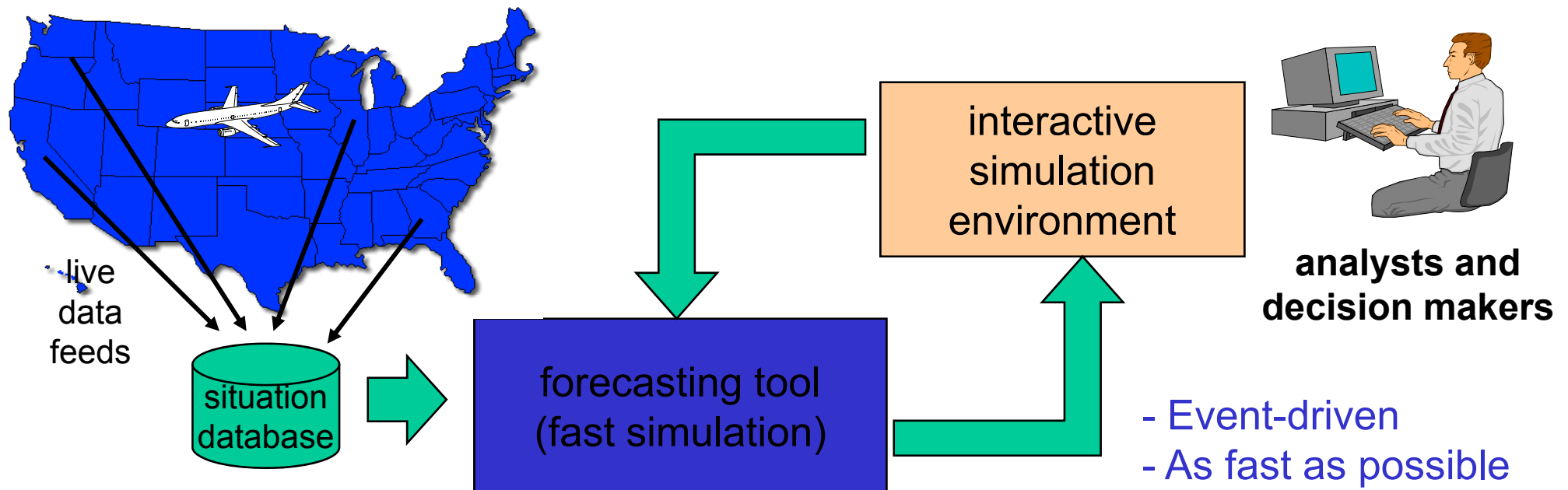
Paced Execution (Time Stepped)



Paced Execution (Event Driven)



Analytical Simulation vs. Virtual Environments



◆ Simulation tool is used for fast analysis of alternate courses of action in time critical situations

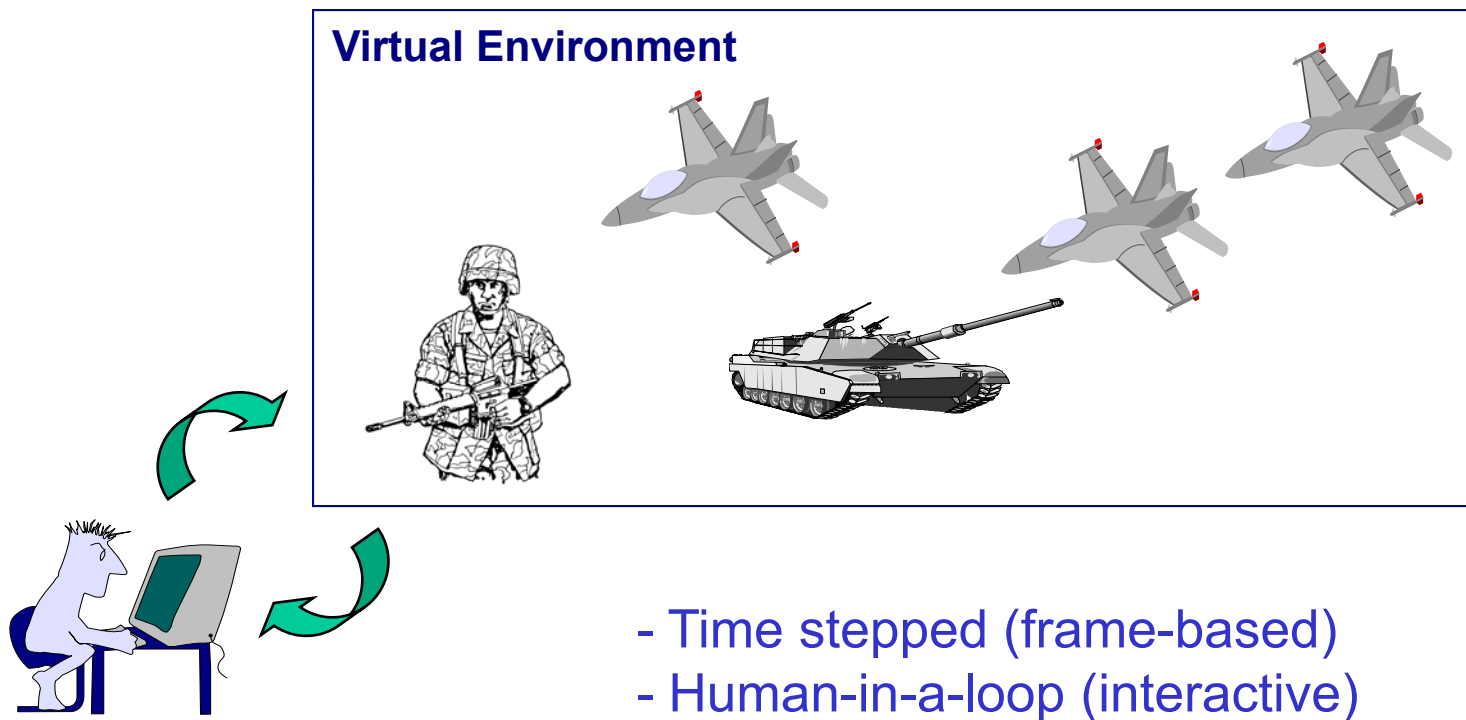
- Initialize simulation from situation database
- Faster-than-real-time execution to evaluate effect of decisions

Applications: air traffic control, battle management

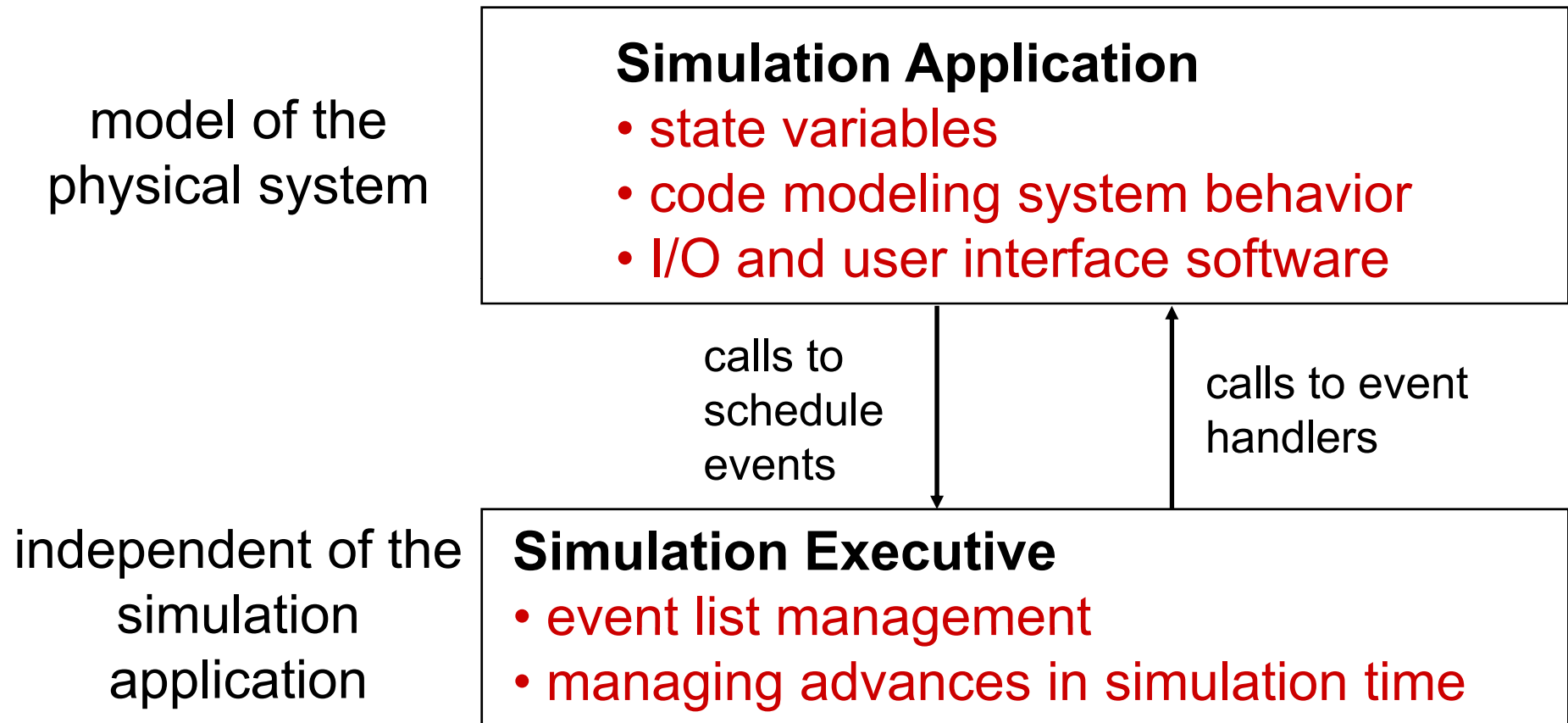
Simulation results may be needed in only seconds

Analytical Simulation vs. Virtual Environments

- ◆ Simulations are used in **virtual environments** to create dynamic computer generated entities

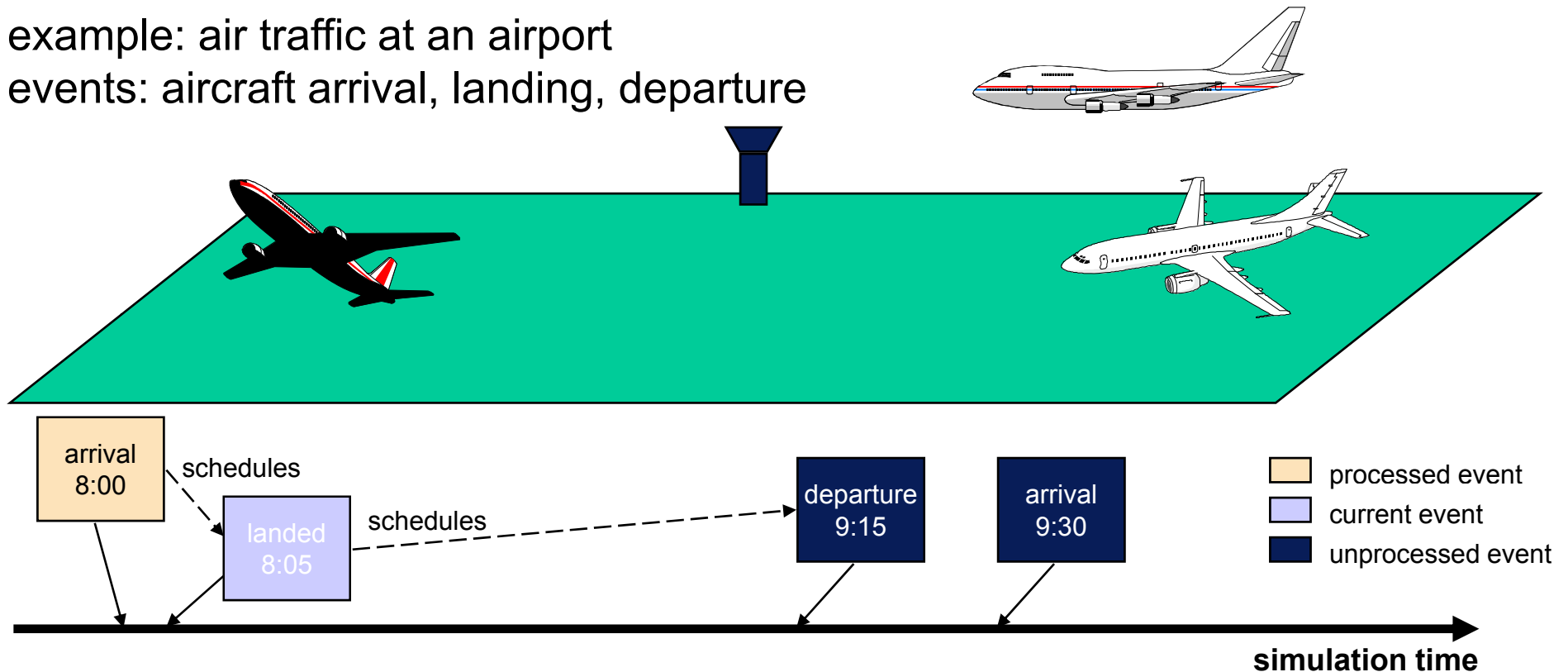


Discrete Event Simulation: Model and Execution



Model and Execution – An Example

example: air traffic at an airport
events: aircraft arrival, landing, departure



- ◆ Unprocessed events are stored in a pending event list
- ◆ Events are processed in time stamp order

Event-Oriented World View

Event Handler Procedures

state Variables

Integer: InTheAir;
Integer: OnTheGround;
Boolean: RunwayFree;

Arrival Event

```
{  
  ...  
}
```

Landed Event

```
{  
  ...  
}
```

Departure Event

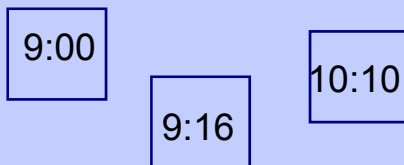
```
{  
  ...  
}
```

Simulation Application

Simulation Executive

Now = 8:45

Pending Event List (PEL)



Event Processing Loop

While (simulation not finished)

E = smallest time stamp event in PEL

Remove E from PEL

Now := time stamp of E

call event handler procedure

Example: Air Traffic at An Airport

Model aircraft arrivals and departures, arrival queueing

Single runway for incoming aircraft, ignore departure queueing

- ◆ **R** = time runway is used for each landing aircraft (constant)
- ◆ **G** = time required on the ground before departing (constant)

State:

- ◆ **Now**: current simulation time
- ◆ **InTheAir**: number of aircraft landing or waiting to land
- ◆ **OnTheGround**: number of landed aircraft
- ◆ **RunwayFree**: Boolean, true if runway available

Events:

- ◆ **Arrival**: denotes aircraft arriving in air space of airport
- ◆ **Landed**: denotes aircraft landing
- ◆ **Departure**: denotes aircraft leaving

Arrival Events

New aircraft arrives at airport. If the runway is free, it will begin to land. Otherwise, the aircraft must circle, and wait to land.

- ◆ **R** = time runway is used for each landing aircraft
- ◆ **G** = time required on the ground before departing
- ◆ **Now**: current simulation time
- ◆ **InTheAir**: number of aircraft landing or waiting to land
- ◆ **OnTheGround**: number of landed aircraft
- ◆ **RunwayFree**: Boolean, true if runway available

Arrival Event:

InTheAir := InTheAir+1;

If (RunwayFree)

RunwayFree:=FALSE;

Schedule Landed event @ Now + R;

Landed Event

An aircraft has completed its landing.

- ◆ **R** = time runway is used for each landing aircraft
- ◆ **G** = time required on the ground before departing
- ◆ **Now**: current simulation time
- ◆ **InTheAir**: number of aircraft landing or waiting to land
- ◆ **OnTheGround**: number of landed aircraft
- ◆ **RunwayFree**: Boolean, true if runway available

Landed Event:

```
InTheAir:=InTheAir-1;  
OnTheGround:=OnTheGround+1;  
Schedule Departure event @ Now + G;  
If (InTheAir>0) Schedule Landed event @ Now + R;  
Else RunwayFree := TRUE;
```

Departure Event

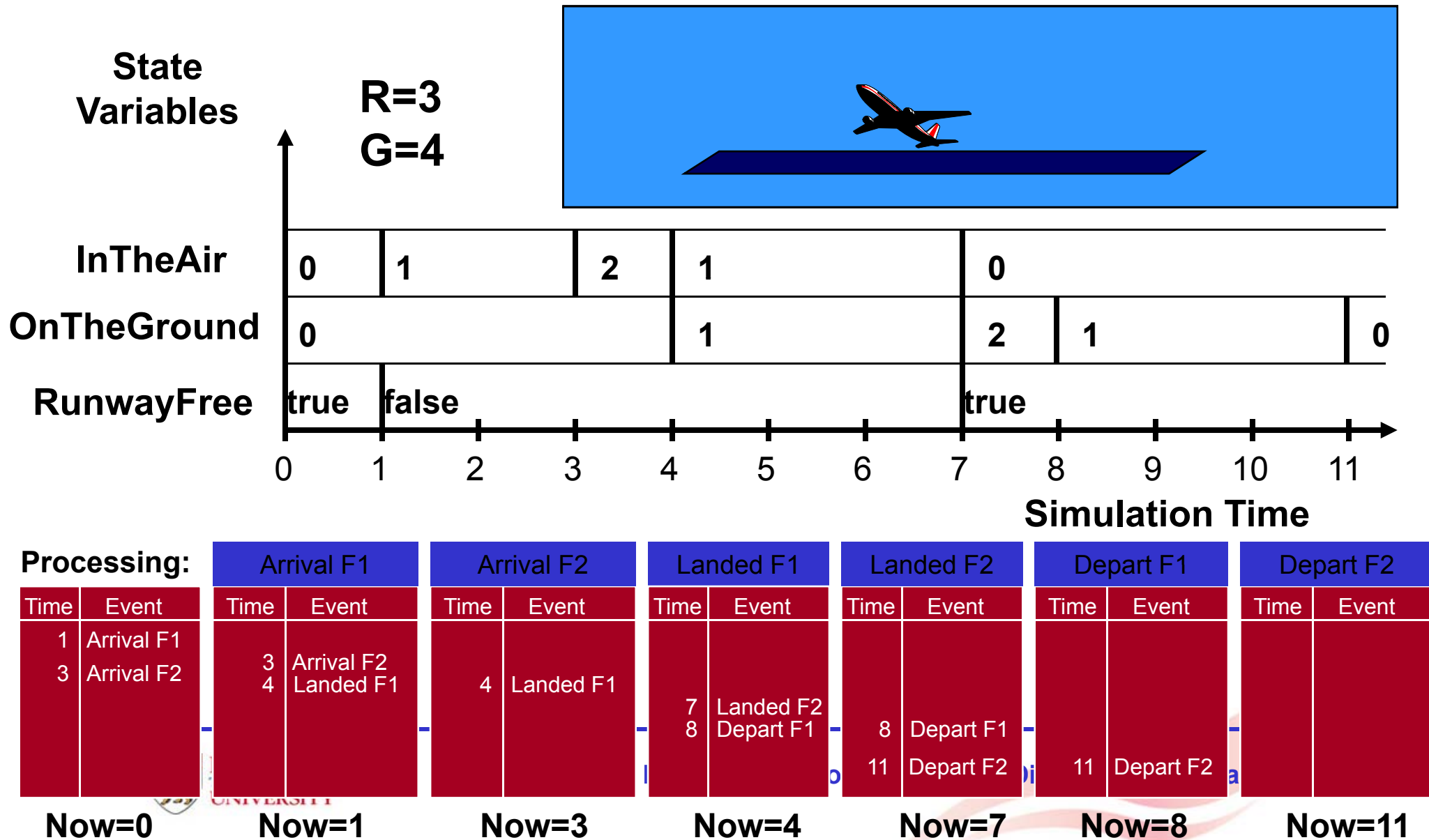
An aircraft now on the ground departs for a new destination.

- ◆ **R** = time runway is used for each landing aircraft
- ◆ **G** = time required on the ground before departing
- ◆ **Now**: current simulation time
- ◆ **InTheAir**: number of aircraft landing or waiting to land
- ◆ **OnTheGround**: number of landed aircraft
- ◆ **RunwayFree**: Boolean, true if runway available

Departure Event:

OnTheGround := **OnTheGround** - 1;

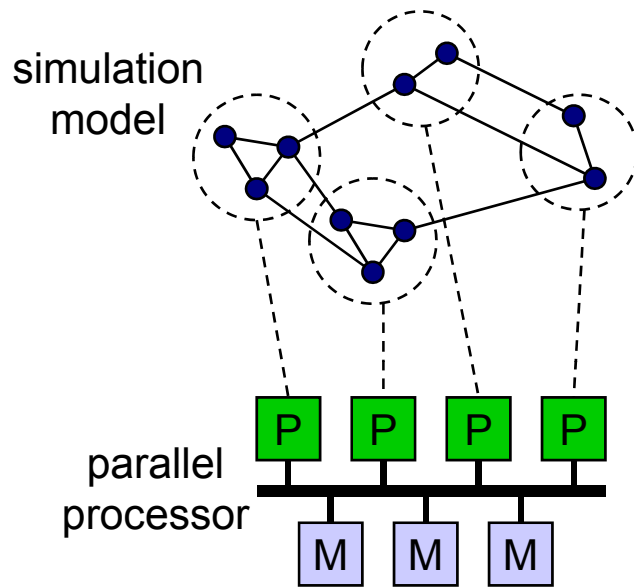
Execution Example – Event Oriented View



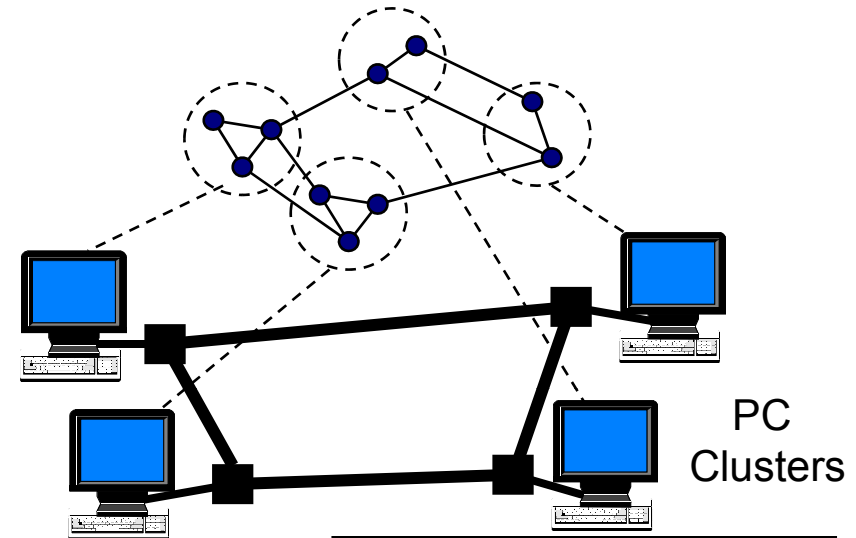
Outline – Introduction

- ◆ **Basic Concepts**
 - **Modelling and Simulation**
 - **Time Advancement Mechanism**
 - **Model and Execution**
- ◆ **Parallel and Distributed Simulation**
 - **What is Parallel and Distributed Simulation?**
 - **Motivation for Parallel and Distributed Simulation**
- ◆ **Simulation Decomposition**
 - **Logical Processes**
 - **Time Stamped Messages**
- ◆ **Causality**
 - **Local Causality Constraint**
 - **Synchronization Problem**
 - **Overview of Synchronization Protocols**

Parallel vs. Distributed Simulation

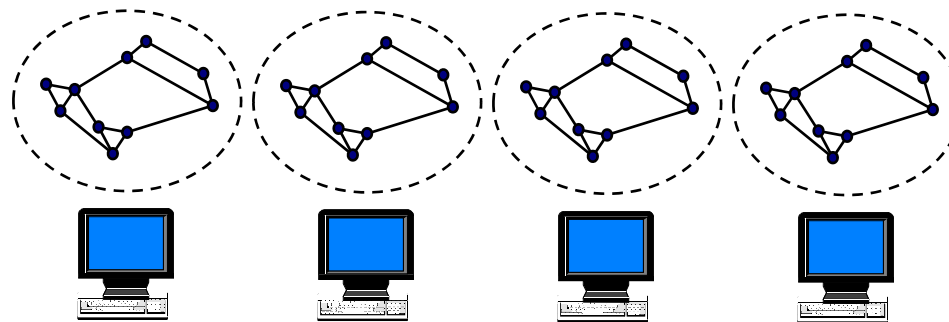


Parallel simulation involves the execution of a *single* simulation program on a collection of *tightly* coupled processors (e.g., a shared memory multiprocessor).



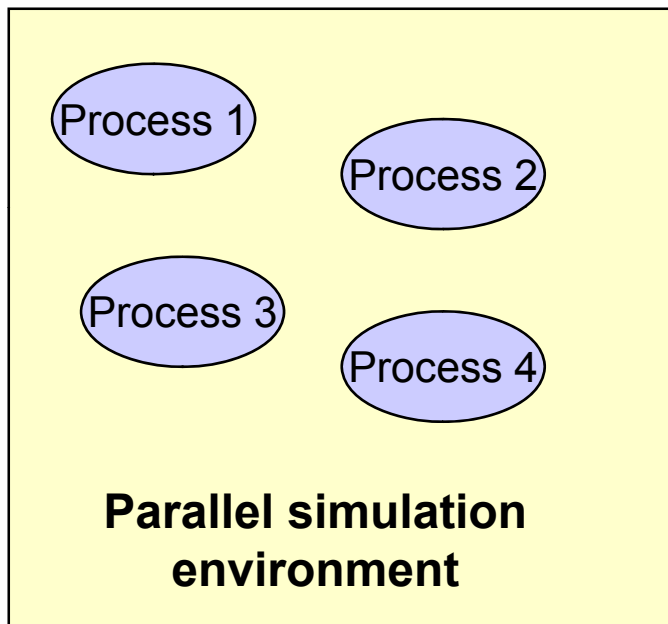
Distributed simulation involves the execution of a *single* simulation program on a collection of *loosely* coupled processors (e.g., PCs interconnected by a LAN or WAN).

Replicated trials involves the execution of *several*, independent simulations concurrently on different processors



Standalone vs. Federated Simulation Systems

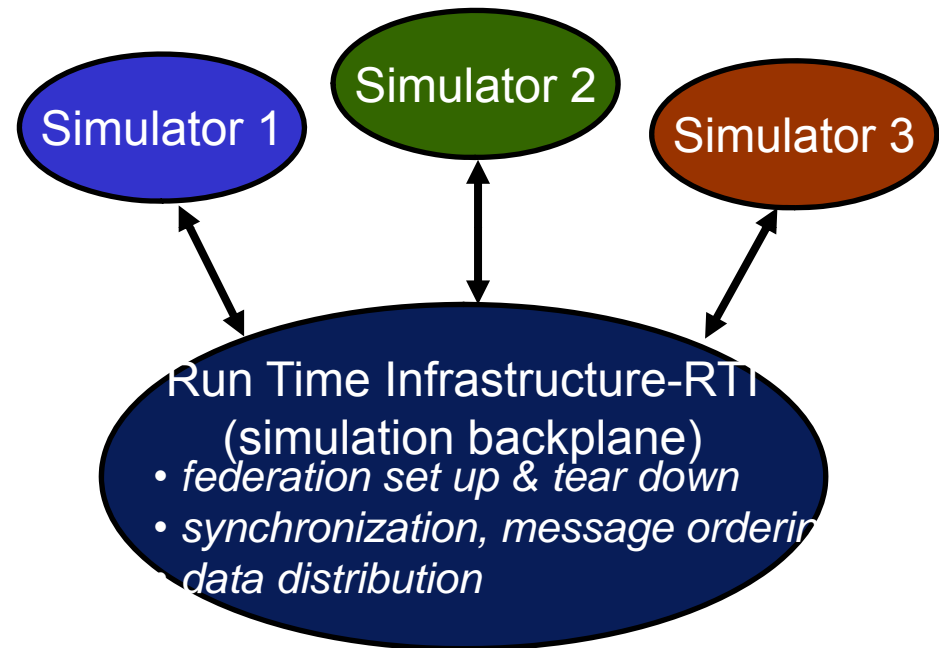
Stand-Alone Simulation System



Homogeneous programming environment

- *simulation language*

Federated Simulation Systems



Interconnect autonomous, heterogeneous simulators

- *interface to RTI software*

Reasons to Use Parallel and Distributed Simulations

- ◆ Enable the execution of time consuming simulations that could not otherwise be performed (e.g., simulation of the Internet)
 - **Reduce model execution time (proportional to # processors)**
 - **Ability to run larger models (more memory)**
- ◆ Enable simulation to be used as a forecasting tool in time critical decision making processes (e.g., air traffic control)
 - **Initialize simulation to current system state**
 - **Faster than real time execution for what-if experimentation**
 - **Simulation results may be needed in seconds**
- ◆ Create distributed virtual environments, possibly including users at distant geographical locations (e.g., training, entertainment)
 - **Real-time execution capability**
 - **Scalable performance for many users & simulated entities**

Enable Simulation of Big Models

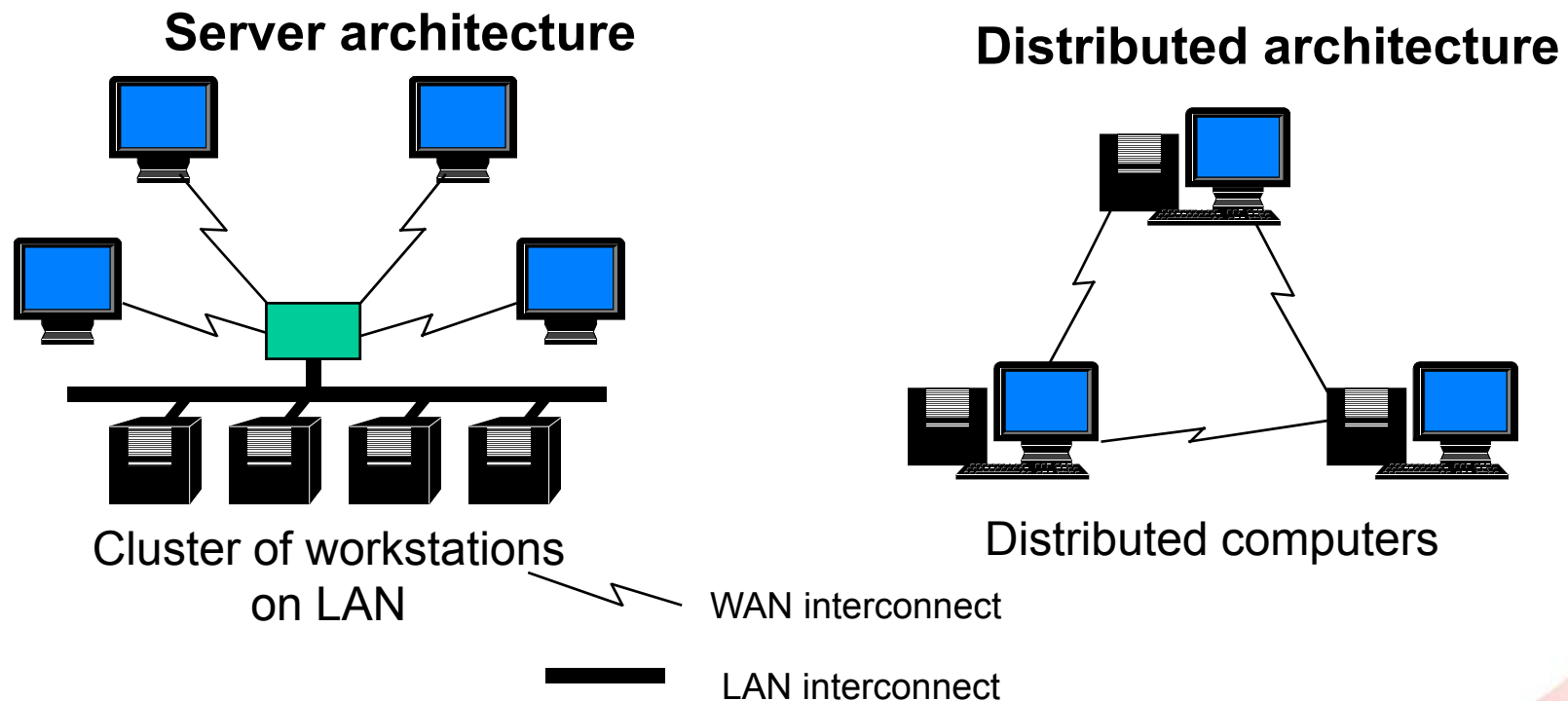
- ◆ Cell level simulation of an ATM (packet) network
 - Simulate one hour of network operation
 - Network with 1000 links
 - 155 Mbits/second links @ 20% utilization
 - 53 byte packets (cells)
 - One simulator event per cell transmission (link)
 - 500 K events / second simulator speed

150 hours for a single simulation run!

- ◆ Larger, more complex networks?
 - Next Generation Internet: Million nodes
- ◆ Higher link bandwidths

Geographically Distributed Users/Resources

- ◆ Geographically distributed users and/or resources are sometime needed
 - Interactive games over the Internet
 - Specialized hardware or databases



Outline – Introduction

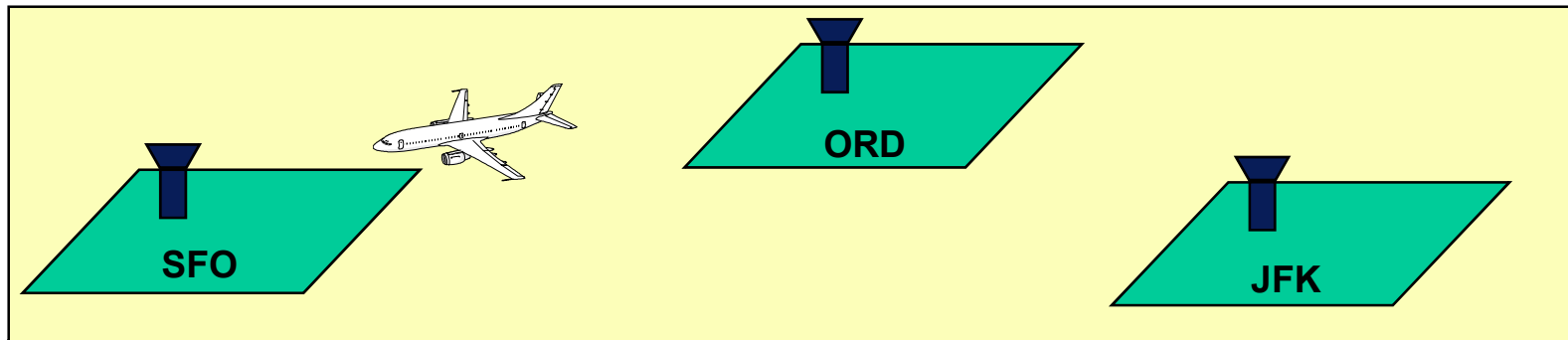
- ◆ **Basic Concepts**
 - **Modelling and Simulation**
 - **Time Advancement Mechanism**
 - **Model and Execution**
- ◆ **Parallel and Distributed Simulation**
 - **What is Parallel and Distributed Simulation?**
 - **Motivation for Parallel and Distributed Simulation**
- ◆ **Simulation Decomposition**
 - **Logical Processes**
 - **Time Stamped Messages**
- ◆ **Causality**
 - **Local Causality Constraint**
 - **Synchronization Problem**
 - **Overview of Synchronization Protocols**

Simulation Decomposition

- ◆ Extend example to model a network of airports
 - Encapsulate each airport simulator in a **logical process**
 - Logical processes can schedule events (**send messages**) to other logical processes
- ◆ Physical system
 - Collection of interacting physical processes
- ◆ Simulation
 - Collection of logical processes (LPs)
 - Each LP models a physical process
 - Interactions between physical processes modeled by scheduling events between LPs

Simulation Decomposition

Physical system



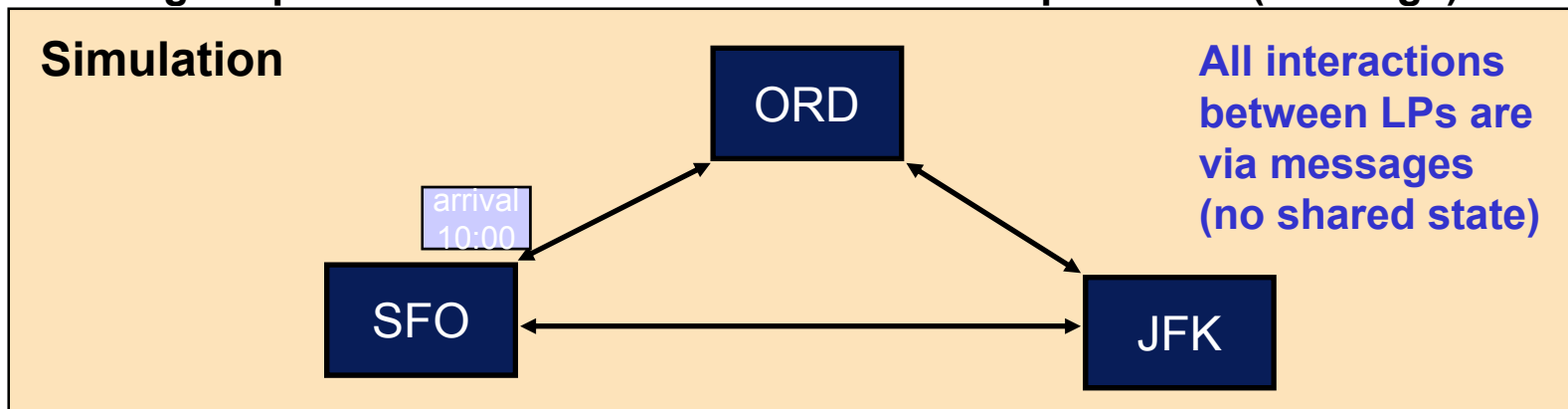
physical process

interactions among physical processes

logical process

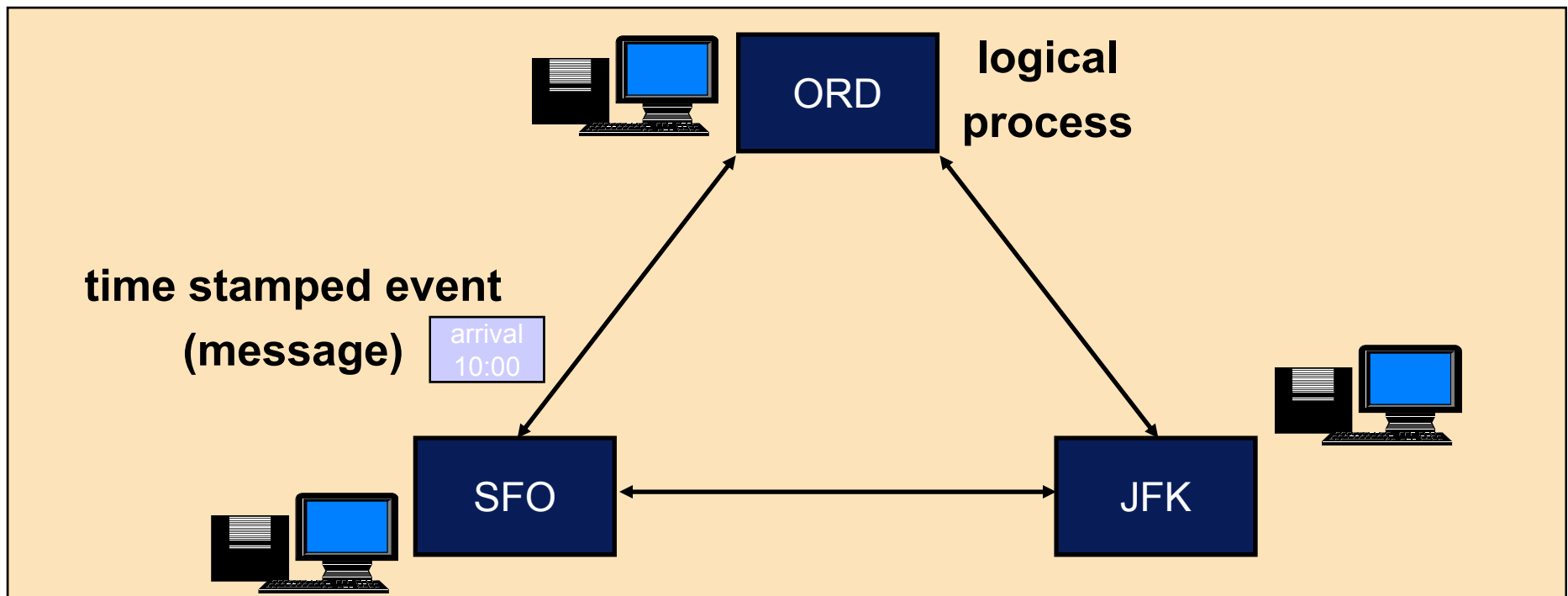
time stamped event (message)

Simulation



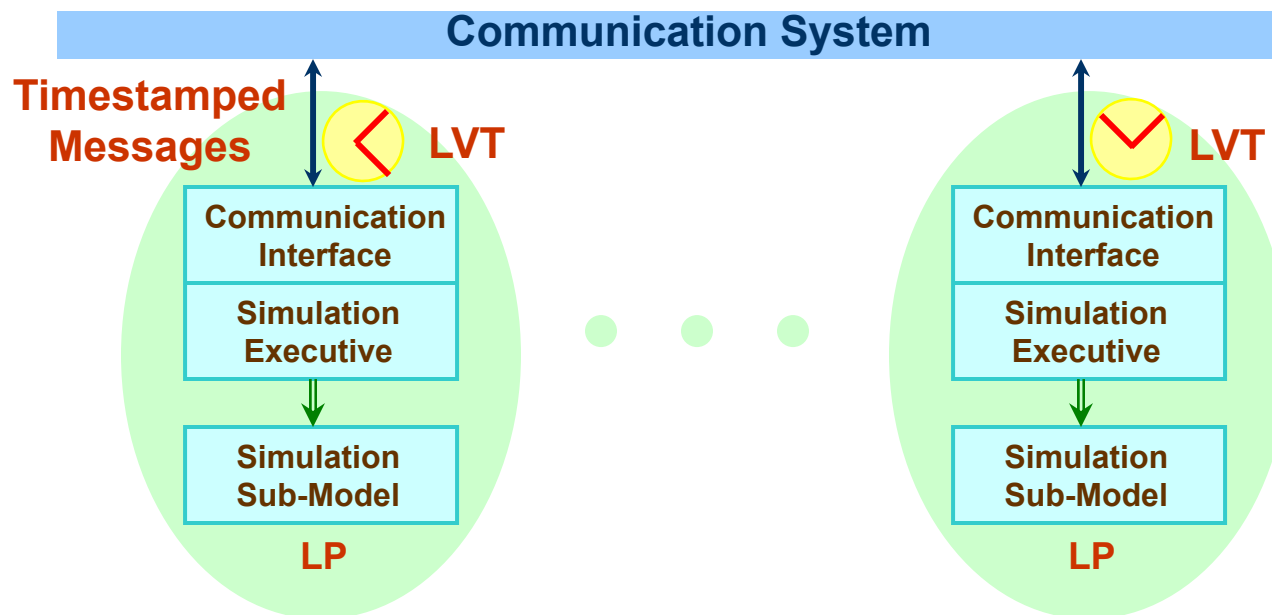
Simulation Decomposition

- ◆ Logical Process (LP) paradigm appears well suited to concurrent execution
- ◆ Map LPs to different processors
 - **Multiple LPs per processor possible**
- ◆ Communication via message passing
 - **All interactions via messages**
 - **No shared state variables**



Simulation Decomposition

- ◆ A simulation application is decomposed into a set of concurrently executing **Logical Processes** (LPs)
- ◆ Each Logical Process has its own simulation time or Local Virtual Time (LVT)

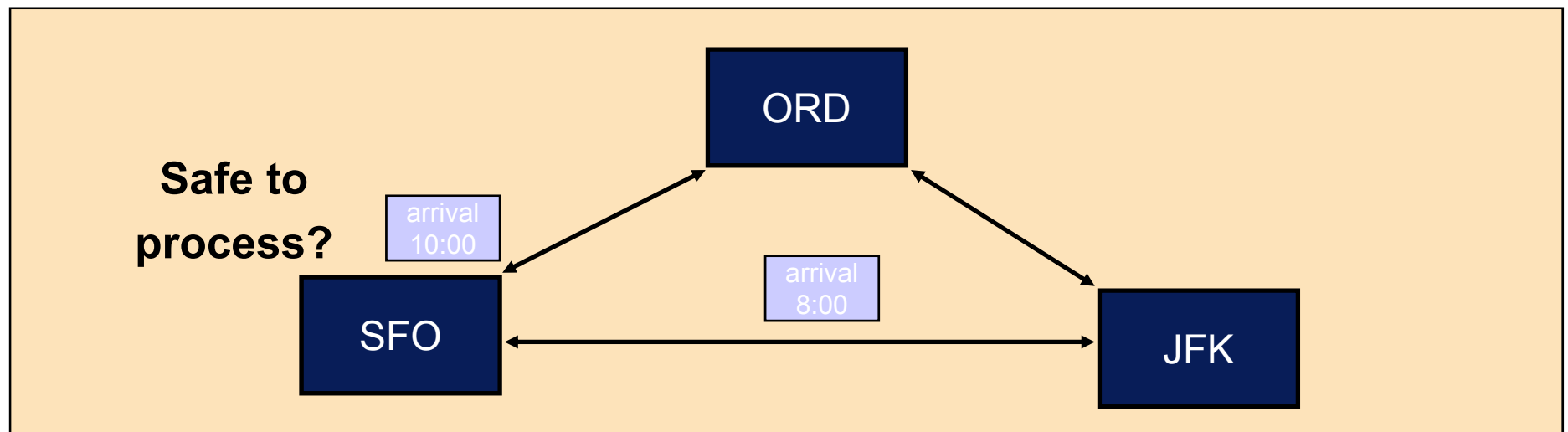


Outline – Introduction

- ◆ **Basic Concepts**
 - **Modelling and Simulation**
 - **Time Advancement Mechanism**
 - **Model and Execution**
- ◆ **Parallel and Distributed Simulation**
 - **What is Parallel and Distributed Simulation?**
 - **Motivation for Parallel and Distributed Simulation**
- ◆ **Simulation Decomposition**
 - **Logical Processes**
 - **Time Stamped Messages**
- ◆ **Causality**
 - **Local Causality Constraint**
 - **Synchronization Problem**
 - **Overview of Synchronization Protocols**

Causality

- ◆ Different LPs may execute events at different points in simulation time
- ◆ Messages may not arrive at an LP in simulation time order
- ◆ Future may be simulated before past (**causality error**)



Local Causality Constraint

◆ Local Causality Constraint

- Events must be executed in non decreasing time stamp order at each LP
- The local causality constraint is sufficient to ensure that the parallel simulation **will produce exactly the same results as a sequential execution** where all events are processed in time stamp order (ignoring problems of events with the same time stamp)

◆ Synchronization Problem

- An algorithm is needed to ensure each LP processes events in non decreasing time stamp order

Overview of Synchronization Protocols

- ◆ Conservative synchronization: avoid violating the local causality constraint (wait until it is safe)
 - **Deadlock avoidance using null messages (Chandy/Misra/Bryant)**
 - **Synchronous algorithms (execute in “rounds”)**
- ◆ Optimistic synchronization: allow violations of local causality to occur, but detect them at runtime and recover using a rollback mechanism
 - **Time Warp (Jefferson)**
 - **Numerous other approaches**

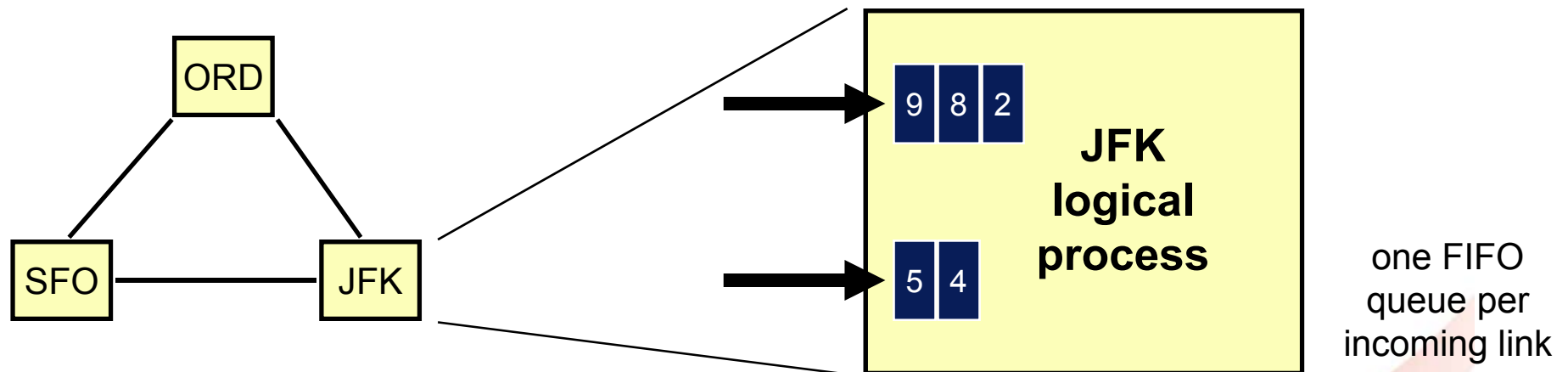
Outline – Conservative Synchronization

- ◆ A Simple Conservative Algorithm
 - **Deadlock**
- ◆ Null Message Algorithm
 - **Deadlock Avoidance Using Null messages**
 - **The Time Creep Problem**
 - **Lookahead**
- ◆ Synchronous Algorithms
 - **Implementation of Barrier Mechanisms**
 - **Computing LBTS**
 - **Transient Messages**
 - **Performance Improvements**

A Simple Conservative Algorithm

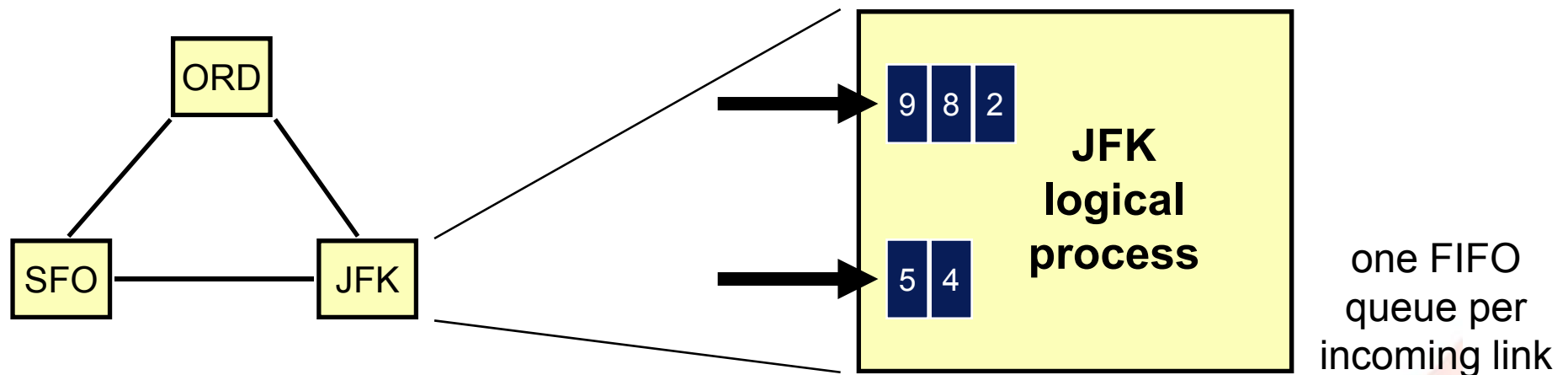
◆ Assumptions

- **Logical Processes (LPs) exchange time stamped events (messages)**
- **Static network topology, no dynamic creation of LPs**
- **Messages sent on each link are sent in time stamp order**
- **Network provides reliable delivery, preserves order**



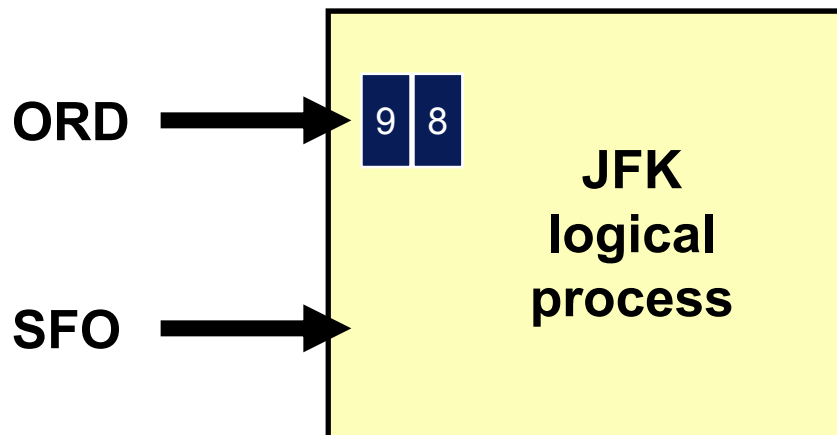
A Simple Conservative Algorithm

- ◆ Goal: Ensure LP processes events in time stamp order
- ◆ **Observation:** The above assumptions imply the time stamp of the last message received on a link is a **lower bound on the time stamp (LBTS)** of subsequent messages received on that link



A Simple Conservative Algorithm

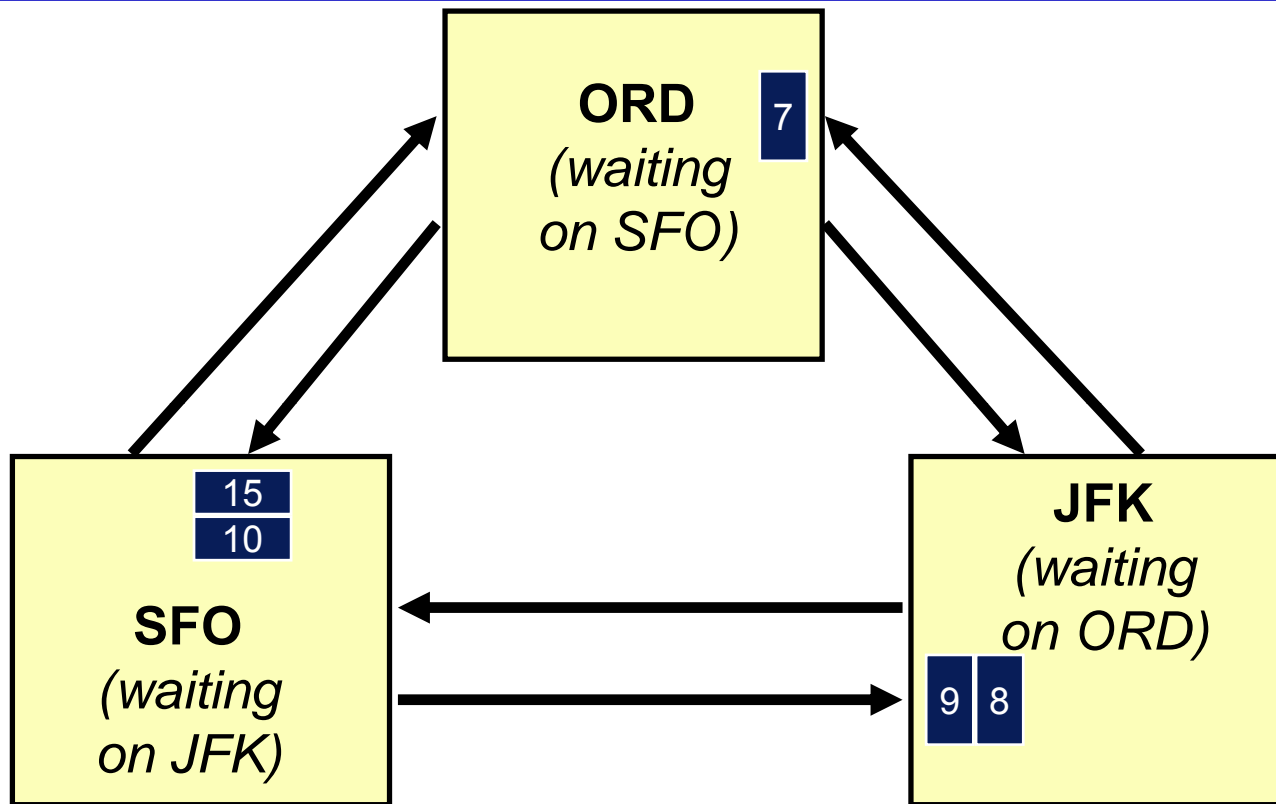
```
WHILE (simulation is not over)
    wait until each FIFO contains at least one message
    remove smallest time stamped event from its FIFO
    process that event
END-LOOP
```



- process time stamp 2 event
- process time stamp 4 event
- process time stamp 5 event
- wait until message is received from SFO

Observation: Algorithm is prone to deadlock!

Deadlock Example



A cycle of LPs forms where each is waiting on the next LP in the cycle
No LP can advance; the simulation is deadlocked

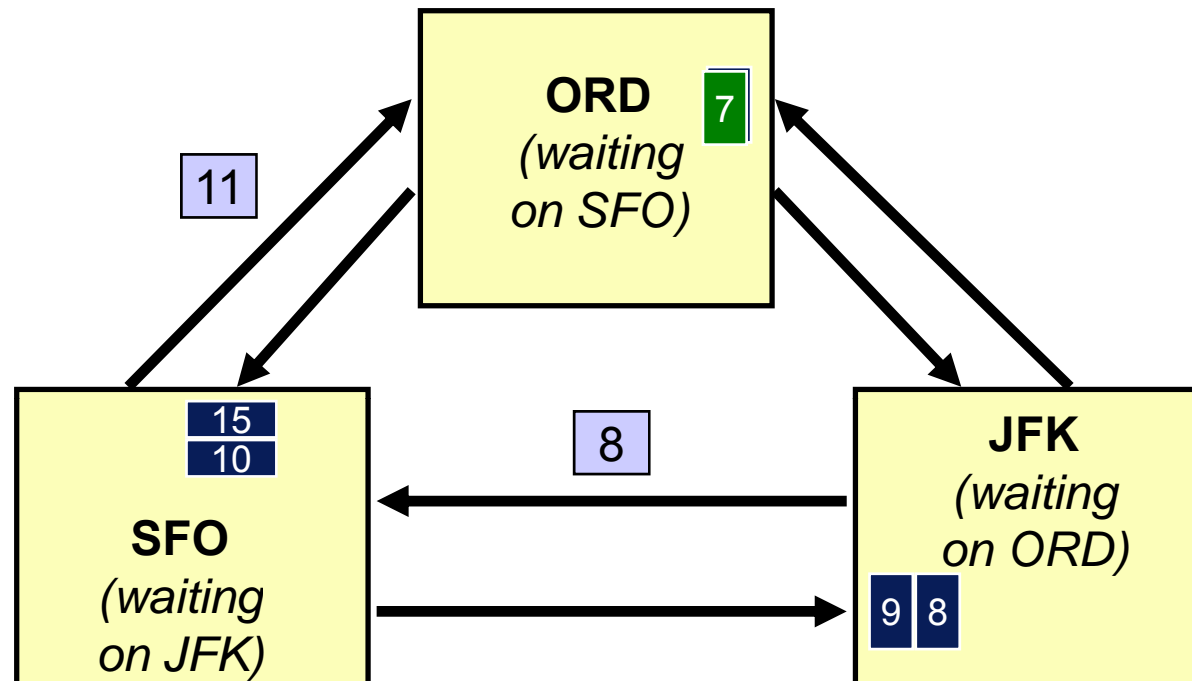
Outline – Conservative Synchronization

- ◆ A Simple Conservative Algorithm
 - **Deadlock**
- ◆ Null Message Algorithm
 - **Deadlock Avoidance Using Null messages**
 - **The Time Creep Problem**
 - **Lookahead**
- ◆ Synchronous Algorithms
 - **Implementation of Barrier Mechanisms**
 - **Computing LBTS**
 - **Transient Messages**
 - **Performance Improvements**

Null Message Algorithm (Chandy/Misra and Bryant)

- ◆ Break deadlock: each LP send “null” messages indicating a lower bound on the time stamp of future messages
- ◆ A null message with timestamp T sent from LP_i to LP_j is a promise by LP_i that it will not later send a message to LP_j carrying a timestamp smaller than T
- ◆ Null message algorithm relies upon a lookahead ability
 - In our example, assume minimum delay between airports is 3 units of time – this minimum delay is called **“lookahead”**
 - “ORD at simulation time 5, minimum transit time between airports is 3, so the next message sent by ORD must have a time stamp of **at least 8**”

Deadlock Avoidance Using Null Messages



JFK initially at time 5

- JFK sends null message to SFO with time stamp 8
- SFO sends null message to ORD with time stamp 11
- ORD may now process message with time stamp 7

Deadlock Avoidance Using Null Messages

Null Message Algorithm (executed by each LP):

Goal: Ensure events are processed in time stamp order and avoid deadlock

WHILE (simulation is not over)

wait until each FIFO contains at least one message

remove smallest time stamped event from its FIFO

process that event

send null messages to neighboring LPs with time stamp indicating a lower bound on future messages sent to that LP (current time plus lookahead)

END-LOOP

The null message algorithm relies on a “lookahead” ability.

Deadlock Avoidance Using Null Messages

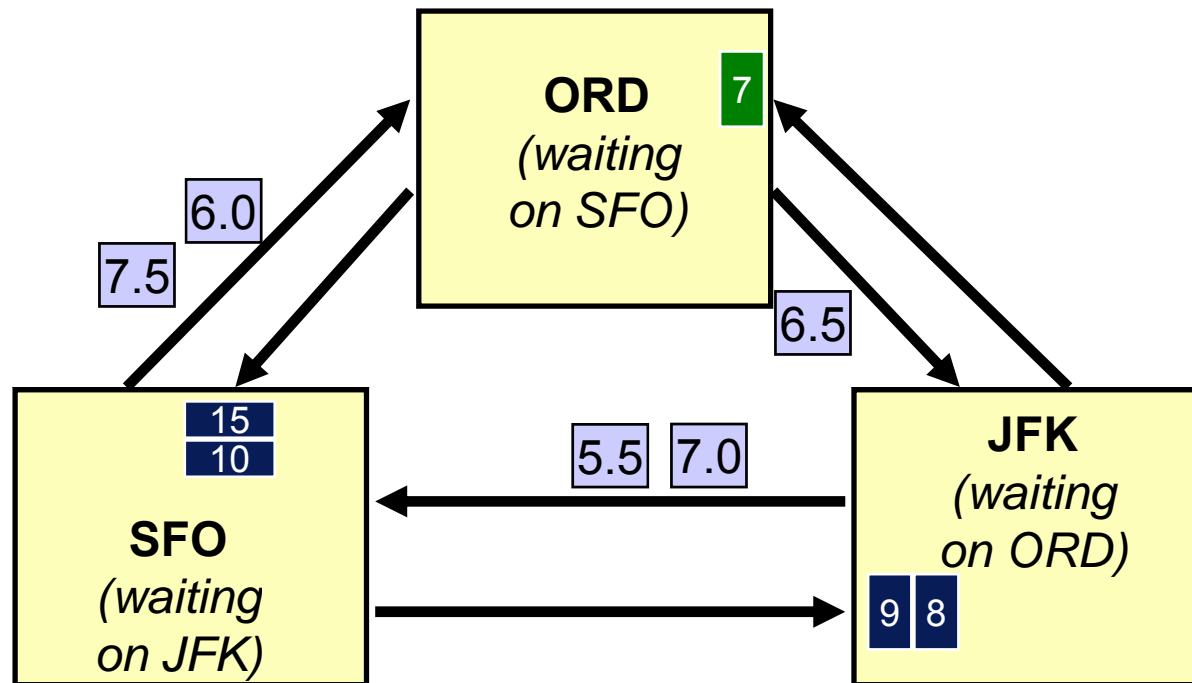
◆ Basic Algorithm

- LP sends null message whenever it advances its Local Virtual Time (LVT)
- Sender initiated null message algorithm
 - ◆ Propagates time information quickly but some null messages may be redundant

◆ Variation

- LP requests null message when FIFO becomes empty
- Receiver initiated null message algorithm
 - ◆ Fewer null messages, but possible delay to get time information

The Time Creep Problem



Null messages:

JFK: timestamp = 5.5

SFO: timestamp = 6.0

ORD: timestamp = 6.5

JFK: timestamp = 7.0

SFO: timestamp = 7.5

ORD: process time stamp 7 message

Five null messages to process a single event!

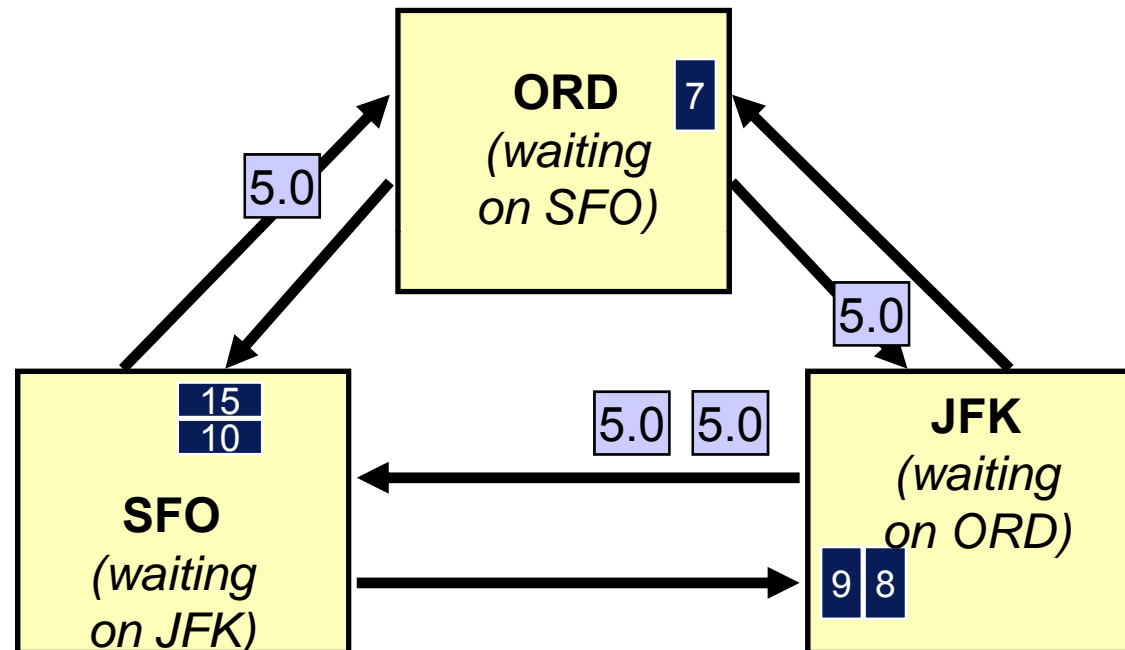
0.5

Assume minimum delay between airports is ~~3~~ units of time
JFK initially at time 5

Many null messages if minimum flight time is small!

Livelock Can Occur!

Suppose the minimum delay between airports is zero!



Livelock: un-ending cycle of null messages where no LP can advance its simulation time

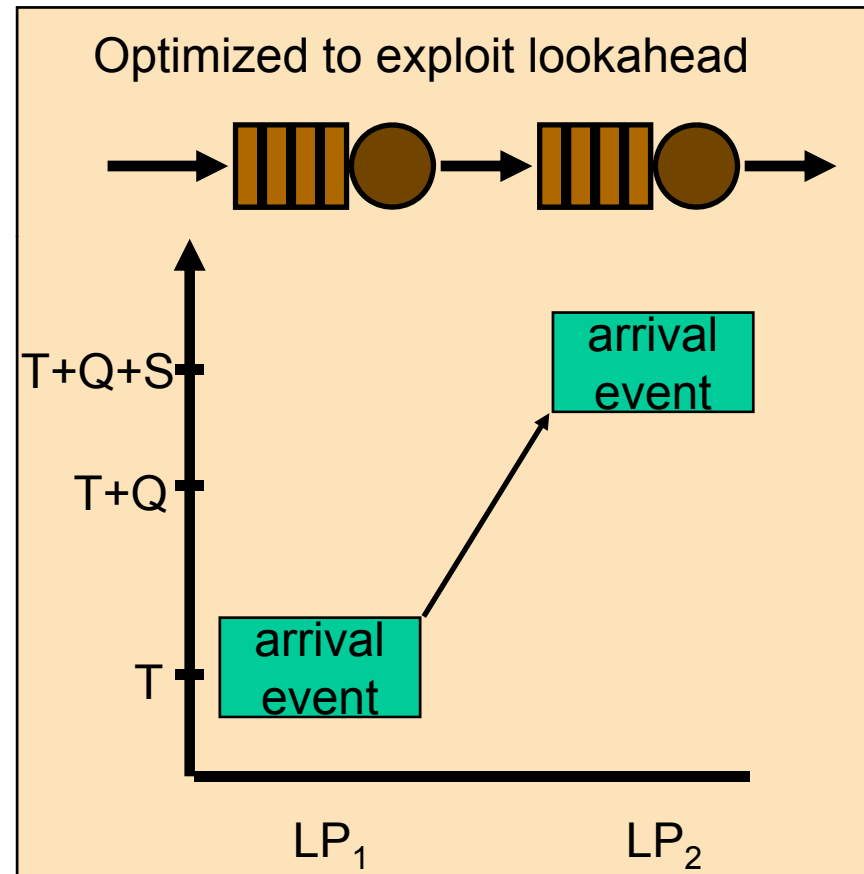
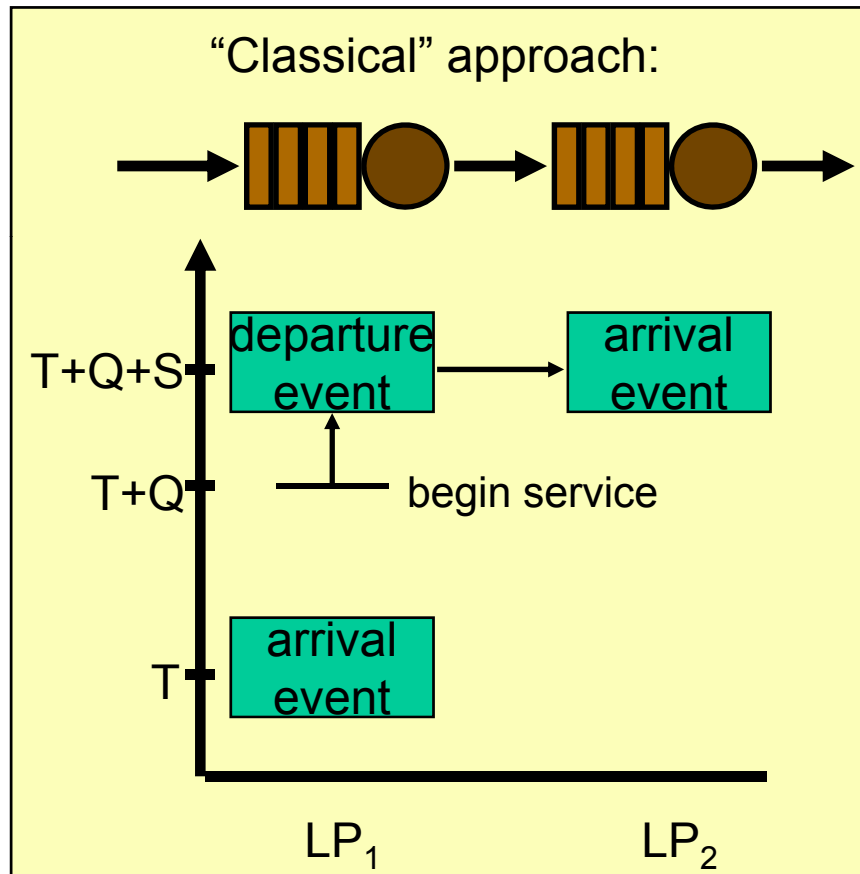
Lookahead

- ◆ Lookahead is a constraint on LP's behavior
 - If an LP is at simulation time T , and has a lookahead of L , then any message sent by that LP must will have a time stamp of at least $T+L$
- ◆ The degree to which the program can exploit lookahead is critical for good performance
- ◆ To avoid livelock, there cannot be a cycle where for each LP in the cycle, an incoming message with time stamp T results in a new message sent to the next LP in the cycle with time stamp T (zero lookahead cycle)

Exploiting Lookahead in Applications

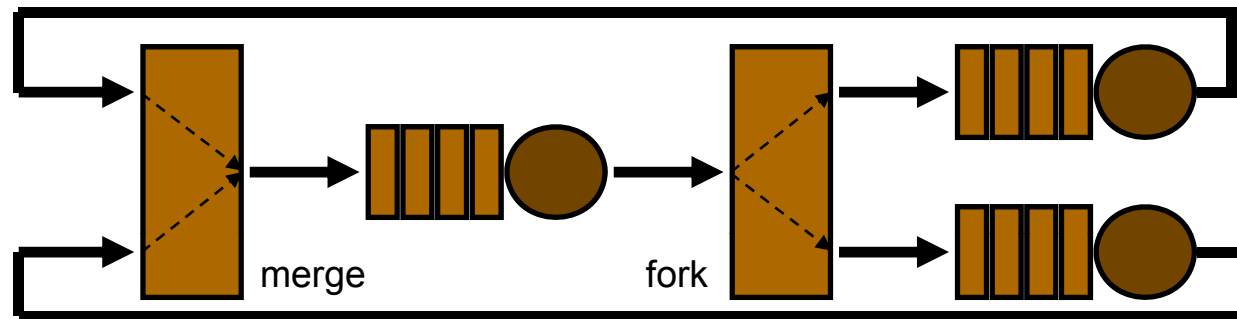
Example: Tandem first-come-first-serve queues

T = arrival time of job
 Q = waiting time in queue
 S = service time



Lookahead Affects Performance

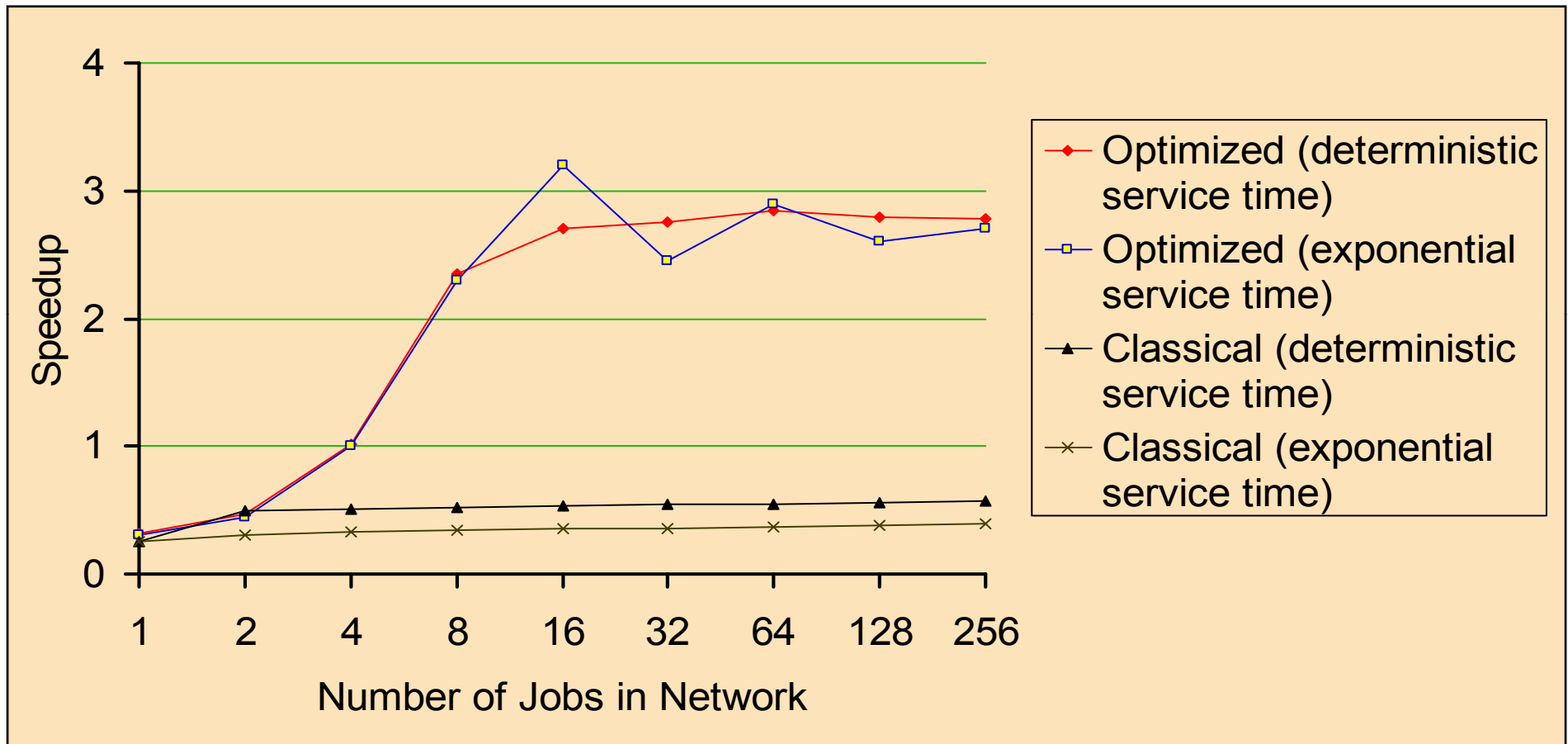
◆ Parallel Simulation of a Central Server Queueing Network



◆ Speedup:

- Sequential execution time / parallel execution time
- Measure of the increase in performance due to parallelism

Lookahead Affects Performance



Conservative Algorithm (5 processors)

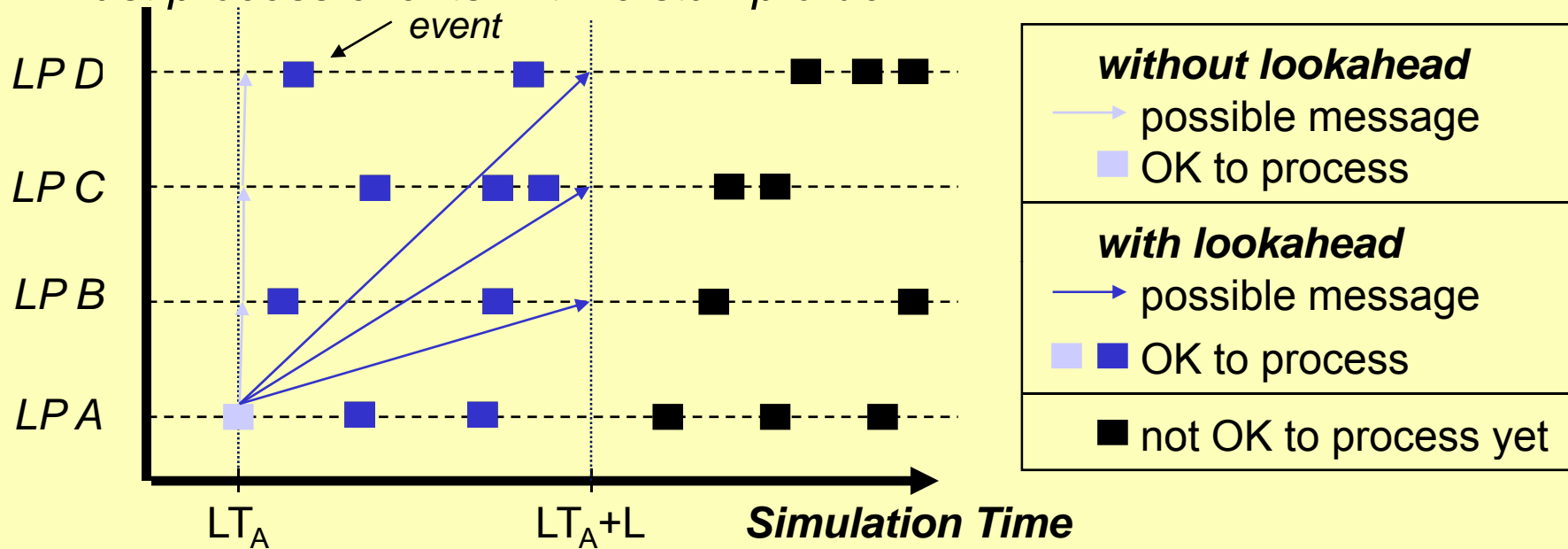
Lookahead in the Simulation Model

- ◆ Lookahead is clearly dependent on the simulation model
 - **Could be derived from physical constraints in the system being modeled, such as minimum simulation time for one entity to affect another**
 - **Examples:**
 - ◆ **An aircraft departing from one airport requires at least L units of time before it arrives at another airport**
 - ◆ **A weapon fired from a tank requires at least L units of time to reach another tank**
 - ◆ **Maximum speed of the tank places lower bound on how soon it can affect another entity**
- ◆ Time-stepped simulations implicitly use lookahead. Events in current time step are considered independent (and can be processed concurrently). New events are generated for the next time step, or later.

Why Lookahead is Important

problem: limited concurrency

each LP must process events in time stamp order



Each LP using logical time declares a lookahead value L ; e.g. the time stamp of any event generated by LP A must be $\geq LT_A + L$

- Lookahead is used in virtually all conservative synchronization protocols
- Essential to allow concurrent processing of events

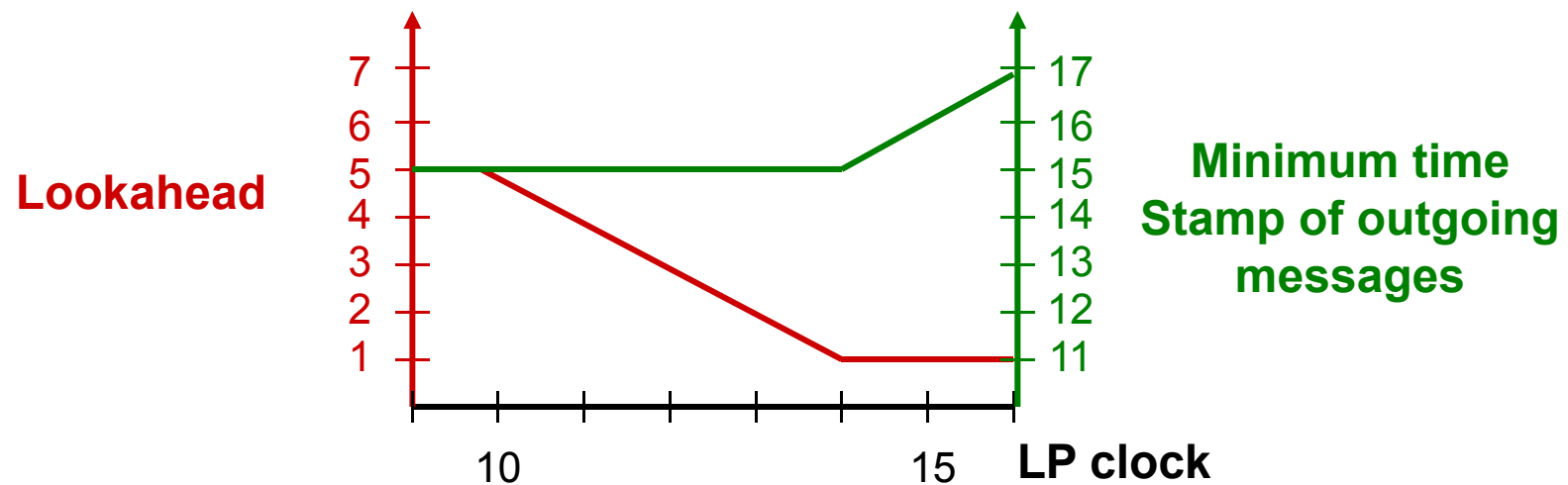
Lookahead is necessary to allow concurrent processing of events with different time stamps (unless optimistic event processing is used)

Changing Lookahead Values

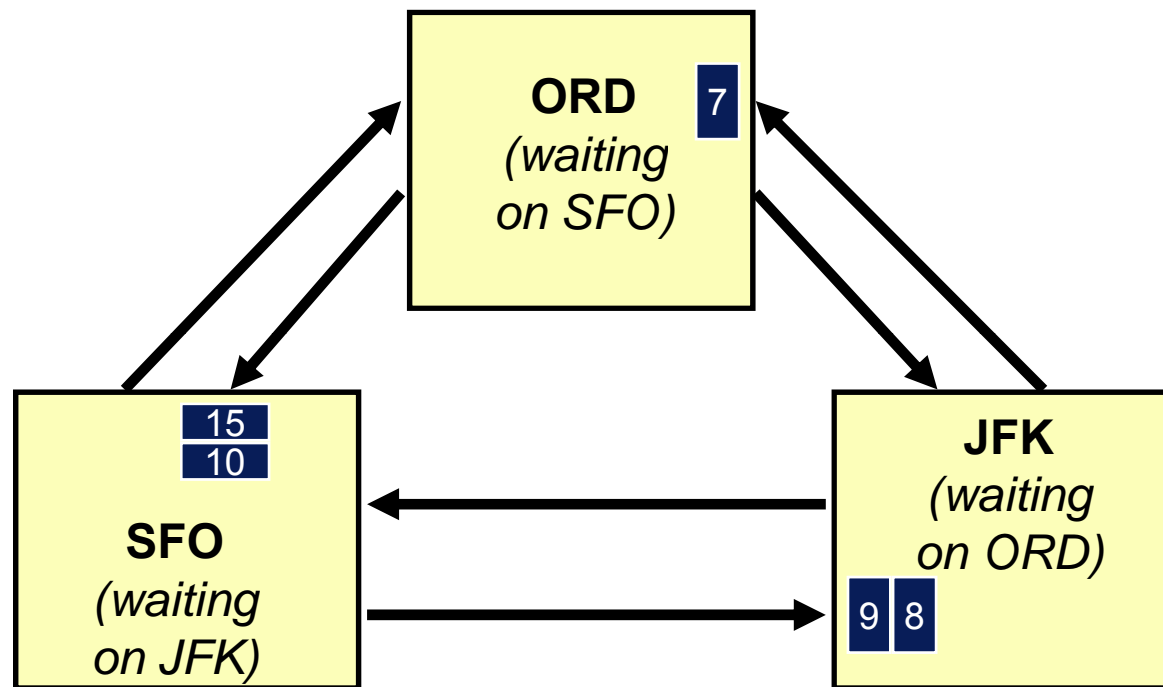
- ◆ Increasing lookahead
 - **No problem; lookahead can immediately be changed**
- ◆ Decreasing lookahead
 - **Previous time stamp guarantees must be honored**
 - **Lookahead cannot immediately be decreased**
 - ◆ If an LP is at simulation time 10 and lookahead is 5, it has promised subsequent messages will have a time stamp of at least 15
 - ◆ If lookahead were immediately set to 1, it could generate a message with time stamp 11
 - **Lookahead can decrease by k units of simulation time only after the LP has advanced k units of simulation time**

Example: Decreasing Lookahead

- ◆ SFO: simulation time = 10, lookahead = 5
- ◆ Future messages sent on link must have time stamp ≥ 15
- ◆ SFO: request its lookahead be reduced to 1



Preventing Time Creep: Next Event Time Information



Observation: smallest time stamped event is safe to process

Preventing Time Creep: Next Event Time Information

- ◆ Lookahead creep avoided by allowing the synchronization algorithm to immediately advance to time of the next (global) event
- ◆ Synchronization algorithm must know time stamp of LP's next event
- ◆ Each LP guarantees a logical time T such that if no additional events are delivered to LP with $TS < T$, all subsequent messages that LP produces have a time stamp at least $T+L$ (L = lookahead)
- ◆ This is a conditional guarantee (as compared with null message algorithm that uses unconditional information)

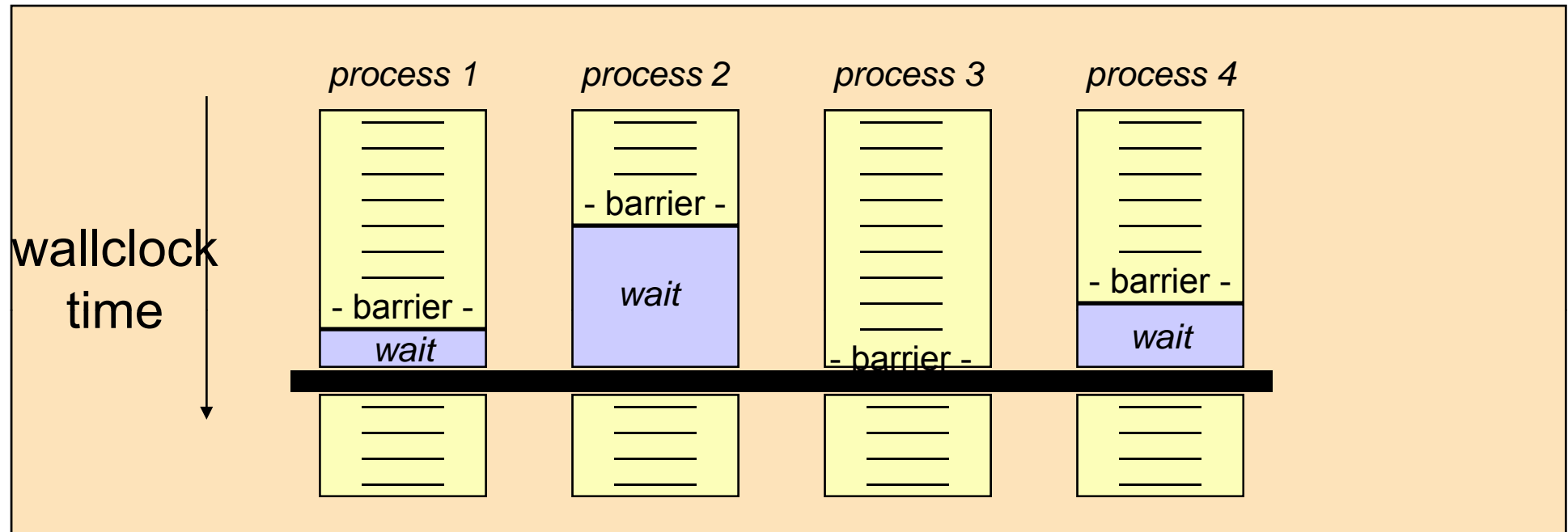
Outline – Conservative Synchronization

- ◆ A Simple Conservative Algorithm
 - **Deadlock**
- ◆ Null Message Algorithm
 - **Deadlock Avoidance Using Null messages**
 - **The Time Creep Problem**
 - **Lookahead**
- ◆ Synchronous Algorithms
 - **Implementation of Barrier Mechanisms**
 - **Computing LBTS**
 - **Transient Messages**
 - **Performance Improvements**

Synchronous Algorithms

- ◆ Synchronous algorithms use a global barrier to ensure events are processed in time stamp order
- ◆ Barrier Synchronization
 - Can be used to synchronize processes at regular intervals
 - A barrier call is inserted in each process at the point where it must synchronize
 - A process must wait at the barrier until all processes have reached the barrier
 - When all processes have reached the barrier, they may all continue

Barrier Synchronization



- ◆ Barrier Synchronization: when a process calls the barrier primitive, it will wait until all other processes have also invoked the barrier primitive

Synchronous Execution

- ◆ Goal is to ensure each LP processes events in time stamp order
- ◆ Basic idea: each Logical Process cycles through the following steps:
 - **Determine the events that are safe to process**
 - ◆ Compute a Lower Bound on the Time Stamp (LBTS_j) of events that LP_j might later receive
 - ◆ Events with time stamp \leq LBTS_j are safe to process
 - **Process safe events, send messages**
 - **Global synchronization (barrier)**
- ◆ Messages sent in one cycle are not eligible for processing until the next cycle

A Simple Synchronous Algorithm

- ◆ Assume instantaneous message transmission (discuss later)
- ◆ Assume any LP can communicate with any other LP
- ◆ N_i = time of next event in LP_i
- ◆ LA_i = lookahead of LP_i
- ◆ Algorithm

WHILE (simulation is not over)

receive messages generated in previous iteration

$LBTS = \min (N_i + LA_i)$ for all LP_i

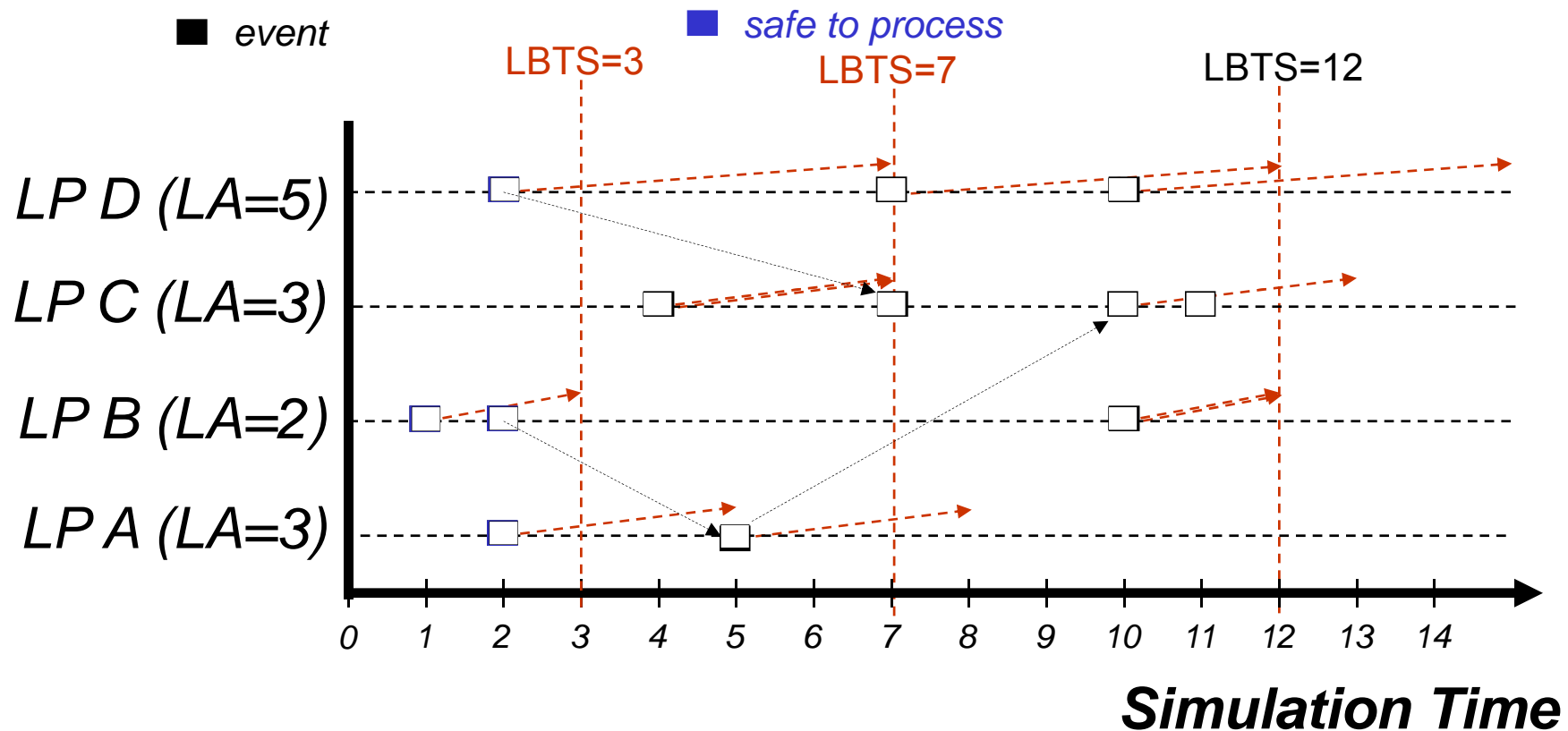
process events with time stamp $\leq LBTS$

barrier synchronization

END

If $LBTS_j$ is the same for all LPs, it is simply called LBTS

Synchronous Execution Example



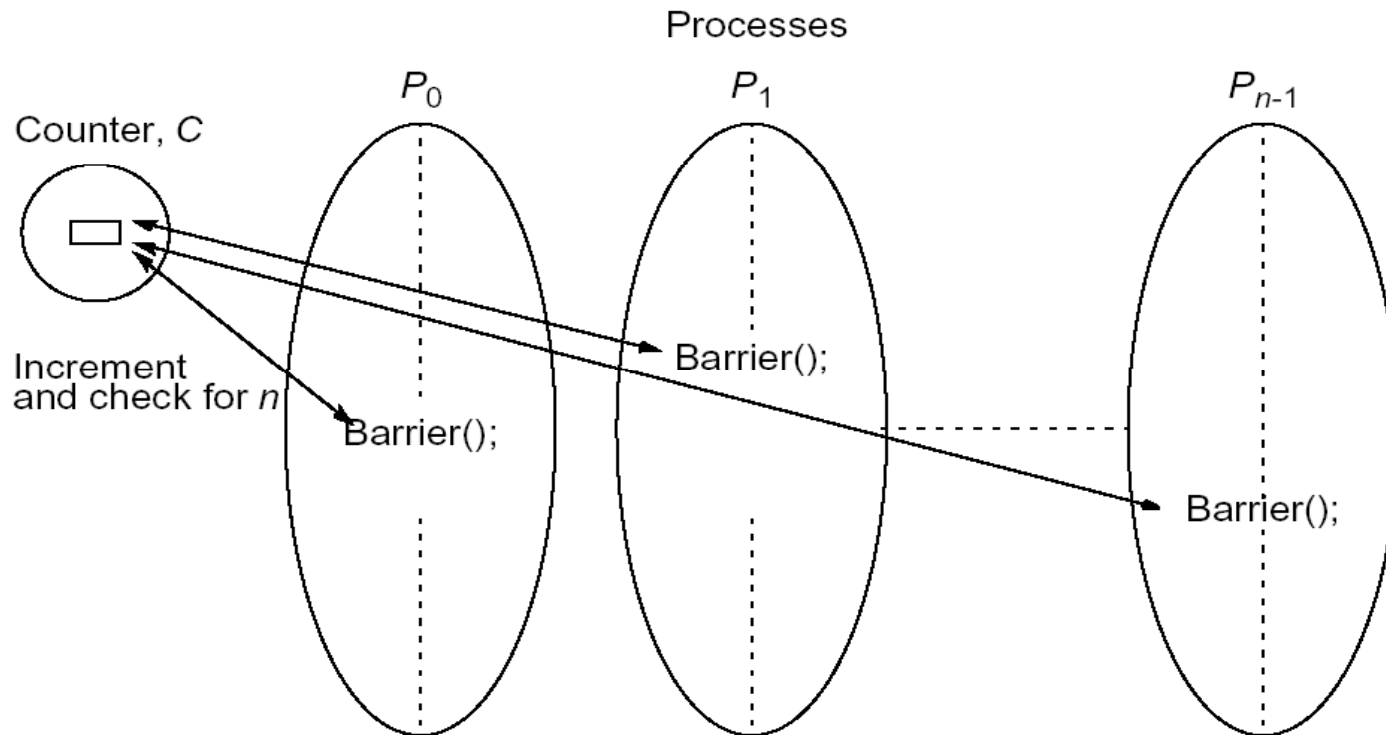
Issues

- ◆ Implementation of Barrier mechanisms
 - **Centralized Barrier**
 - **Broadcast Barrier**
 - **Tree Barrier**
 - **Butterfly Barrier**
- ◆ Computing LBTS (global minimum)
 - **The LBTS computation can be “piggybacked” onto the barrier synchronization algorithm**
- ◆ Transient messages
 - **Flush Barrier**

Barrier Using a Centralized Controller

- ◆ Centralized controller process used to implement barrier
- ◆ Overall, a two step process
 - **Controller determines when barrier reached**
 - **Broadcast message to release processes from the barrier**
- ◆ Barrier primitive for non-controller processes:
 - **Send a message to central controller**
 - **Wait for a reply**
- ◆ Barrier primitive for controller process
 - **Receive barrier messages from other processes**
 - **When a message is received from each process, broadcast message to release barrier**

Centralized Controller

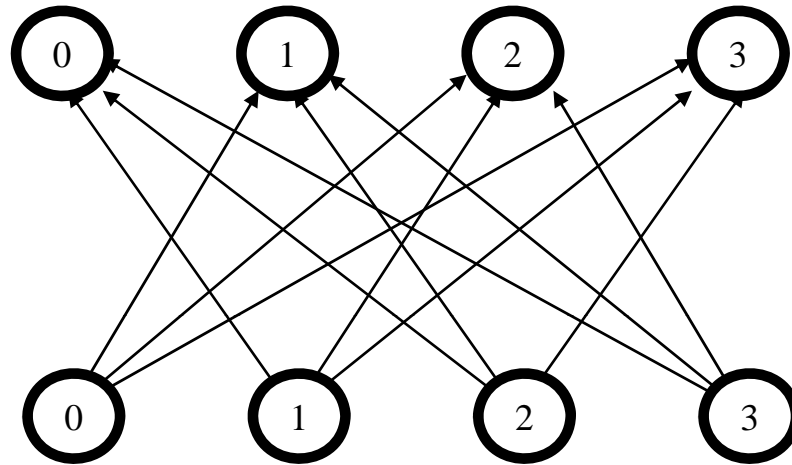


Performance

Controller must send and receive $N-1$ messages

Potential bottleneck $O(N)$ time

Broadcast Barrier



One step approach; no central controller

Each process:

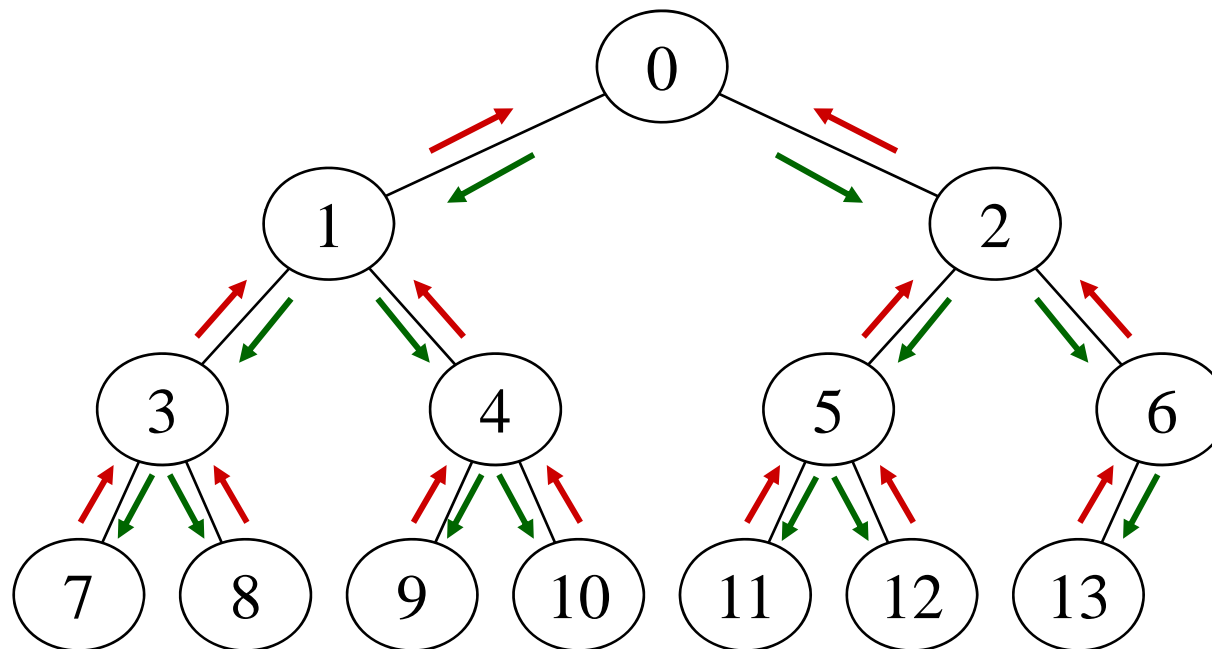
- ◆ Broadcast message when barrier primitive is invoked
- ◆ Wait until a message is received from each other process
- ◆ $N(N-1)$ messages but time per process is halved

Tree Barrier

- ◆ Organize processes into a tree
- ◆ A process sends a message to its parent process when
 - **The process has reached the barrier, and**
 - **A message has been received from each of its child processes**
- ◆ Root detects completion of barrier, and sends release message to child processes
- ◆ Child processes sends release messages down tree
- ◆ $2 \log N/2$ time if all processes reach barrier at same time

Tree Barrier

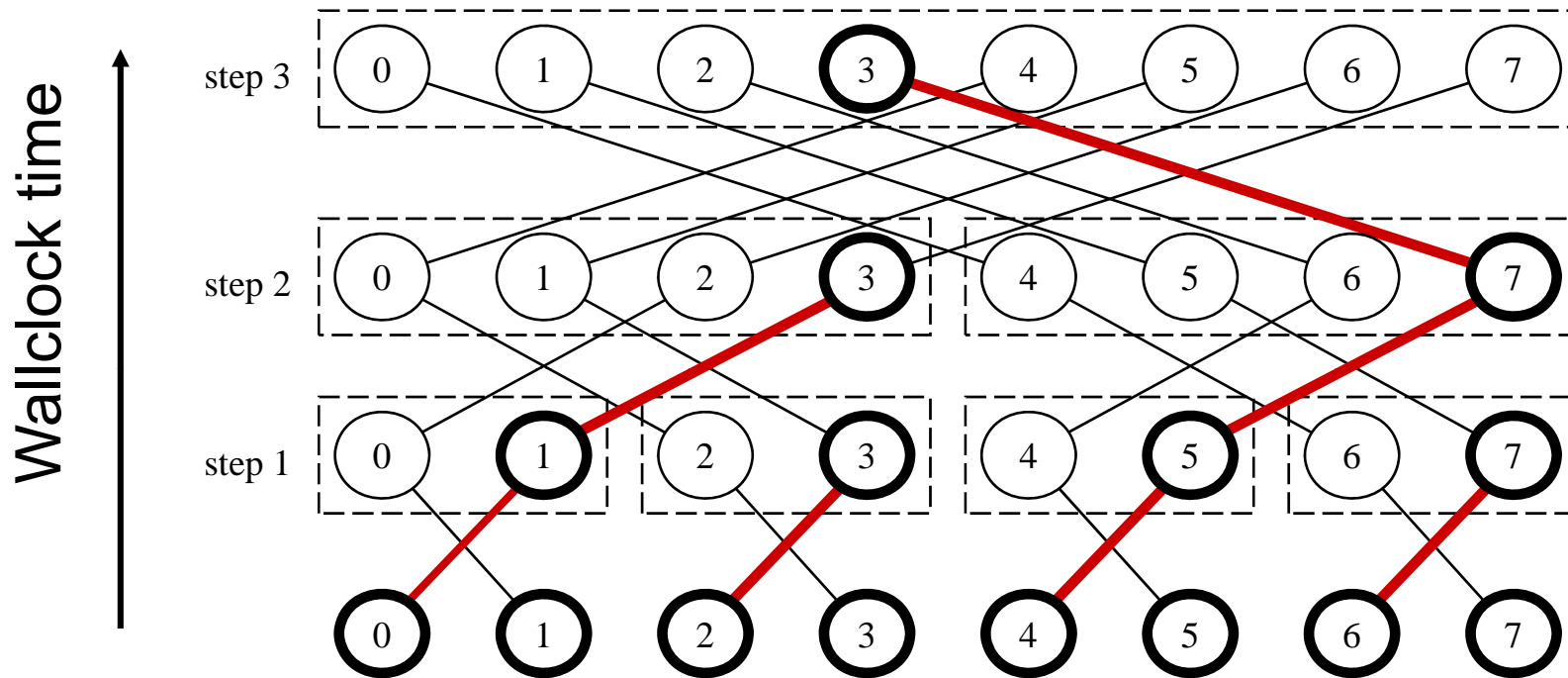
- ◆ Organize processes into a tree



Butterfly Barrier

- ◆ N processes (assume N is a power of 2)
- ◆ Sequence of $\log_2 N$ pairwise barriers (let $k = \log_2 N$)
- ◆ Pairwise barrier:
 - **Send message to partner process**
 - **Wait until message is received from that process**
- ◆ Process p: $b_k b_{k-1} \dots b_1$ = binary representation of p
- ◆ Step i: perform barrier with process $b_k \dots b_i' \dots b_1$
(complement ith bit of the binary representation)

Butterfly Barrier Example



Example: Process 3 (011)

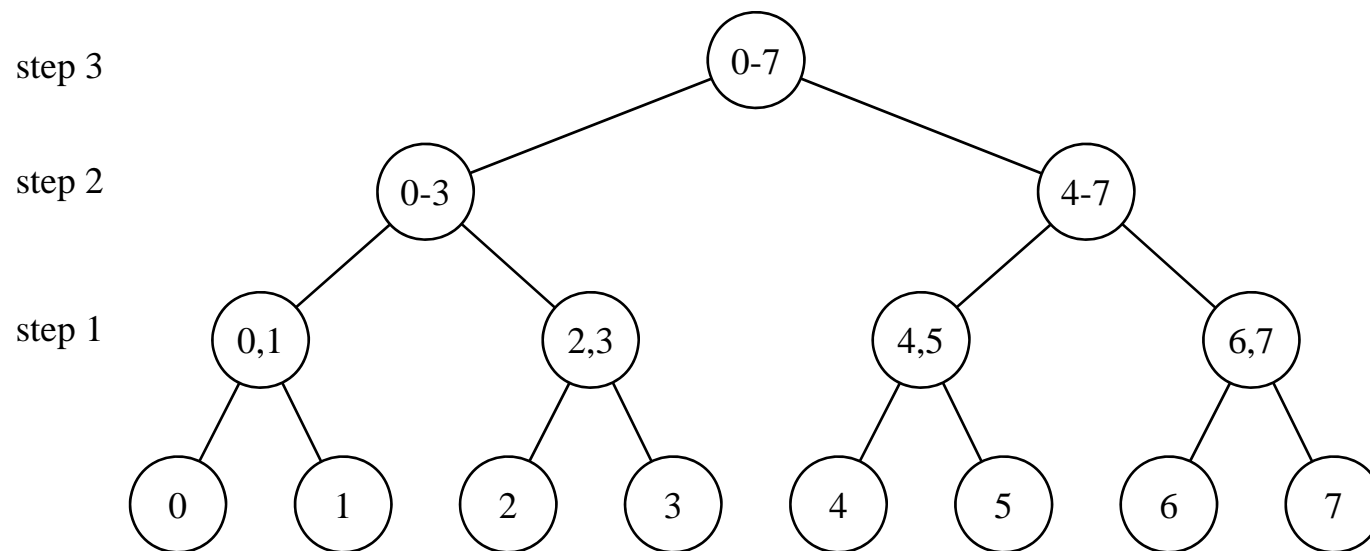
Step 1: pairwise barrier with process 2 (010)

Step 2: pairwise barrier with process 1 (001)

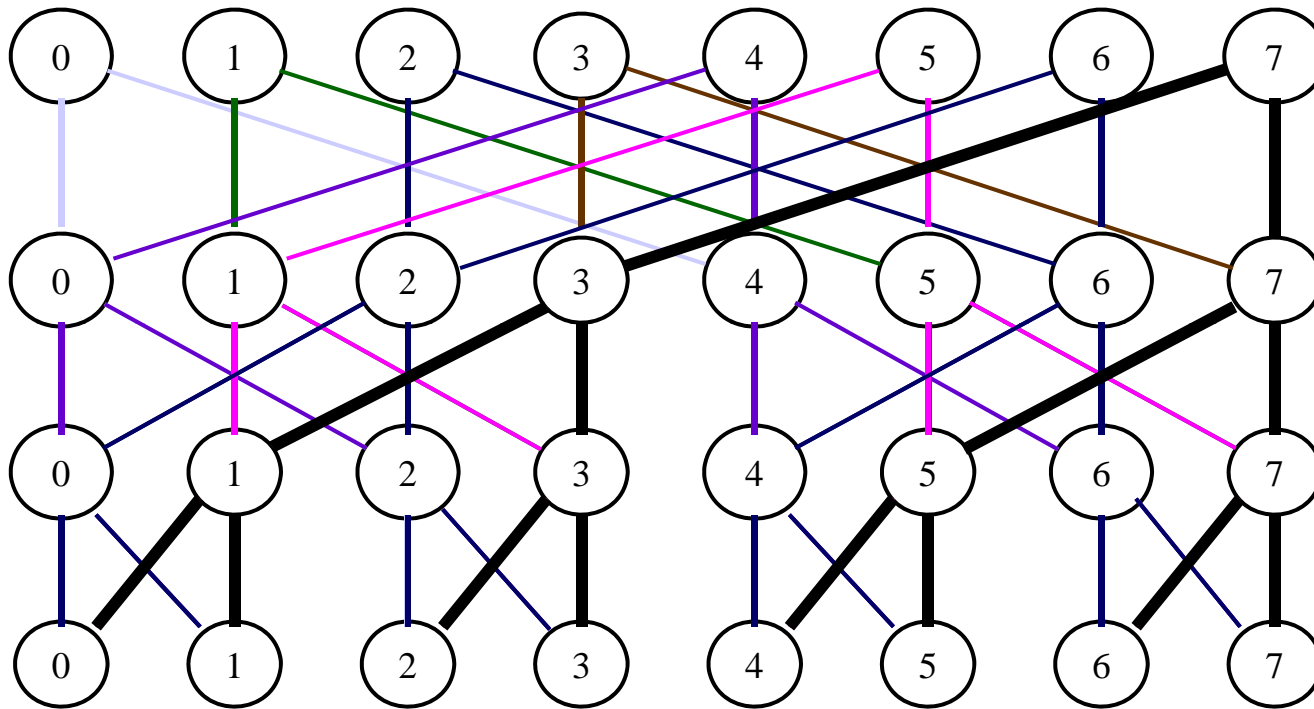
Step 3: pairwise barrier with process 7 (111)

Butterfly Barrier: Superimposed Trees

- ◆ An N node butterfly can be viewed as N trees superimposed over each other
 - **The communication pattern forms a tree from the perspective of any process**



Butterfly Barrier: Superimposed Trees

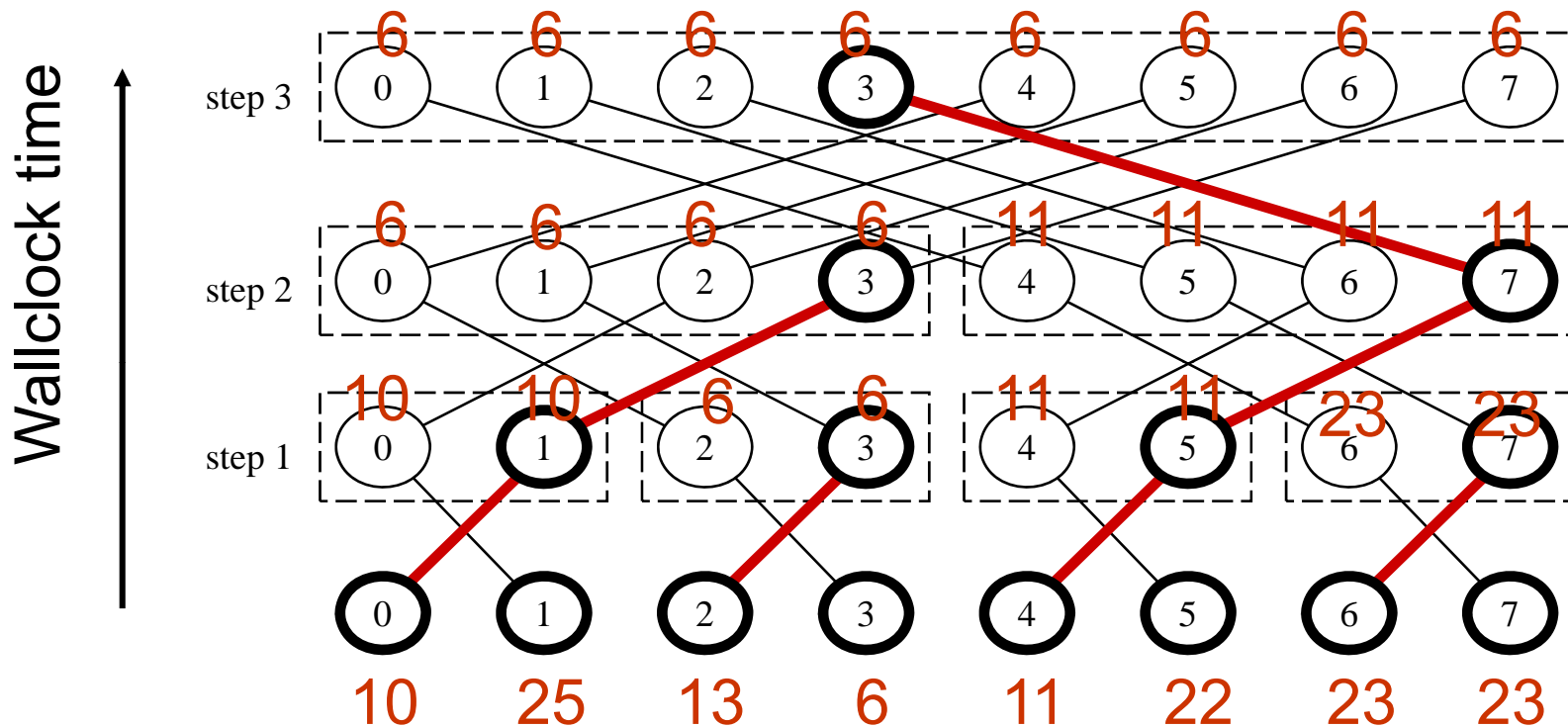


After $\log_2 N$ steps the barrier operation is completed

Computing LBTS

- ◆ It is easy to extend any of these barrier algorithms so that they also compute a global minimum (LBTS)
- ◆ Piggyback local time value on each barrier message
- ◆ Compute new minimum among local value, incoming message(s)
- ◆ Transmit new minimum in next step

Computing LBTS



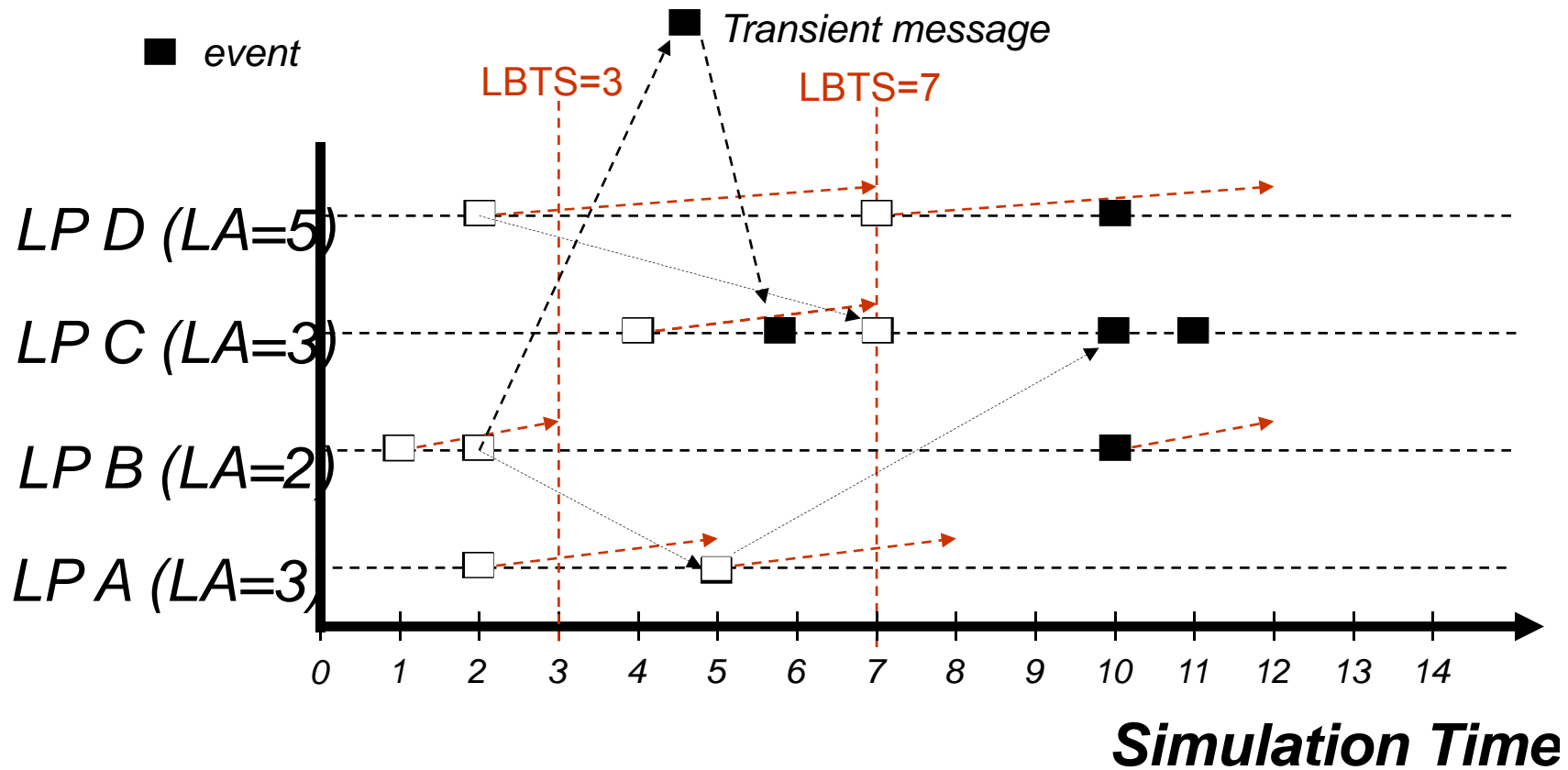
After log N steps

- (1) LBTS has been computed,
- (2) each process has the LBTS value

Transient Messages

- ◆ A transient message is a message that has been sent, but has not yet been received at its destination
- ◆ The message could be “in the network” or stored in an operating system buffer (waiting to be sent or delivered)
- ◆ The simple synchronous algorithm fails if transient message(s) remain after the processes are released from the barrier

Transient Message Example



Message arrives in LP C's past!

Solution: Flush Barrier

- ◆ No process will be released from the barrier until
 - All processes have reached the barrier
 - Any message sent by a process before reaching the barrier has arrived at its destination

- ◆ Revised Algorithm

WHILE (simulation is not over)

 receive messages generated in previous iteration

$LBTS = \min (N_i + LA_i)$ for all LP_i

 process events with time stamp $\leq LBTS$

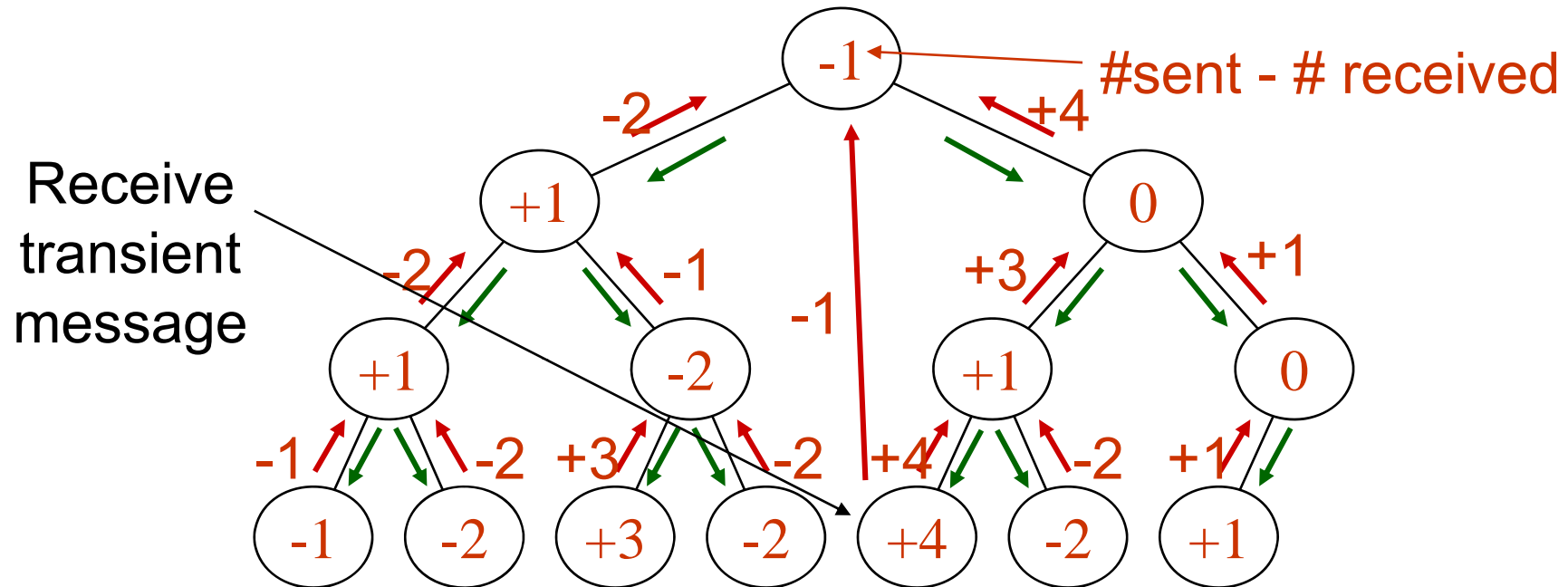
 flush barrier synchronization

END

Implementation

- ◆ Use FIFO communication channels
- ◆ Send a “dummy message” on each channel; wait until such a message is received on each incoming channel to guarantee transient messages have been received
 - May require a large number of messages
- ◆ Another approach: **message counters**
 - Send_i = number of messages sent by LP_i (this iteration)
 - Rec_i = number of messages received by LP_i (this iteration)
 - There are no transient messages when
 - ◆ All processes are blocked (i.e., at the barrier) and
 - ◆ $\sum \text{Send}_i = \sum \text{Rec}_i$

Tree: Flush Barrier



- ◆ When a leaf process reaches flush barrier, include counter ($\#sent - \#received$) in messages sent to parent
- ◆ Parent adds counters in incoming messages with its own counter, sends sum in message sent to its parent
- ◆ If sum at root is zero, broadcast “go” message, else wait until sum is equal to zero
- ◆ If receive message after reporting sum: send update message to root

Butterfly: Flush Barrier

◆ Butterfly

For ($i = 1$ to $\log N$)

send local counter to partner at step i

wait for message from partner at step i

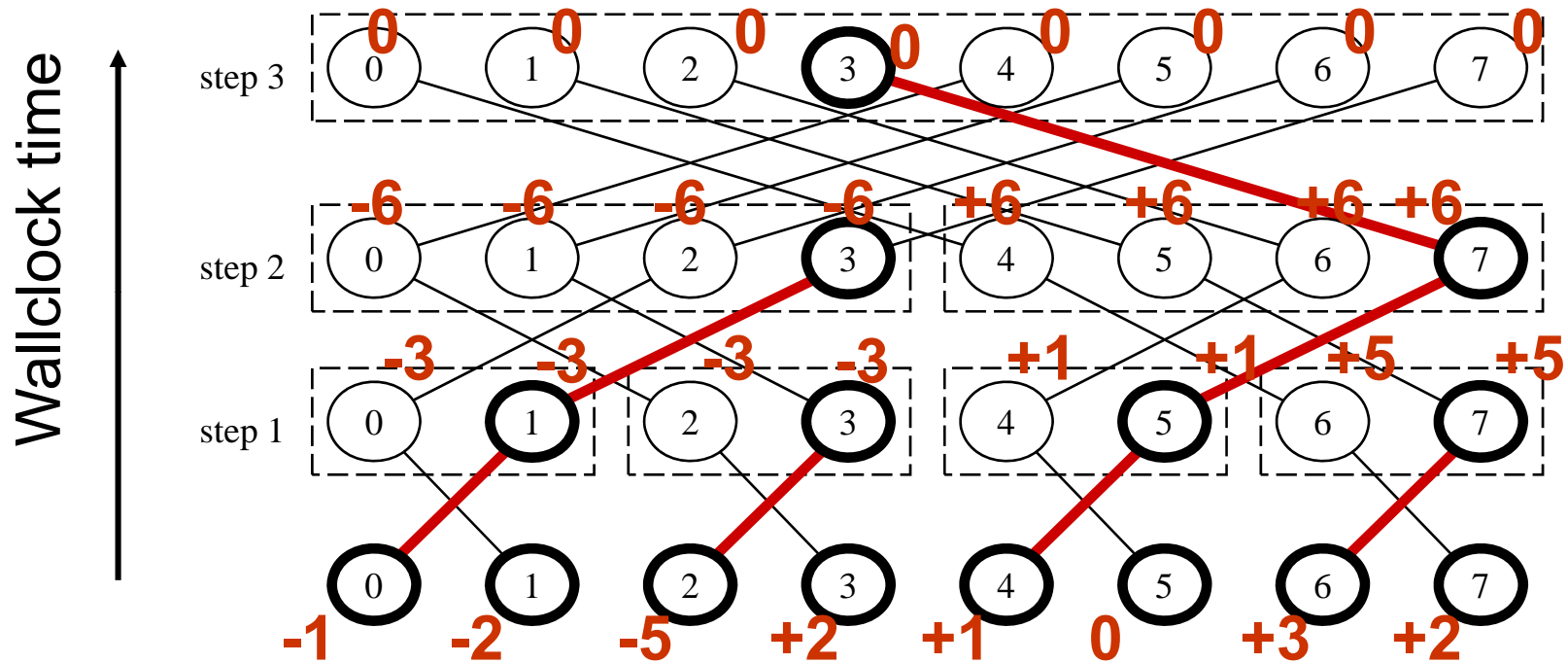
local counter = local counter + counter in message

End-for

◆ If local counter not zero after last step:

- **Send update messages up butterfly**
- **Alternatively, abort and retry**

Butterfly: Flush Barrier



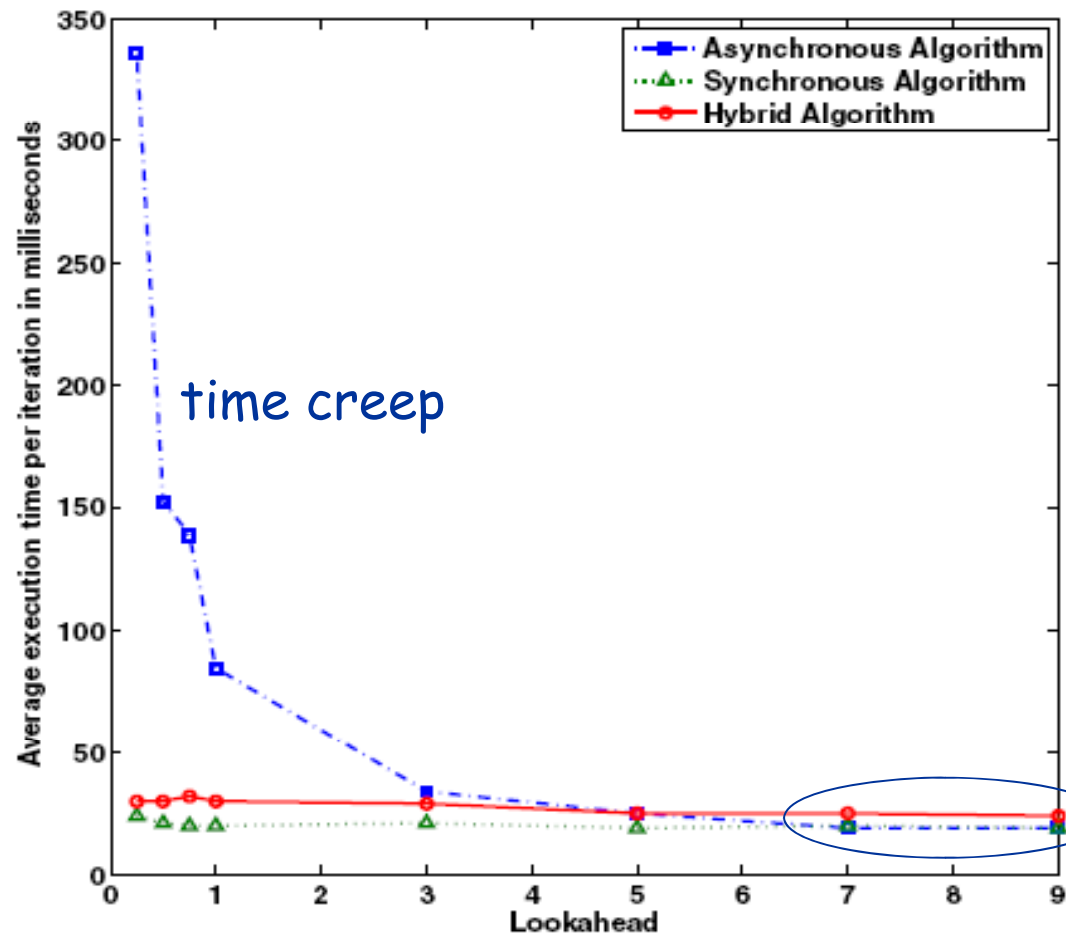
Synchronous Algorithms

- ◆ Synchronous algorithms use a global barrier to ensure events are processed in time stamp order
 - **Requires computation of a Lower Bound on Time Stamp (LBTS) on messages each LP might later receive**
- ◆ There are several ways to implement barriers
 - **Central controller**
 - **Broadcast**
 - **Tree**
 - **Butterfly**
- ◆ The LBTS computation can be “piggybacked” onto the barrier synchronization algorithm

Synchronous Algorithms

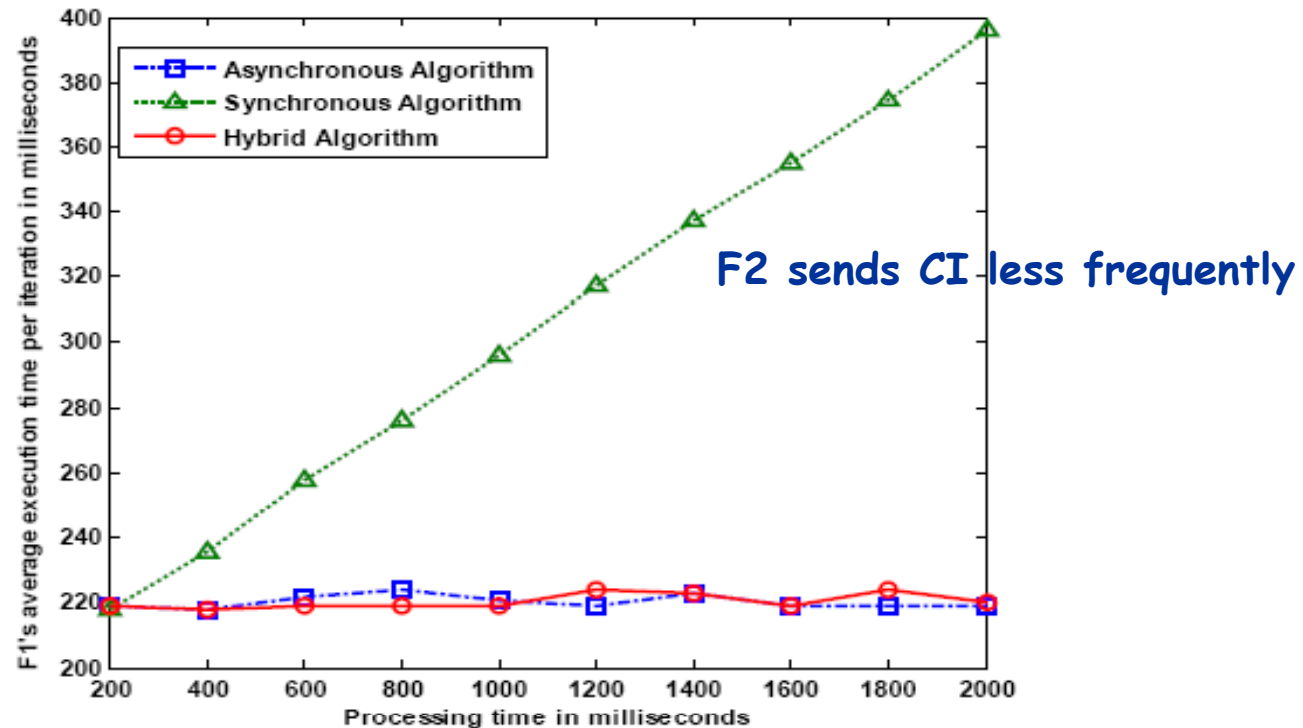
- ◆ Avoid Time Creep problem associated with Null Message Algorithm
- ◆ However, must wait for all LPs to arrive at the barrier
- ◆ An LP that involves a lot of processing may delay LBTS computation for other LPs
- ◆ Possible solution is to combine synchronous and asynchronous (null message) algorithms into a hybrid algorithm
- ◆ Hybrid algorithm uses both unconditional and conditional information

Comparison of Algorithms



hybrid algorithm performing slightly worse due to its requirement of exchanging both UI and CI

Comparison of Algorithms



Two LPs F1 and F2 where F2 has a large lookahead and large processing time

Summary

- ◆ **Parallel Discrete Event Simulation**
 - **Collection of LPs possibly running on different processors**
 - **LP communicate exclusively by exchanging messages**
- ◆ **Null Message Algorithm (Chandy/Misra/Bryant)**
 - **Null messages: Lower bound on the time stamp of future messages the LP will send – avoid deadlock**
- ◆ **Lookahead creep problem**
 - **No zero lookahead cycles allowed**
- ◆ **Lookahead**
 - **Constraint on time stamps of subsequent messages**
 - **Has large effect on performance**
 - **Programs must be coded to exploit lookahead**
 - **Use time of next event to avoid lookahead creep**

Summary

- ◆ Null message algorithm
 - Time creep problem
 - No zero lookahead cycles allowed
- ◆ Deadlock Detection and Recovery
 - Simple signaling protocol detects deadlock
 - Smallest time stamp event safe to process
- ◆ Synchronous algorithms
 - Barrier to ensure events are processed in time stamp order
 - LBTS computation can be “piggybacked” onto the barrier synchronization algorithm
 - Transient messages must be accounted for by the synchronization algorithm – Flush barrier

Summary

Pro:

- ◆ Good performance reported for many applications containing good lookahead (queueing networks, communication networks, wargaming)
- ◆ Relatively easy to implement
- ◆ Well suited for “federating” autonomous simulations, provided there is good lookahead

Con:

- ◆ Cannot fully exploit available parallelism in the simulation because they must protect against a “worst case scenario”
- ◆ Lookahead is essential to achieve good performance
- ◆ Writing simulation programs to have good lookahead can be very difficult or impossible, and can lead to code that is difficult to maintain

Outline – Optimistic Synchronization

- ◆ Optimistic Synchronization (Time Warp)
- ◆ Local Control Mechanism
 - **Rollback**
 - **Event cancellation**
- ◆ Global Control Mechanism
 - **Global Virtual Time**
 - **Fossil Collection**
- ◆ Other Mechanisms
 - **State Saving**
 - **Lazy Cancellation**
 - **Lazy Re-evaluation**

Synchronization Problem

- ◆ Local causality constraint: Events within each logical process must be processed in time stamp order
- ◆ Adherence to the local causality constraint is sufficient to ensure that the parallel simulation will produce exactly the same results as the corresponding sequential simulation
 - **provided events with the same time stamp are processed in the same order as in the sequential execution**
- ◆ Synchronization Problem
 - **An algorithm is needed to ensure each LP processes events according to the local causality constraint**

Synchronization Protocols

- ◆ Conservative synchronization: avoid violating the local causality constraint (wait until it is safe)
 - **1st generation: null messages (Chandy/Misra/Bryant)**
 - **2nd generation: time stamp of next event (avoid time creep problem)**
- ◆ Optimistic synchronization: allow violations of local causality to occur, but detect them at runtime and recover using a rollback mechanism
 - **Time Warp (Jefferson)**
 - **Approaches limiting amount of optimistic execution**

Time Warp Algorithm (Jefferson)

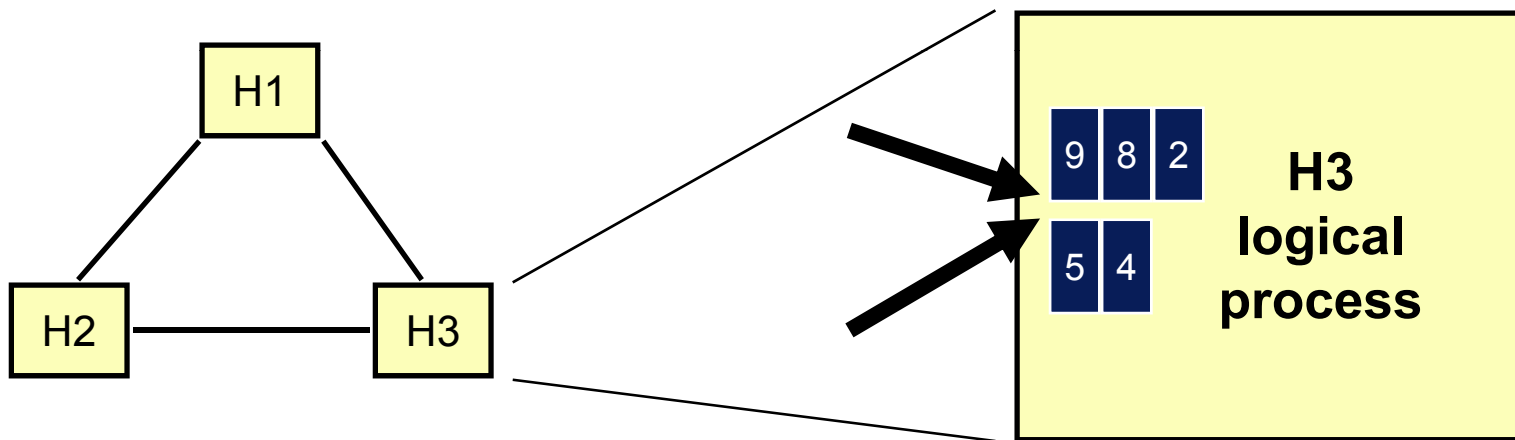
◆ Assumptions

- **Logical Processes (LPs) communicate only by exchanging time stamped event messages**
- **Dynamic network topology, dynamic creation of LPs possible**
- **Messages sent on each link need not be sent in time stamp order**
- **Network provides reliable delivery, but need not preserve order**

Time Warp Algorithm (Jefferson)

Basic idea:

- ◆ process events without worrying about messages that will arrive later
- ◆ detect out of order execution, recover using rollback



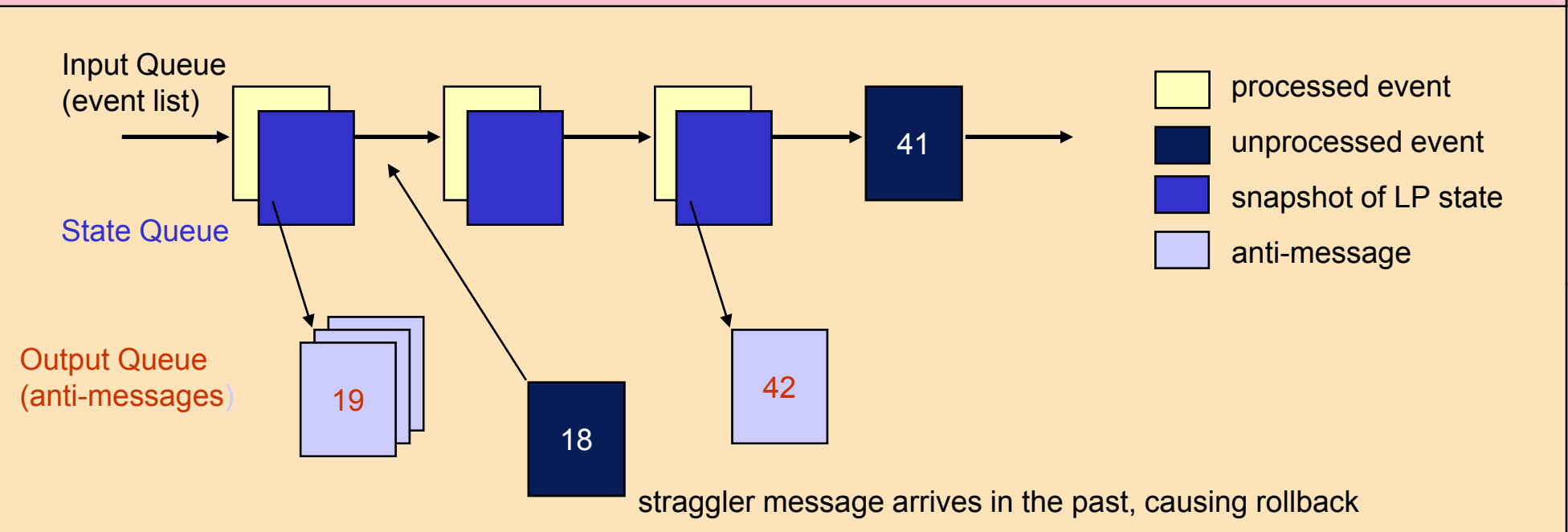
process all available events (2, 4, 5, 8, 9) in time stamp order

Outline – Optimistic Synchronization

- ◆ Optimistic Synchronization (Time Warp)
- ◆ Local Control Mechanism
 - **Rollback**
 - **Event cancellation**
- ◆ Global Control Mechanism
 - **Global Virtual Time**
 - **Fossil Collection**
- ◆ Other Mechanisms
 - **State Saving**
 - **Lazy Cancellation**
 - **Lazy Re-evaluation**

Time Warp: Local Control Mechanism

Each LP: process events in time stamp order, like a sequential simulator, except:
(1) do NOT discard processed events and (2) add a rollback mechanism



Adding rollback:

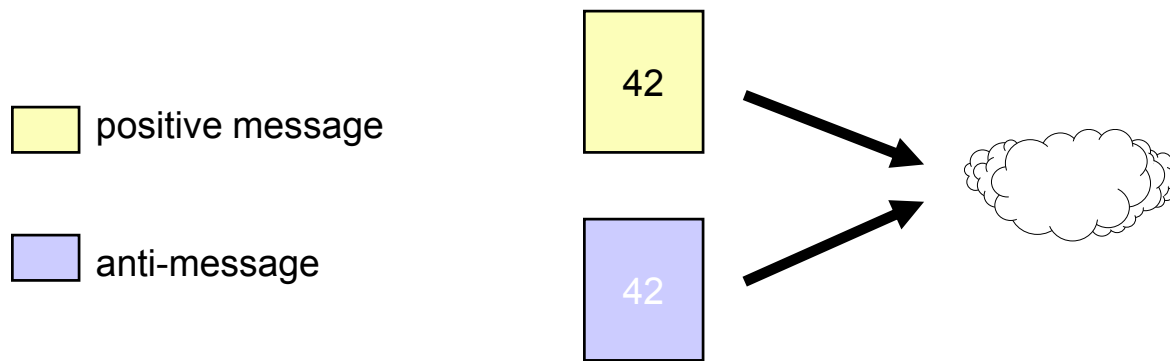
- a message arriving in the LP's past initiates rollback
- to roll back an event computation we must undo:
 - changes to state variables performed by the event;
solution: checkpoint state or use incremental state saving (state queue)
 - message sends
solution: anti-messages and message annihilation (output queue)

Anti-Messages

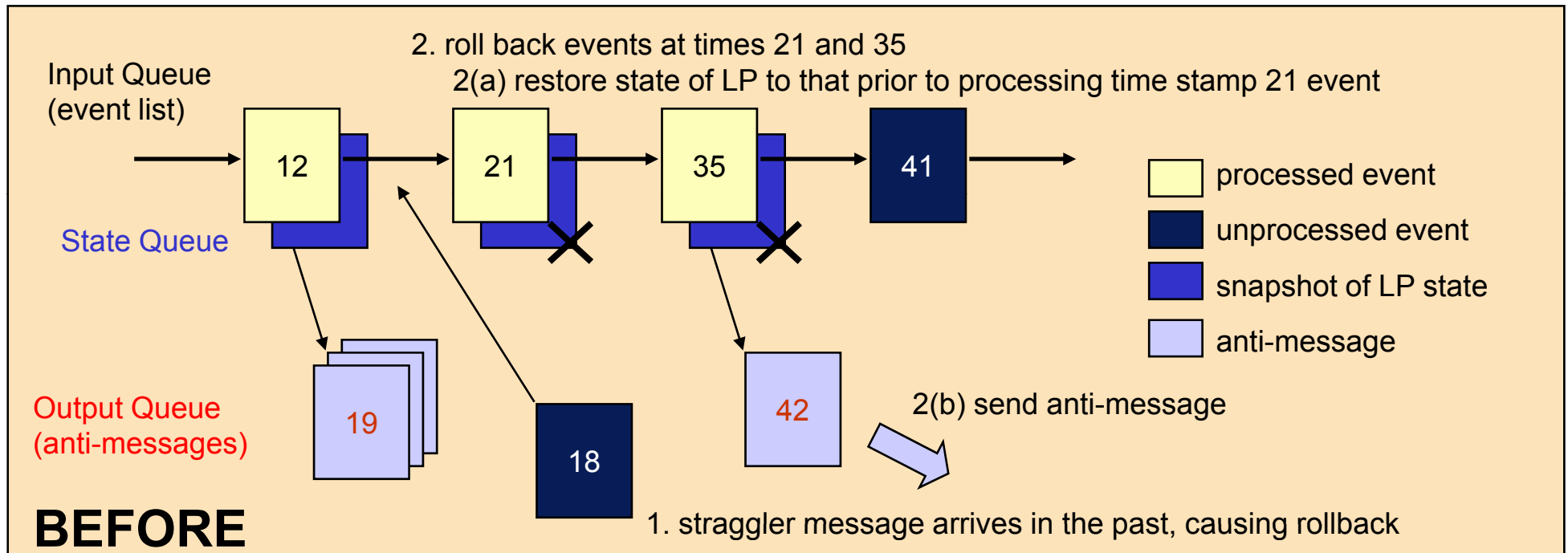
- ◆ Used to cancel a previously sent message
- ◆ Each positive message sent by an LP has a corresponding anti-message
- ◆ Anti-message is identical to positive message, except for a sign bit
- ◆ Message send: in addition to sending the message, leave a copy of the corresponding anti-message in a data structure in the sending LP called the output queue

Anti-Messages

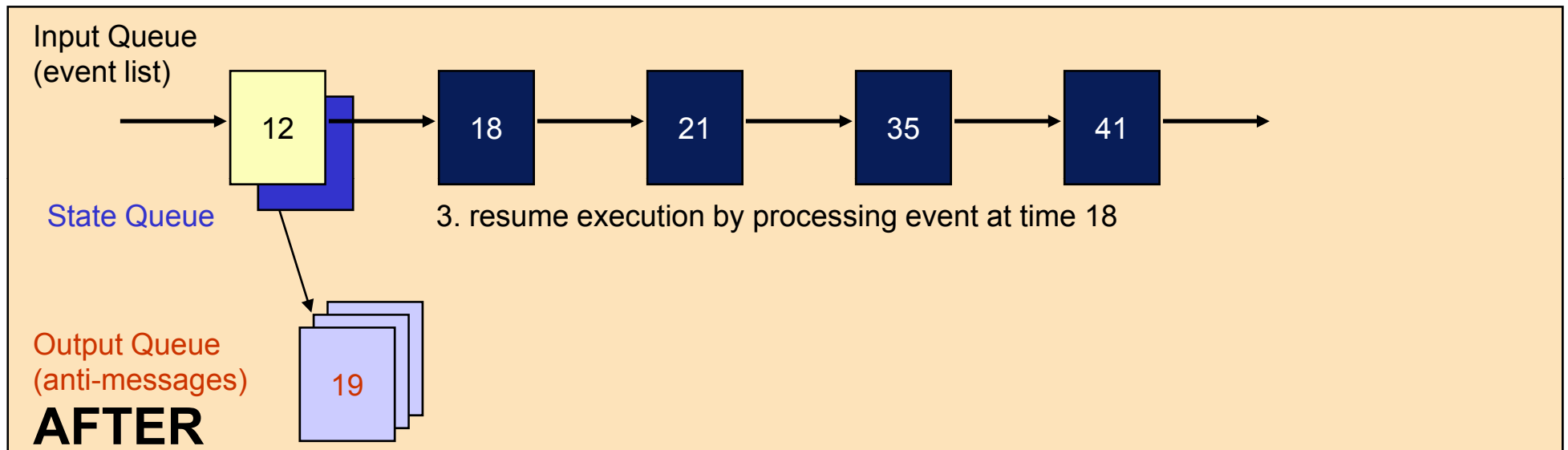
- ◆ When an anti-message and its matching positive message meet in the same queue, the two annihilate each other (analogous to matter and anti-matter)
- ◆ To undo the effects of a previously sent (positive) message, the LP need only send the corresponding anti-message



Rollback: Receiving a Straggler Message



Rollback: Receiving a Straggler Message



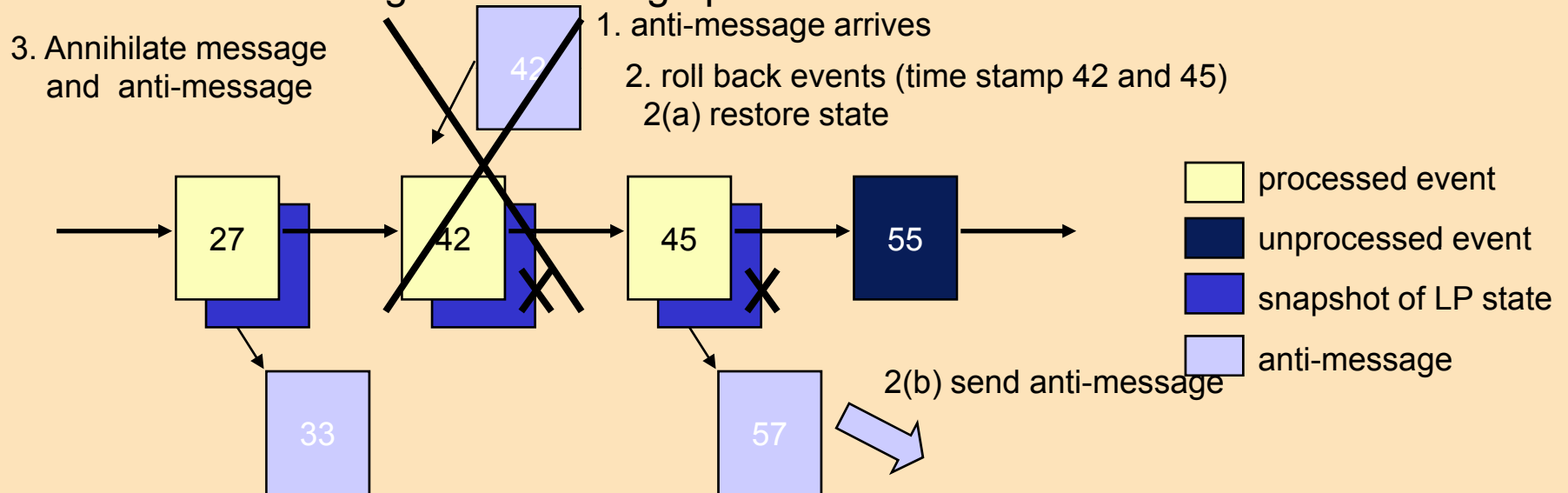
Processing Incoming Anti-Messages

Case I: corresponding message has not yet been processed

- annihilate message/anti-message pair

Case II: corresponding message has already been processed

- roll back to time prior to processing message (secondary rollback)
- annihilate message/anti-message pair



- may cause “cascaded” rollbacks; recursively applying eliminates all effects of error

Case III: corresponding message has not yet been received

- queue anti-message
- annihilate message/anti-message pair when message is received

Outline – Optimistic Synchronization

- ◆ Optimistic Synchronization (Time Warp)
- ◆ Local Control Mechanism
 - **Rollback**
 - **Event cancellation**
- ◆ Global Control Mechanism
 - **Global Virtual Time**
 - **Fossil Collection**
- ◆ Other Mechanisms
 - **State Saving**
 - **Lazy Cancellation**
 - **Lazy Re-evaluation**

Time Warp: Global Control Mechanism

- ◆ A mechanism is needed to:
 - **reclaim memory resources (e.g., old state and events) referred to as fossil collection**
 - **perform irrevocable operations (e.g., I/O)**
- ◆ A lower bound on the time stamp of any rollback that can occur in the future is needed
- ◆ The computation before this lower bound will not be rolled back, guaranteeing forward progress
- ◆ The lower bound must be computed using a global mechanism

Global Virtual Time and Fossil Collection

- ◆ Global Virtual Time (GVT) is defined as the minimum time stamp of any unprocessed (or partially processed) message or anti-message in the system
- ◆ GVT provides a lower bound on the time stamp of any future rollback
- ◆ Storage for events and state vectors older than GVT (except one state vector) can be reclaimed
- ◆ I/O operations with time stamp less than GVT can be performed

Global Virtual Time

- ◆ GVT(t): minimum time stamp among all unprocessed or partially processed messages at wallclock time t
 - **Computing GVT easy if an instantaneous snapshot of the computation could be obtained: compute minimum time stamp among**
 - ◆ Unprocessed events & anti-messages within each LP
 - ◆ Transient messages (messages sent before time t that are received after time t)
- ◆ Synchronous vs. Asynchronous GVT computation
 - **Synchronous** GVT algorithms: LPs stop processing events once a GVT computation has been detected
 - **Asynchronous** GVT algorithms: LPs can continue processing events and schedule new events while the GVT computation proceeds “in background”

GVT vs. LBTS

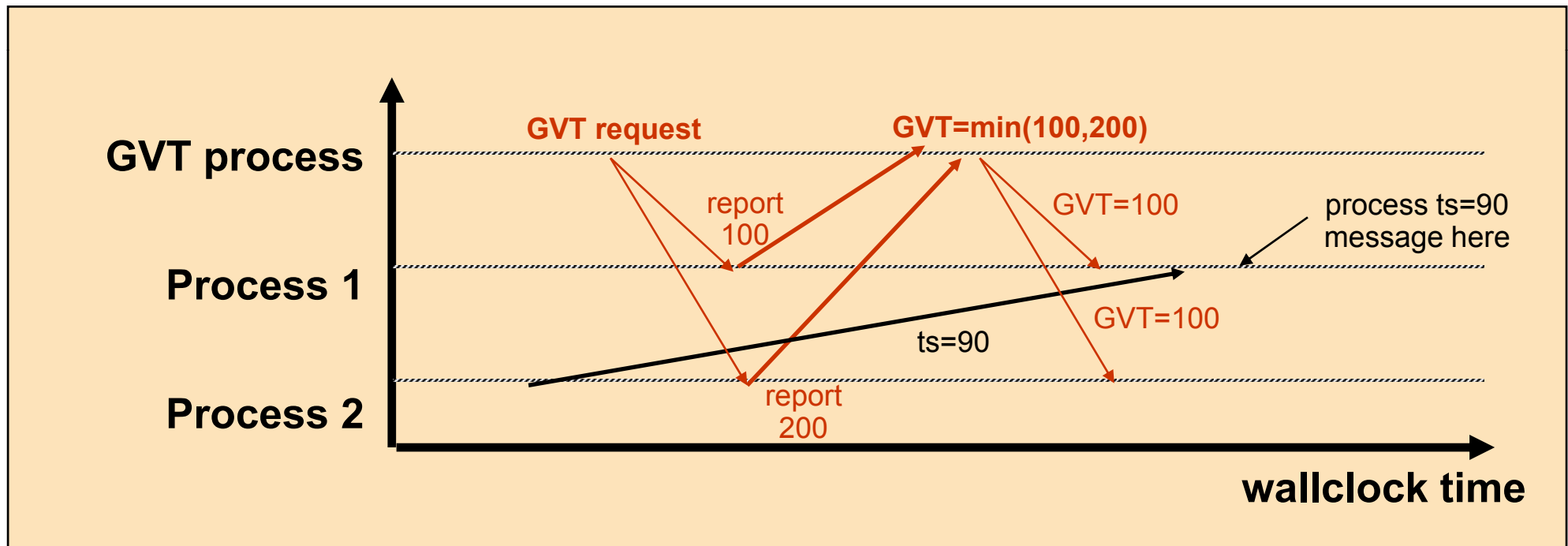
- ◆ Computing GVT is similar to computing the lower bound on time stamp (LBTS) of future events in conservative algorithms
 - **GVT algorithms can be used to compute LBTS and vice versa**
- ◆ Both determine the minimum time stamp of messages (or anti-message) that may later arrive
 - **Historically, developed separately**
 - **Often developed using different assumptions**
- ◆ Time Warp
 - **Latency to compute GVT typically less critical than the latency to compute LBTS**
 - **Asynchronous execution of GVT computation preferred to allow optimistic event processing to continue**

Asynchronous GVT

- ◆ An incorrect GVT algorithm
 - **Controller process: broadcast “compute GVT request”**
 - **Upon receiving the GVT request, each process computes its local minimum and reports it back to the controller**
 - **Controller computes global minimum, broadcast to others**
- ◆ Difficulties
 - ***transient message problem***: messages sent, but not yet received must be considered in computing GVT
 - ***simultaneous reporting problem***: different processors report their local minima at different points in wallclock times, leading to an incorrect GVT value

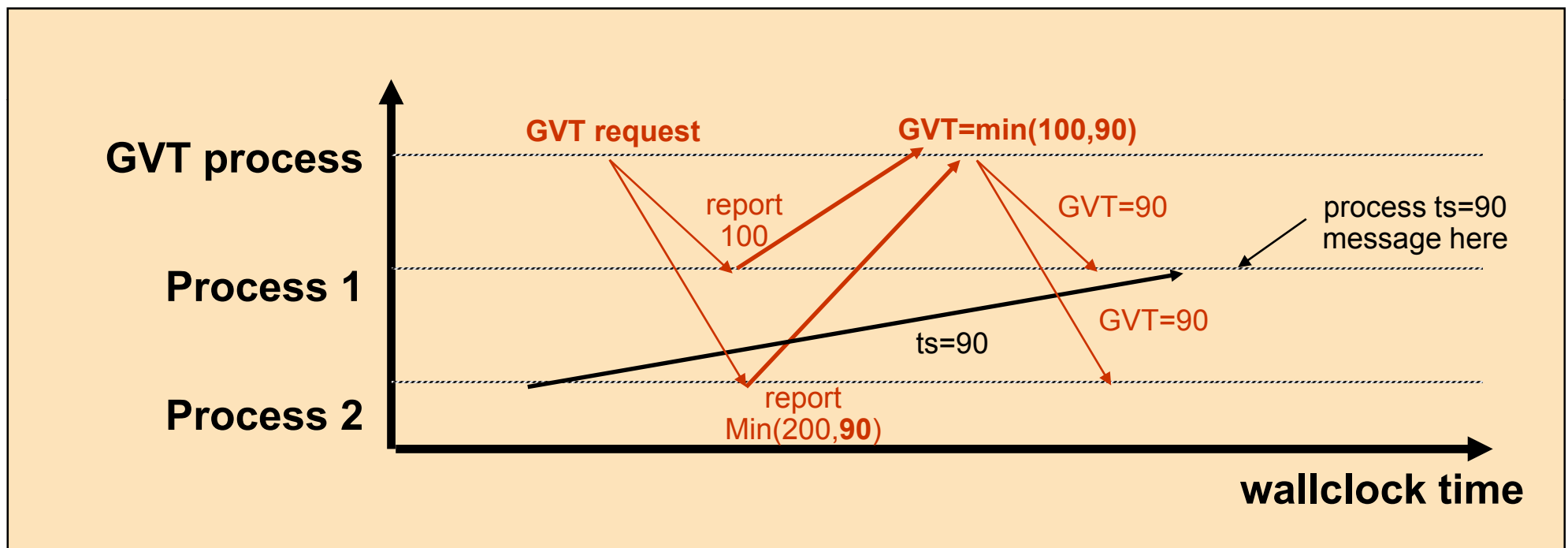
The Transient Message Problem

- ◆ Transient message: A message that has been sent, but has not yet been received at its destination
- ◆ Erroneous values of GVT may be computed if the algorithm does not take into account transient messages

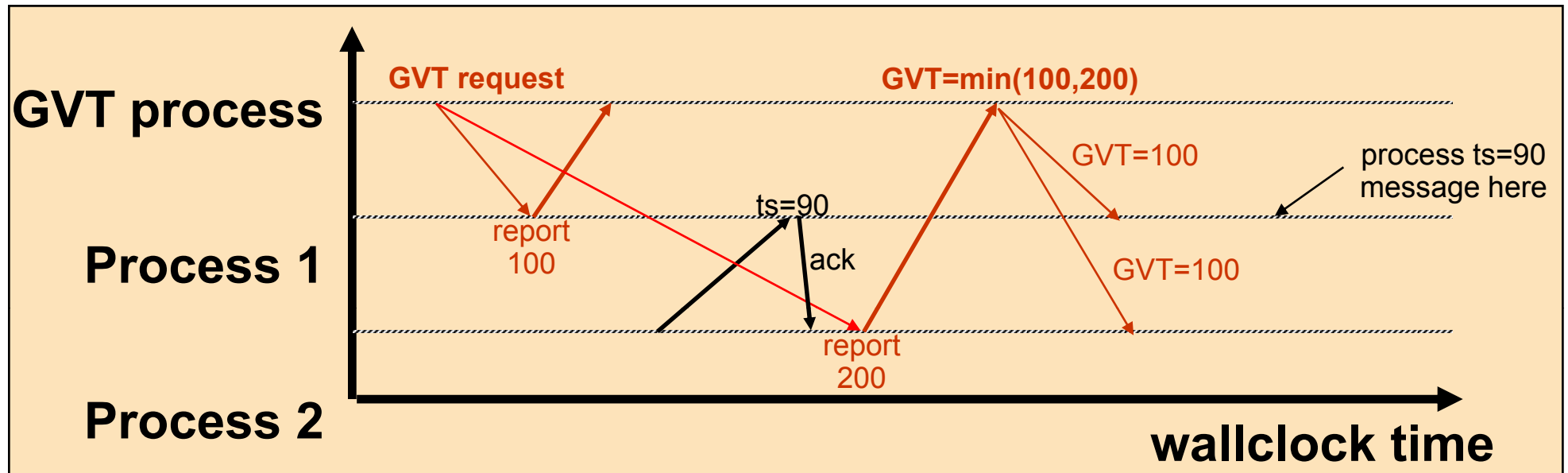


Transient Messages: A Solution

- ◆ Send an acknowledgement message for each message
- ◆ Report minimum of (local messages/anti-messages, time stamp of any unacknowledged messages)



The Simultaneous Reporting Problem



Do message acknowledgements solve this problem?

No, at least not by themselves

Solution: Mark acks that are sent after local min has been reported

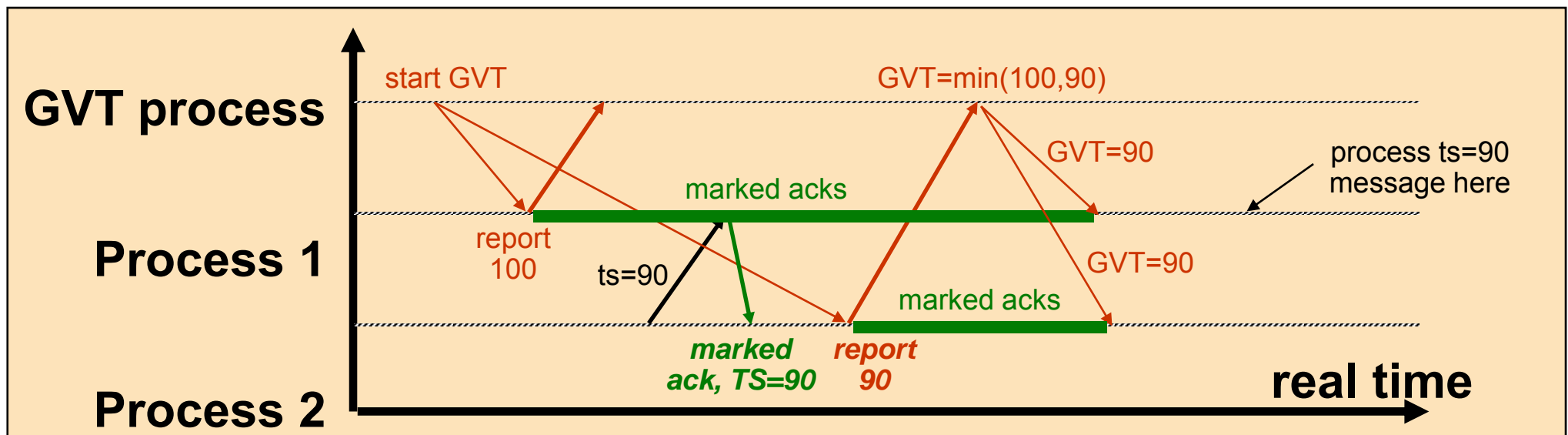
Samadi's Algorithm

Basic Idea

- ◆ Send an ack for each event messages & anti-messages received
- ◆ “Mark” acks sent after the processor has reported its local minimum

Algorithm

- ◆ Controller broadcasts “start GVT” message
- ◆ Each processor reports minimum time stamp among (1) local messages, (2) unacknowledged sent messages, (3) marked acks that were received
- ◆ Subsequent acks sent by process are marked until new GVT is received
- ◆ Controller computes global minimum as GVT value, broadcasts new GVT



Global Virtual Time

- ◆ Similar to lower bound on time stamp (LBTS)
 - **Time Warp: GVT usually not as time critical as LBTS**
 - **Asynchronous GVT computation preferred**
- ◆ Samadi's Algorithm
 - **Transient message problem: Message acknowledgements on event messages**
 - **Simultaneous reporting problem: Mark acknowledgements sent after reporting local minimum**
- ◆ Many other GVT algorithms
 - **Mattern's Algorithm**
 - ◆ **Asynchronous**
 - ◆ **Avoids message acknowledgements**
 - ◆ **Based on techniques for creating distributed snapshots**

Fossil Collection

◆ Batch Fossil Collection

- **After GVT computation, scan through list of LPs on that processor to reclaim memory and commit I/O operations**
- **May be time consuming if many LPs**

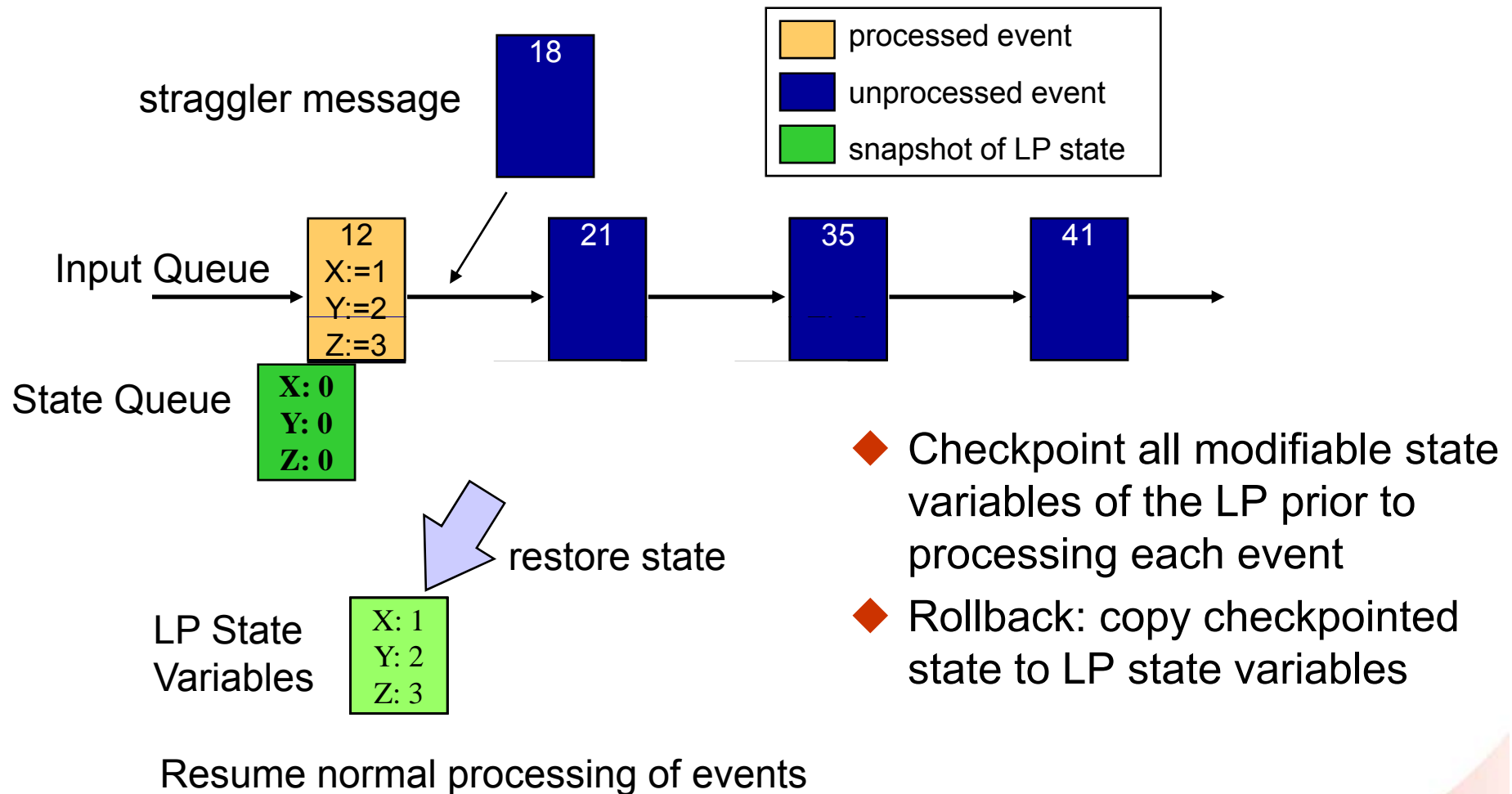
◆ On-the-fly Fossil Collection

- **After processing event, place memory into “free memory” list**
- **Before allocating memory, check that time stamp is less than GVT before reusing memory**

Outline – Optimistic Synchronization

- ◆ Optimistic Synchronization (Time Warp)
- ◆ Local Control Mechanism
 - **Rollback**
 - **Event cancellation**
- ◆ Global Control Mechanism
 - **Global Virtual Time**
 - **Fossil Collection**
- ◆ Other Mechanisms
 - **State Saving**
 - **Lazy Cancellation**
 - **Lazy Re-evaluation**

Copy State Saving



Copy State Saving

◆ Drawbacks

- **Forward execution slowed by checkpointing**

- ◆ **Must state save even if no rollbacks occur**

- ◆ **Inefficient if most of the state variables are not modified by each event**

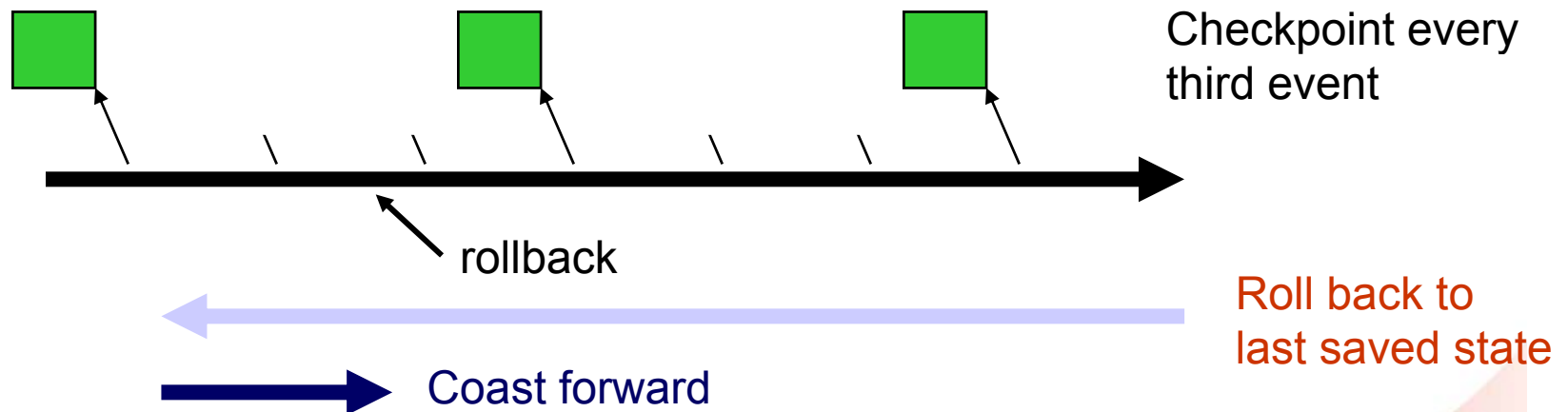
- ◆ **Consumes large amount of memory**

- ◆ **Copy state saving is only practical for LPs that do not have a large state vector**

- ◆ **Largely transparent to the simulation application (only need locations of LP state variables)**

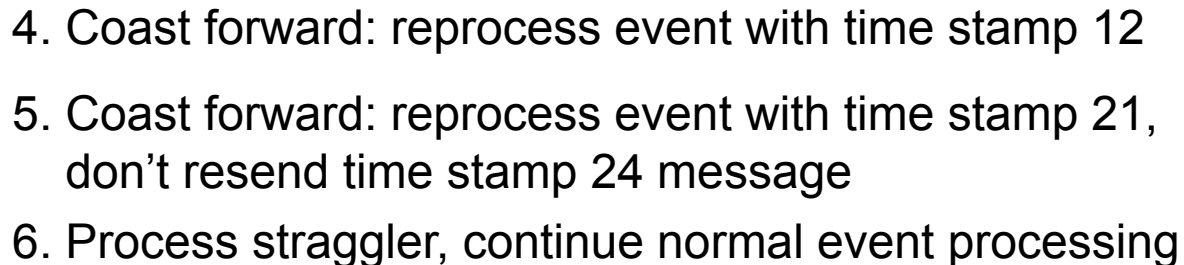
Infrequent State Saving

- ◆ Checkpoint LP periodically, e.g., every Nth event
- ◆ Rollback to time T: May not have saved state at time T
 - **Roll back to most recent checkpointed state prior to simulation time T**
 - **Execute forward (“coast forward”) to time T**



Infrequent State Saving

- ◆ Coast forward phase
 - Only needed to recreate state of LP at simulation time T
 - Coast forward execution identical to the original execution
 - Must “turn off” message sends during coast forward, or else
 - ◆ Rollback to T could cause new messages with time stamp $< T$, and roll backs to times earlier than T
 - ◆ Could lead to rollbacks earlier than GVT



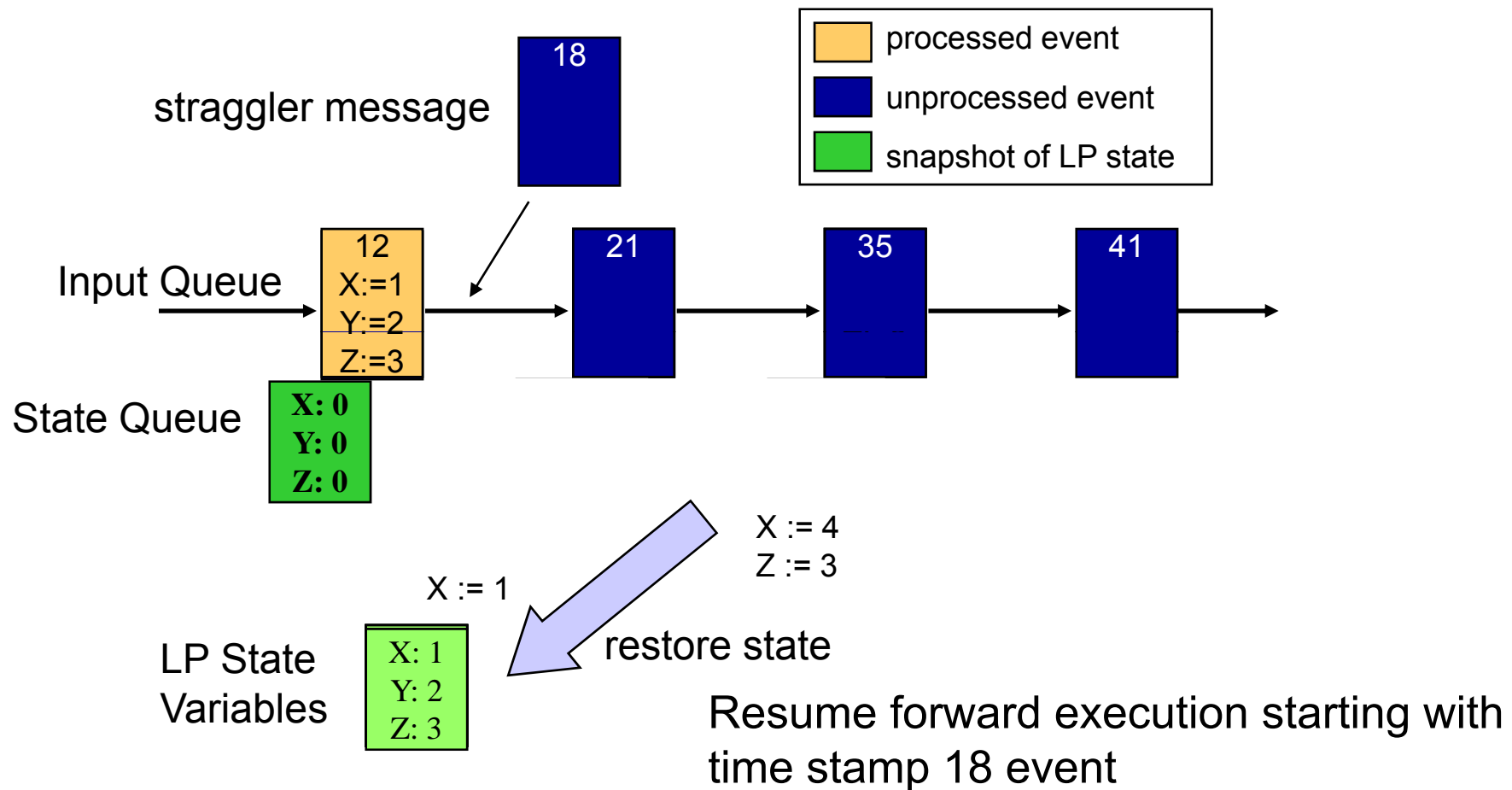
Infrequent State Saving

- ◆ Reduces time required for state saving
- ◆ Reduces memory requirements
- ◆ Increases time required to roll back LP
- ◆ Increases complexity of Time Warp executive
- ◆ Largely transparent to the simulation application (only need locations of LP state variables and frequency parameter)

Incremental State Saving

- ◆ Only state save variables modified by an event
 - **Generate “change log” with each event indicating previous value of state variable before it was modified**
- ◆ Rollback
 - **Scan change log in reverse order, restoring old values of state variables**

Incremental State Save Example



Before modifying a state variable, save current version in state queue

Rollback: Scan state queue from back, restoring old values

Incremental State Saving

- ◆ Must log addresses of modified variables in addition to state
- ◆ More efficient than copy state save if most state variables are not modified by each event
- ◆ Can be used *in addition* to copy state save
- ◆ Implementation
 - **Manual insertion of state save primitives**
 - **Compiler inserts checkpoint primitives**
 - **Executable editing: modify executable to insert checkpoint primitives**
 - **Overload assignment operator**

Lazy Cancellation

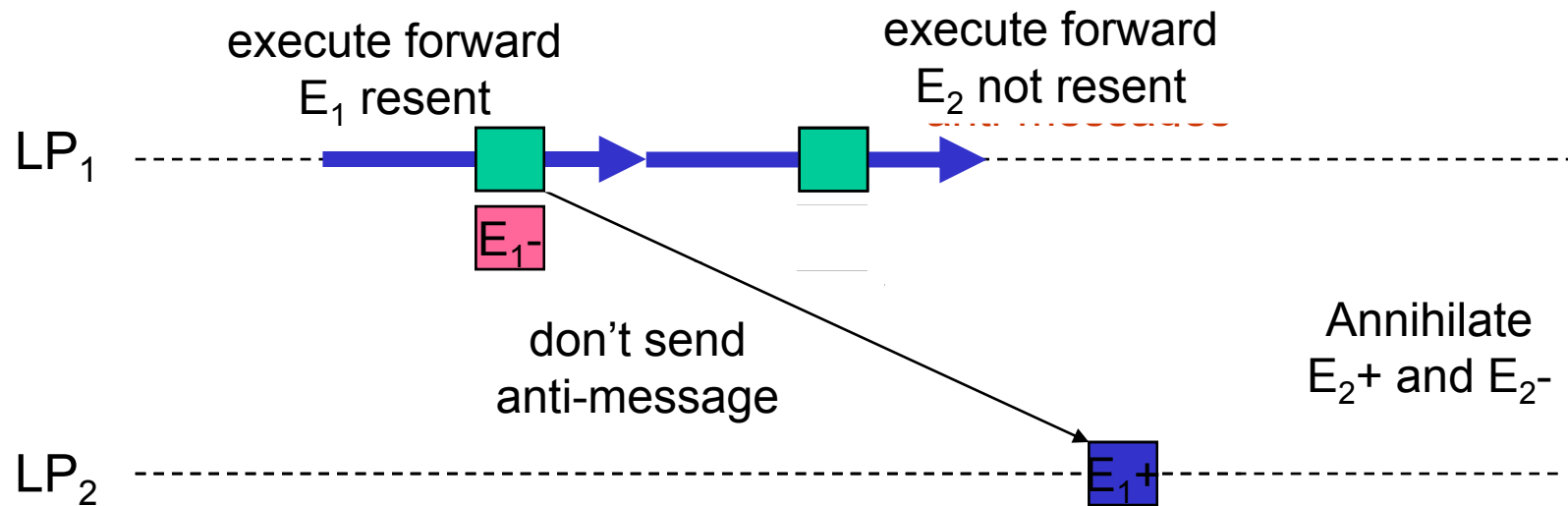
◆ Motivation

- **Re-execution after rollback may generate the same messages as the original execution**
- **In this case, need not cancel original message**

◆ Mechanism

- **Rollback: do not immediately send anti-messages**
- **After rollback, recompute forward**
- **Only send anti-message if recomputation does NOT produce the same message again**

Example: Lazy Cancellation



Lazy cancellation avoids unnecessary rollback

Lazy Cancellation

◆ Benefit

- **Avoid unnecessary message cancellations**

◆ Liabilities

- **Extra overhead (message comparisons)**
- **Delay in canceling wrong computations**
- **More memory required**

◆ In Practice

- **Lazy cancellation typically improves performance**
- **Empirical data indicate 10% improvement typical**

Lazy Re-evaluation

◆ Motivation

- **Re-execution of event after rollback may produce same result (LP state) as the original execution**
- **In this case, original rollback was unnecessary**

◆ Mechanism

- **Rollback: do not discard state vectors of rolled back computations**
- **Process straggler event, recompute forward**
- **During recomputation, if the state vector and input queue match that of the original execution, immediately “jump forward” to state prior to rollback**

Lazy Re-evaluation

◆ Benefit

- **Avoid unnecessary recomputation on rollback**
- **Works well if straggler does not affect LP state (query events)**

◆ Liabilities

- **Extra overhead (state comparisons)**
- **More memory required**

◆ In Practice

- **Typically does not improve overall performance**
- **Useful in certain special cases (e.g., query events)**

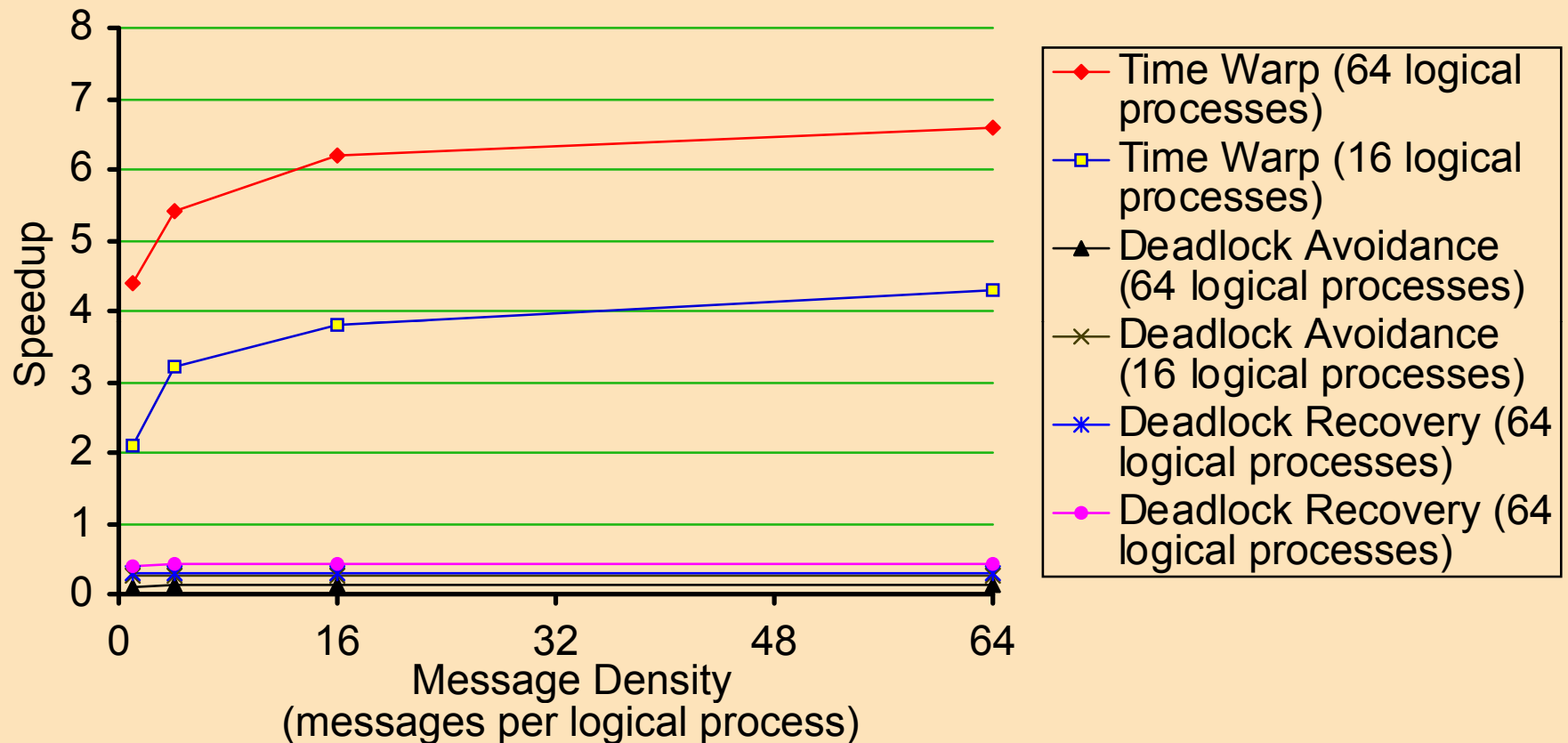
Other Optimistic Algorithms

Principal goal: avoid excessive optimistic execution

A variety of protocols have been proposed, including:

- window-based approaches
 - only execute events in a moving window (simulated time, memory)
- risk-free execution
 - only send messages when they are guaranteed to be correct
- add optimism to conservative protocols
 - specify “optimistic” values for lookahead
- hybrid approaches
 - mix conservative and optimistic LPs
- adaptive protocols
 - dynamically adjust protocol during execution as workload changes

Time Warp vs Conservative Performance



- eight processors
- closed queueing network, hypercube topology
- high priority jobs preempt service from low priority jobs (1% high priority)
- exponential service time (poor lookahead)

Summary

- ◆ Optimistic synchronization: detect and recover from synchronization errors rather than prevent them
- ◆ Time Warp
 - **Local control mechanism**
 - ◆ Rollback
 - ◆ Anti-messages
 - ◆ State saving
 - **Global control mechanism**
 - ◆ Global Virtual Time (GVT)
 - ◆ Fossil collection to reclaim memory
 - ◆ Commit irrevocable operations (e.g., I/O)

Summary

Pro:

- ◆ good performance reported for a variety of application (queuing networks, communication networks, logic circuits, combat models, transportation systems)
- ◆ offers the best hope for “general purpose” parallel simulation software (not as dependent on lookahead as conservative methods)
- ◆ “Federating” autonomous simulations
 - **avoids specification of lookahead**
 - **caveat: requires providing rollback capability in the simulation**

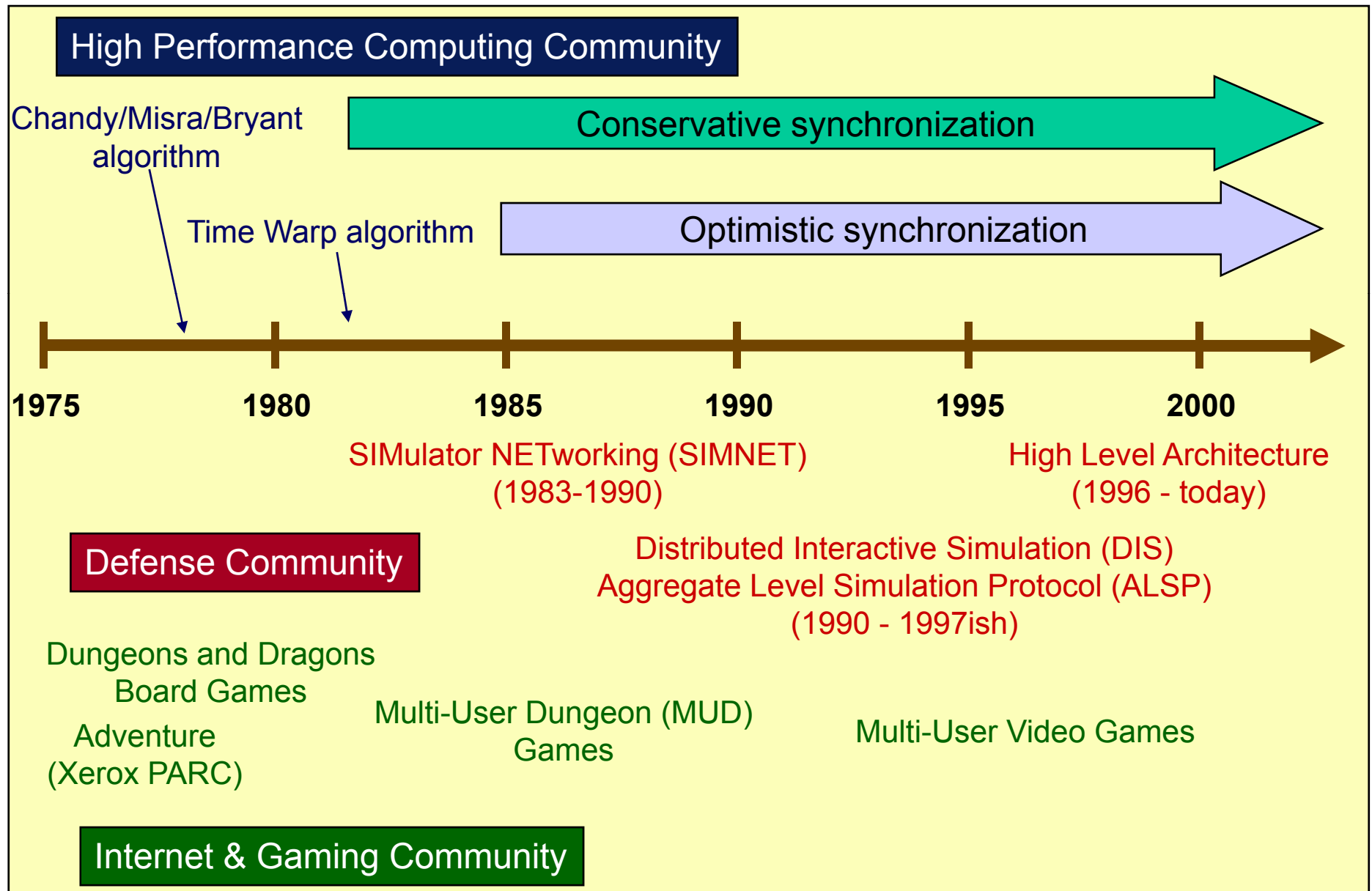
Con:

- ◆ state saving overhead may severely degrade performance
- ◆ rollback thrashing may occur (though a variety of solutions exist)
- ◆ implementation is generally more complex and difficult to debug than conservative mechanisms; careful implementation is required or poor performance may result
- ◆ must be able to recover from exceptions (may be subsequently rolled back)

Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - What is HLA?
 - Federate vs. RTI
- ◆ HLA/RTI Management Areas
- ◆ Time Management
 - Rationale for Time Management
 - Goal of Time Management
 - Implementation of Time Management
 - Regulating and Constrained Federate
 - Time Advancement
 - Lookahead
- ◆ Data Distribution Management

Historic Perspective

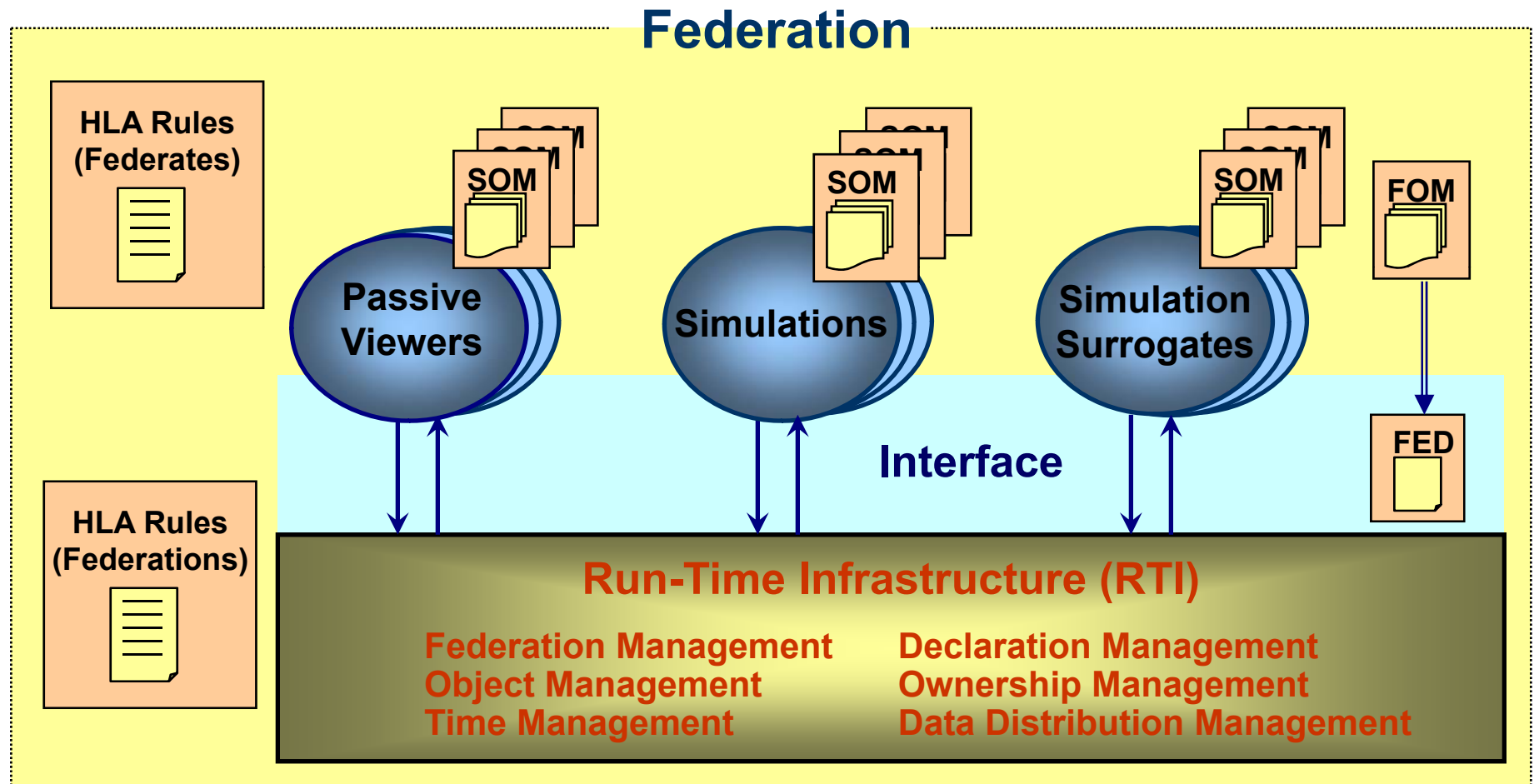


Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
 - **Rationale for Time Management**
 - **Goal of Time Management**
 - **Implementation of Time Management**
 - **Regulating and Constrained Federate**
 - **Time Advancement**
 - **Lookahead**
- ◆ Data Distribution Management

What is HLA?

- ◆ An HLA federation has software and data components

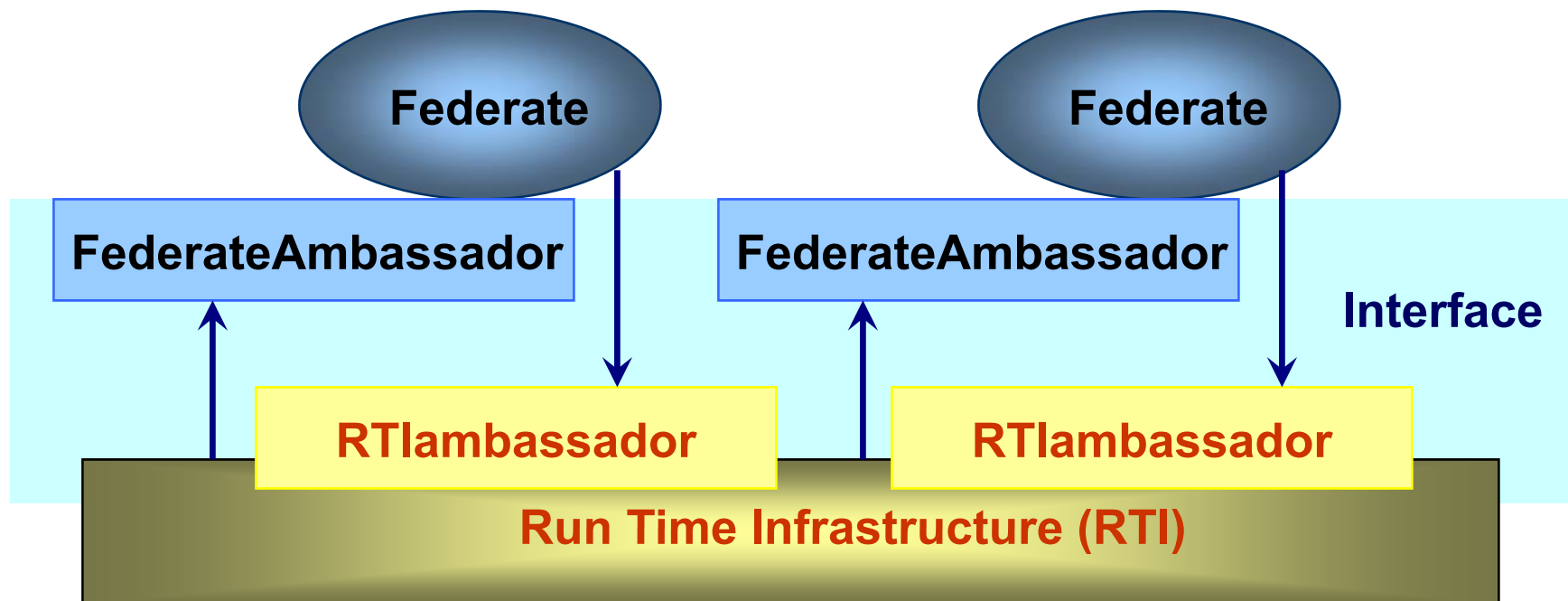


Federate vs. RTI

- ◆ Interface Specification defines not just the interface RTI presents to federates, but also the interface federates present to RTI.
 - RTI offers an interface called **RTIambassador** to each federate.
 - Services defined in RTIambassador are called **federate-initiated services**.
 - Each federate also presents an interface called **FederateAmbassador** to the RTI.
 - Services defined in FederateAmbassador are called **RTI-initiated services**.
- ◆ The HLA partitions functions in a federation between simulation-specific functions and the infrastructure.

Federate vs. RTI

- ◆ HLA splits functions between simulations and runtime infrastructure

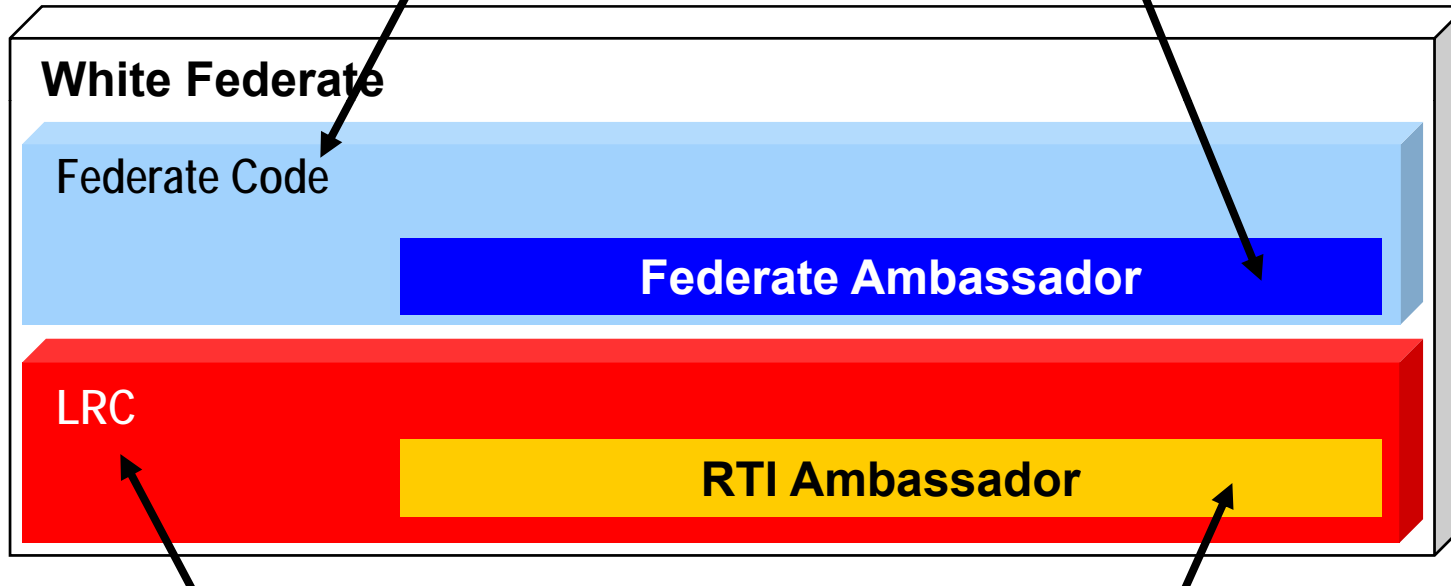


Federate vs. RTI

◆ RTI and Federate Code Responsibilities

The Federate's Code provides internal functionality.

The Federate's code must define the abstract `RTI::FederateAmbassador` class



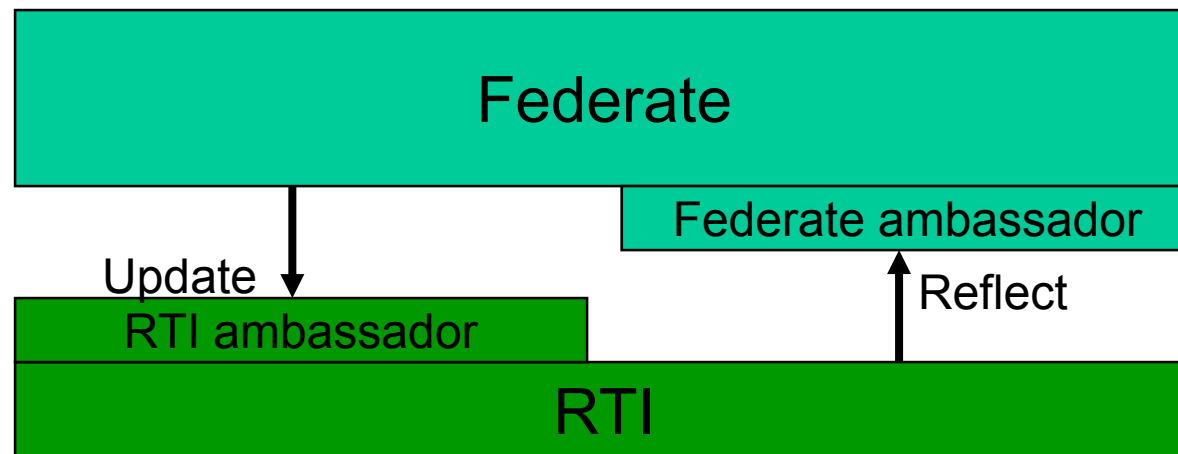
The Local RTI Components (LRC) provide external functionality as specified by the Spec.

The LRC include the methods for the `RTI::RTIambassador` class

Federate vs. RTI

Some services are initiated by the federate, others by the RTI

- ◆ Federate invoked services
 - **Publish, subscribe, register, update**
 - **Not unlike calls to a library**
 - **Procedures defined in the RTI ambassador**
- ◆ RTI invoked services
 - **Discover, reflect**
 - **Federate defined procedures, in Federate Ambassador**



Outline – HLA And Distributed/Federated Simulation

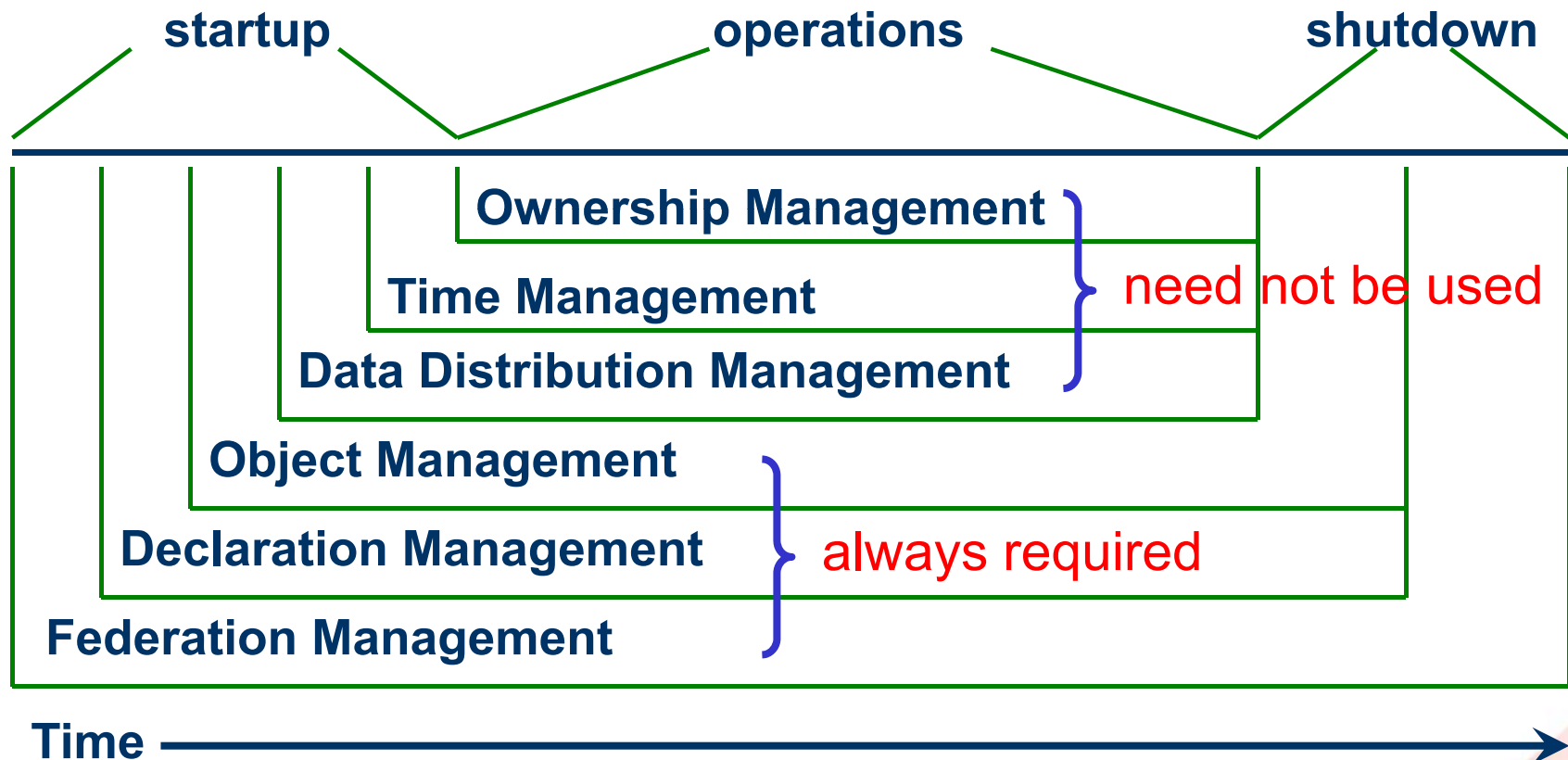
- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
 - **Rationale for Time Management**
 - **Goal of Time Management**
 - **Implementation of Time Management**
 - **Regulating and Constrained Federate**
 - **Time Advancement**
 - **Lookahead**
- ◆ Data Distribution Management

Run-Time Infrastructure Management Areas

Category	Functionality
Federation Management	Create and delete federation executions join and resign federation executions control checkpoint, pause, resume, restart
Declaration Management	Establish intent to publish and subscribe to object attributes and interactions
Object Management	Create and delete object instances Control attribute and interaction publication Create and delete object reflections
Ownership Management	Transfer ownership of object attributes
Time Management	Coordinate the advance of logical time and its relationship to real time
Data Distribution Management	Supports efficient routing of data

Run-Time Infrastructure Management Areas

◆ Six Management Areas



A Typical Federation Execution

initialize federation

- Create Federation Execution (Federation Mgt)
- Join Federation Execution (Federation Mgt)

declare objects of common interest among federates

- Publish Object Class Attributes (Declaration Mgt)
- Subscribe Object Class Attributes (Declaration Mgt)

exchange information

- Update/Reflect Attribute Values (Object Mgt)
- Send/Receive Interaction (Object Mgt)
- Time Advance Request, Time Advance Grant (Time Mgt)
- Request Attribute Ownership Assumption (Ownership Mgt)
- Send Interaction with Regions (Data Distribution Mgt)

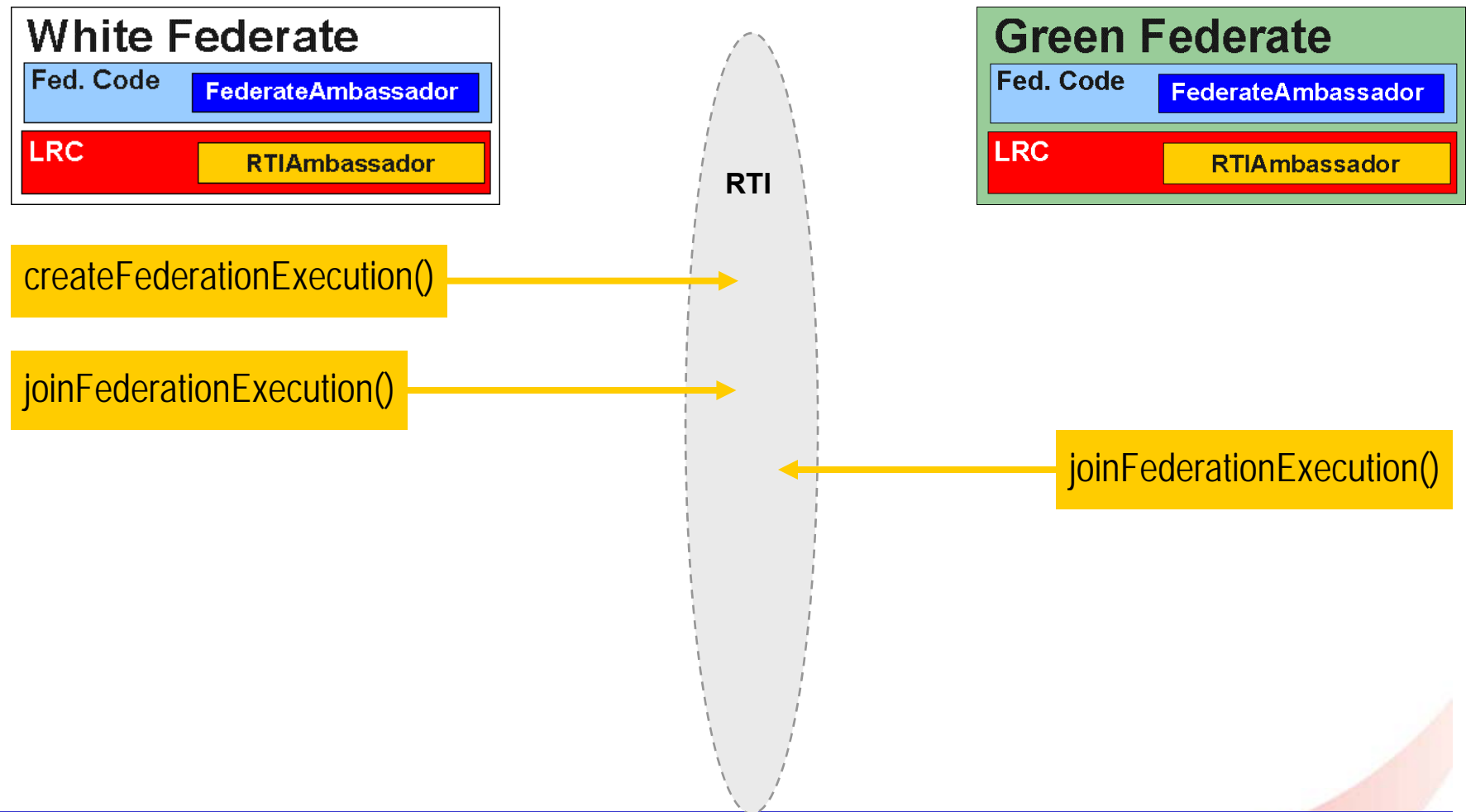
terminate execution

- Resign Federation Execution (Federation Mgt)
- Destroy Federation Execution (Federation Mgt)

Federation Management

- ◆ **Federation Management** allows the creation of the federation execution and the coordination of federation wide activities
 - **Creating & Destroying the Federation Execution**
 - **Joining & Resigning the Federation Execution**
 - **Coordination of Federation wide Synchronization**
 - **Coordination of Federation Save & Restore**
- ◆ An RTI can support several federation executions simultaneously, each distinguished by a unique name
- ◆ Each Federation Execution must be associated with a FOM (specifically, the FED file)

Creating & Joining the Federation Execution



Declaration Management

- ◆ **Declaration Management** allows federates to specify the kind of data they will send (**publish**) or receive (**subscribe**)
- ◆ The data can be specified by:
 - **Interaction class**
 - **Object class and attribute names**
- ◆ Declaration Management Services in the RTI include:
 - **Publishing & Unpublishing Interaction Classes**
 - **Subscribing & Unsubscribing Interaction Classes**
 - **Publishing & Unpublishing Object Classes**
 - **Subscribing & Unsubscribing Object Classes**

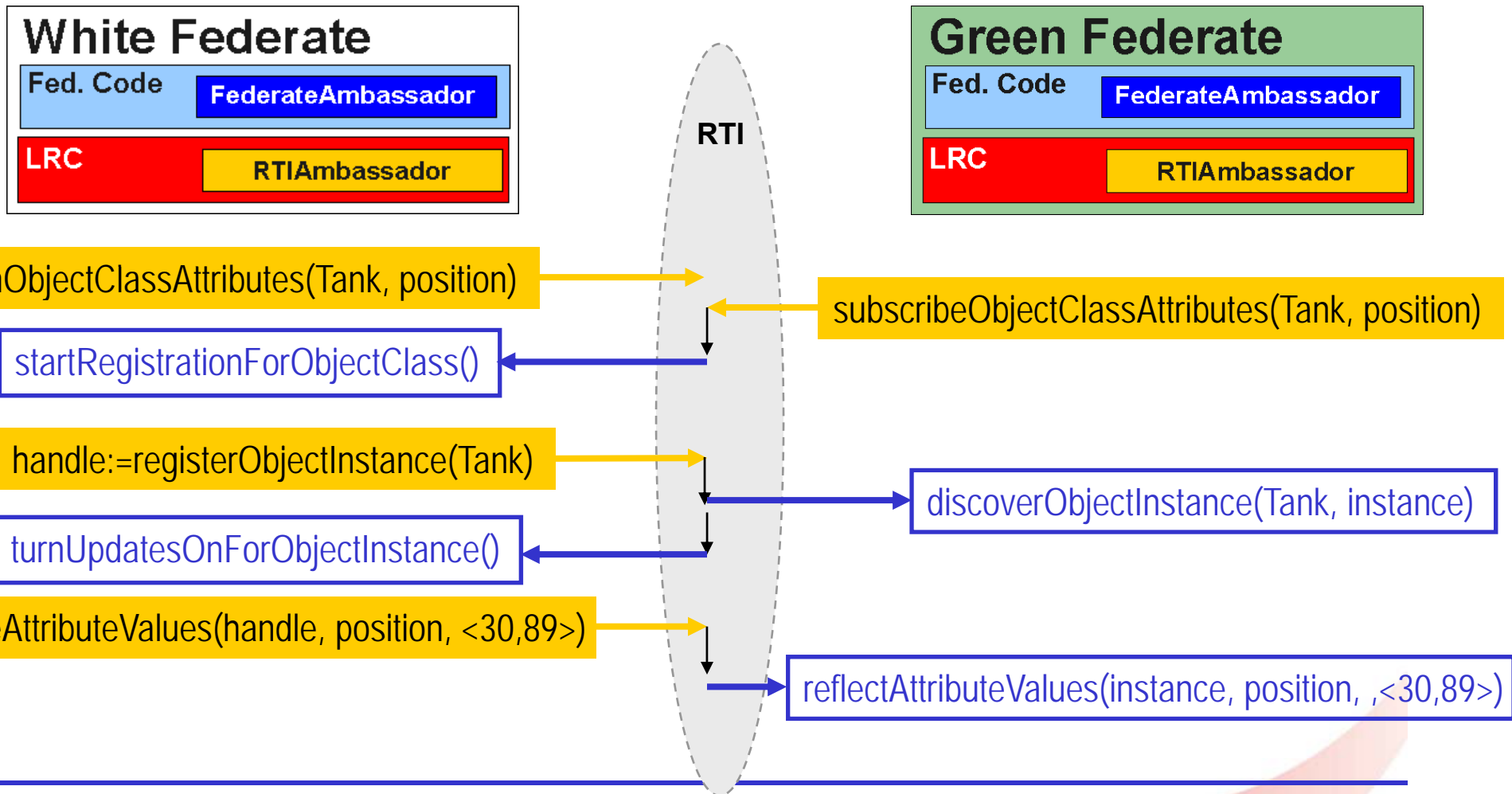
Object Management

- ◆ **Object Management** provides the actual exchange of data through the RTI and supports the lifecycle of objects - creating (**registering**) new object instances, **updating** their attributes and **deleting** objects
- ◆ Object Management Services for Interactions include:
 - **Sending & Receiving Interactions**
- ◆ Object Management Services for Objects include:
 - **Registering & Discovering Object Instances**
 - **Updating & Reflecting Object Attribute Values**
 - **Deleting & Removing Object Instances**

Message Passing Alternatives

- ◆ Traditional message passing mechanisms: Sender explicitly identifies receivers
 - Destination process, port, etc.
 - Poorly suited for federated simulations
- ◆ Broadcast
 - Receiver discards messages not relevant to it
 - Used in SIMNET, DIS (initially)
 - Doesn't scale well to large federations
- ◆ Publication / Subscription mechanisms
 - Analogous to newsgroups
 - Producer of information has a means of describing data it is producing
 - Receiver has a means of describing the data it is interested in receiving
 - Used in High Level Architecture (HLA)

An Example



Ownership Management

- ◆ **Ownership Management** allows federates to transfer ownership of object instance attributes
 - An instance attribute is owned by at most one federate at any given time
 - The federate that registers an object instance initially owns those attributes it publishes
 - A federate must own an instance attribute before it can update it
 - Different federates may own different attributes of the same object instance
 - Federates may transfer ownership in accordance with the federation design

Time Management

- ◆ **Time Management** allows the proper ordering of events between federates executing their own threads of control
 - Ordering of events is expressed using **Logical Time**
 - Each federate can advance its logical time in coordination with other federates
 - The delivery of timestamped events can be controlled so that a federate never receives events from other federates in its “past”
- ◆ Time Management is designed to support federates with differing ordering and delivery requirements

Data Distribution Management

- ◆ **Data Distribution Management (DDM)** provides detailed control over producer/consumer relationships
 - **Declaration Management** describes those relationships in terms of interaction classes and object classes
 - **Data Distribution Management** allows the relationships to be redefined for individual interactions and object instance attributes
- ◆ DDM can significantly reduce the bandwidth requirements for federations with:
 - a large number of interaction and/or object classes
 - a large number of object instances per object class
 - frequent interactions and/or attribute updates

Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
 - **Rationale for Time Management**
 - **Goal of Time Management**
 - **Implementation of Time Management**
 - **Regulating and Constrained Federate**
 - **Time Advancement**
 - **Lookahead**
- ◆ Data Distribution Management

Rationale for Time Management

◆ Avoiding Causality Errors

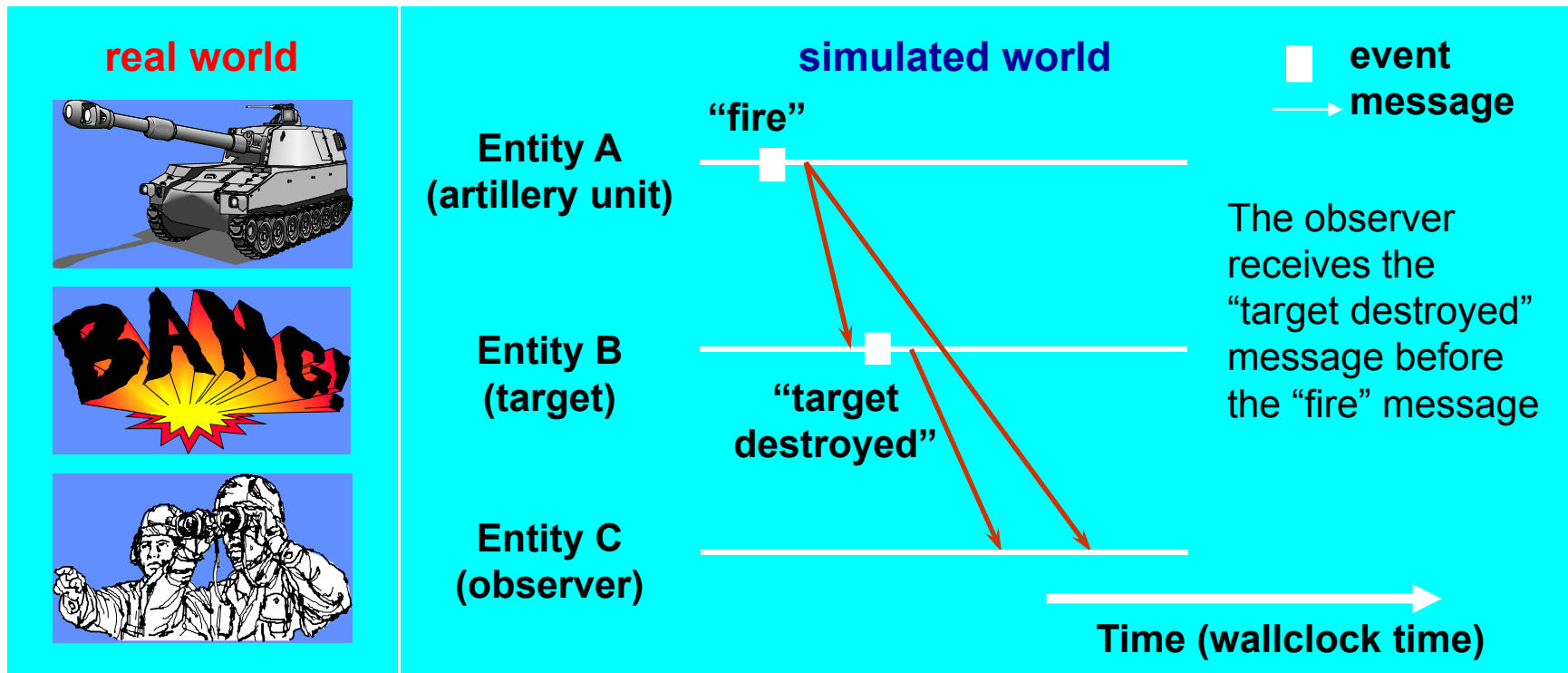
- **Without Time Management, a simulator may incorrectly reproduce temporal aspects of the real world system being modeled**
- **The importance of correctly reproducing temporal aspects depends on the simulation application – in a training simulation, non-causal ordering of events may be acceptable if they are not perceptible or occur infrequently**

◆ Repeatability of the Simulation

- **Without Time Management, repeated executions of the simulation with the same initial state and external inputs might give entirely different results**

Rationale for Time Management

- ◆ Causality – the future should not affect the past



Goal of Time Management

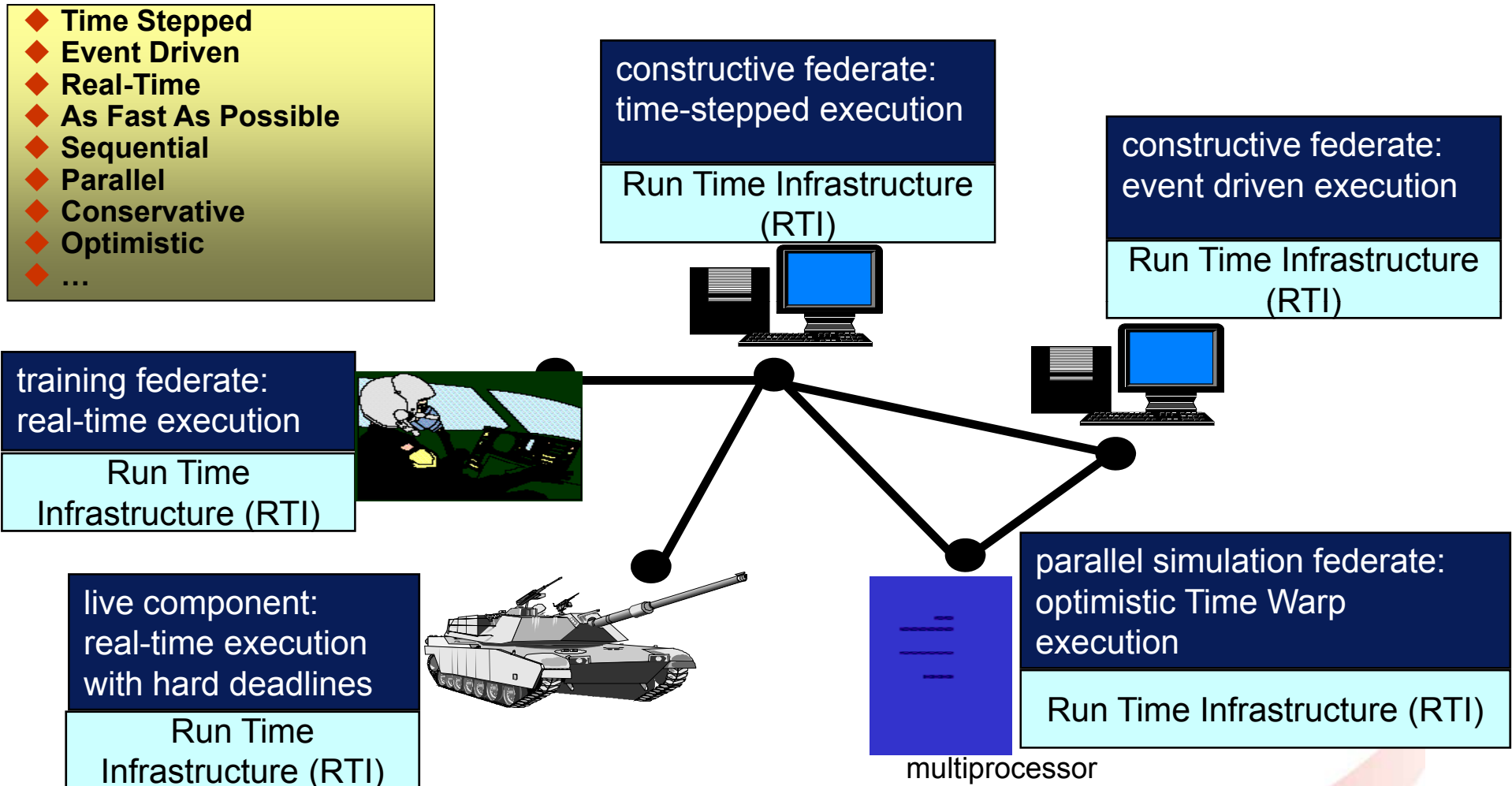
- ◆ A key goal of the HLA Time Management services is to facilitate reuse and interoperability of federates:
 - Facilitate reuse by allowing a federate to adopt a wide range of internal time management mechanisms
 - Support interoperability by allowing federates with different internal time management mechanisms to be combined in a single federation execution
 - ◆ This requires **Time Management Transparency** – the internal time management mechanism used within each federate must not be visible to other federates
 - ◆ Federates do not explicitly indicate to the RTI the internal time management mechanism being used, and can even change the mechanism during execution

Goal of Time Management

◆ Time Management Schemes

- **No Explicit Time Management:** each federate advances at its own rate
- **Time Stepped:** each federate advances its simulation time by fixed increments
- **Conservative (Event Driven) Synchronization:** federates advance simulation time in such a way as to avoid causality errors
- **Optimistic (Event Driven) Synchronization:** federates are free to advance simulation time but must recover from causality errors
- **Real-Time:** federates advance simulation time in synchrony with wall-clock time

Integrating Time Management Schemes



Goal: Provide services for interoperability among simulators with different internal time management mechanisms in a single federation execution

Implementation of Time Management

- ◆ Time Management controls the sending and receiving of **events** – these correspond to:
 - **Sending/Receiving Interactions**
 - **Updating/Reflecting Attribute Values**
 - **Deleting/Removing Object Instances**
- ◆ Other services are not affected by Time Management
- ◆ Time Management in any federation must be realized jointly by the federates and the RTI
 - **Key question: what functionality should be implemented within the RTI, and what should be realized within individual federates?**

Message Ordering

- ◆ The HLA provides two types of message (event) ordering:
 - **Receive Order** (unordered): messages are passed to the receiving federate in an arbitrary order
 - ◆ Receive order minimizes latency but does not prevent causality errors
 - **Time Stamp Order** (TSO): the sender assigns a time stamp to the message and messages that are passed to each federate have non-decreasing time stamps
 - ◆ TSO prevents causality errors, but has a higher latency
 - ◆ Note: TSO messages need **not** be **sent** in time stamp order

Message Ordering

Property	Receive Order (RO)	Time Stamp Order (TSO)
Latency	<i>low</i>	<i>higher</i>
reproduce before and after relationships?	<i>no</i>	<i>yes</i>
all federates see same ordering of events?	<i>no</i>	<i>yes</i>
execution repeatable?	<i>no</i>	<i>yes</i>
typical applications	<i>training, T&E</i>	<i>analysis</i>

Regulating and Constrained Federates

- ◆ Federates must declare their intent to use time management services by setting their *time regulating* and/or *time constrained* flags
 - **Time regulating federates:** can send TSO messages
 - ◆ Can prevent other federates from advancing their logical time
 - ◆ Enable Time Regulation ... **Time Regulation Enabled †**
 - ◆ Disable Time Regulation
 - **Time constrained federates:** can receive TSO messages
 - ◆ Time advances are constrained by other federates
 - ◆ Enable Time Constrained ... **Time Constrained Enabled †**
 - ◆ Disable Time Constrained

Regulating and Constrained Federates

- ◆ A federate may be regulating, constrained, both or neither

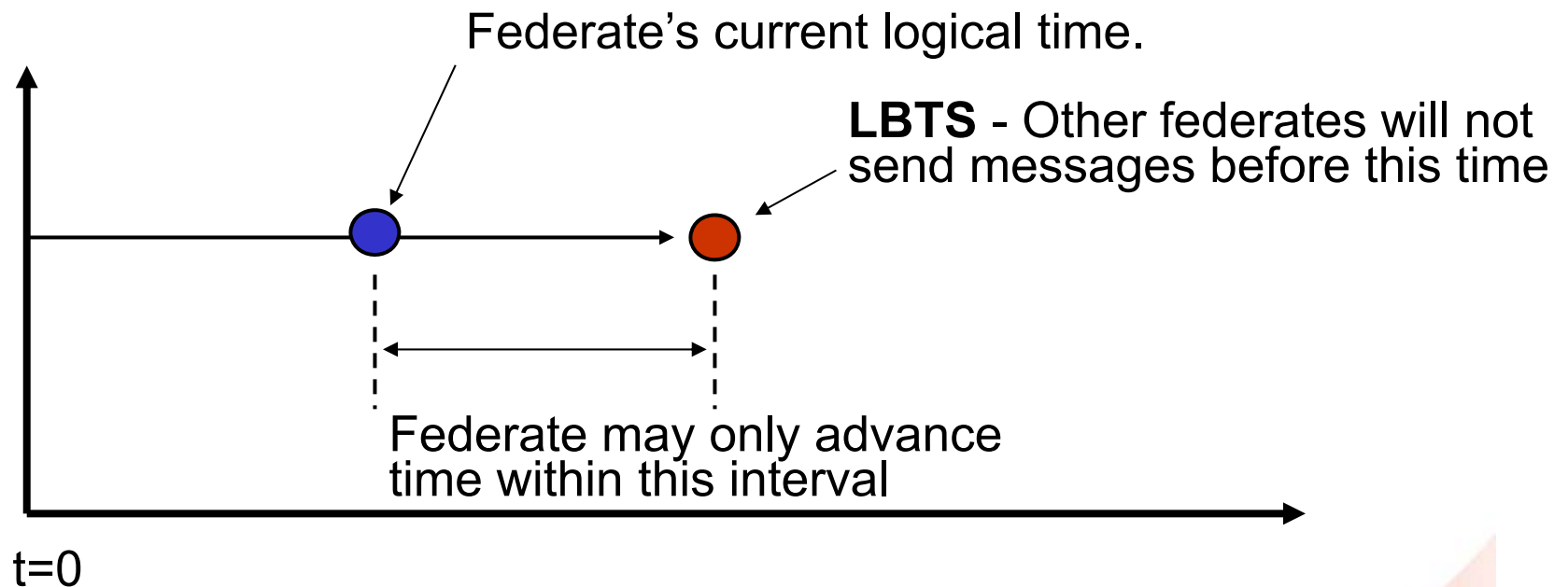
		Time Regulating	
		true	false
Time Constrained	true	Strictly Time Synchronized conservative & optimistic	Viewer or FMT stays synchronized to the federation, but generates no events
	false	Unconstrained federate (e.g. message source) operating with conservative federates	Externally Synchronized Simulation (e.g. DIS) or no Time Management

Lookahead and LBTS

- ◆ **Lookahead** is required otherwise deadlock may occur.
 - A Time Regulating federate guarantees that it will not send events with time stamp less than its current simulation time plus lookahead
 - For a time-stepped simulation, lookahead is usually the step size (fixed increment), because a federate can only schedule events into the next time step or later
- ◆ The RTI computes a **Lower Bound on Time Stamp (LBTS)** for each federate
 - LBTS is a lower bound on the time stamps of events which may be delivered to that federate later in the execution
 - The RTI prevents a federate from advancing its logical time beyond LBTS

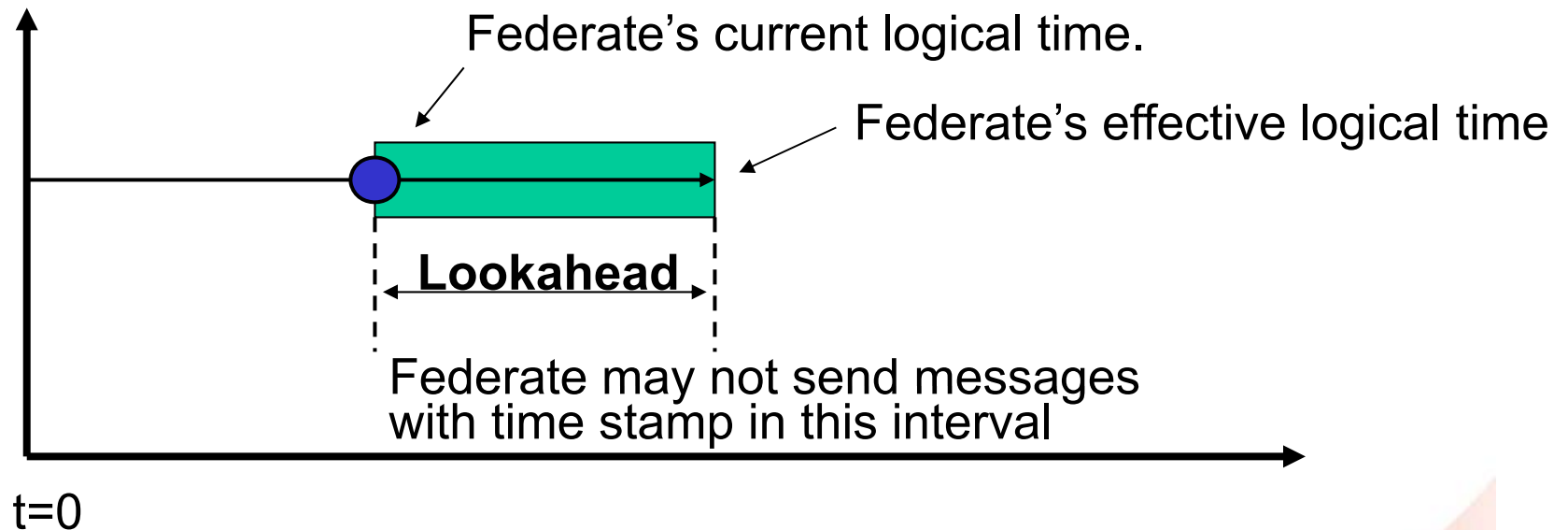
Constrained Federates

- ◆ Time Constrained federates receive TSO data, with messages delivered in order of time stamps

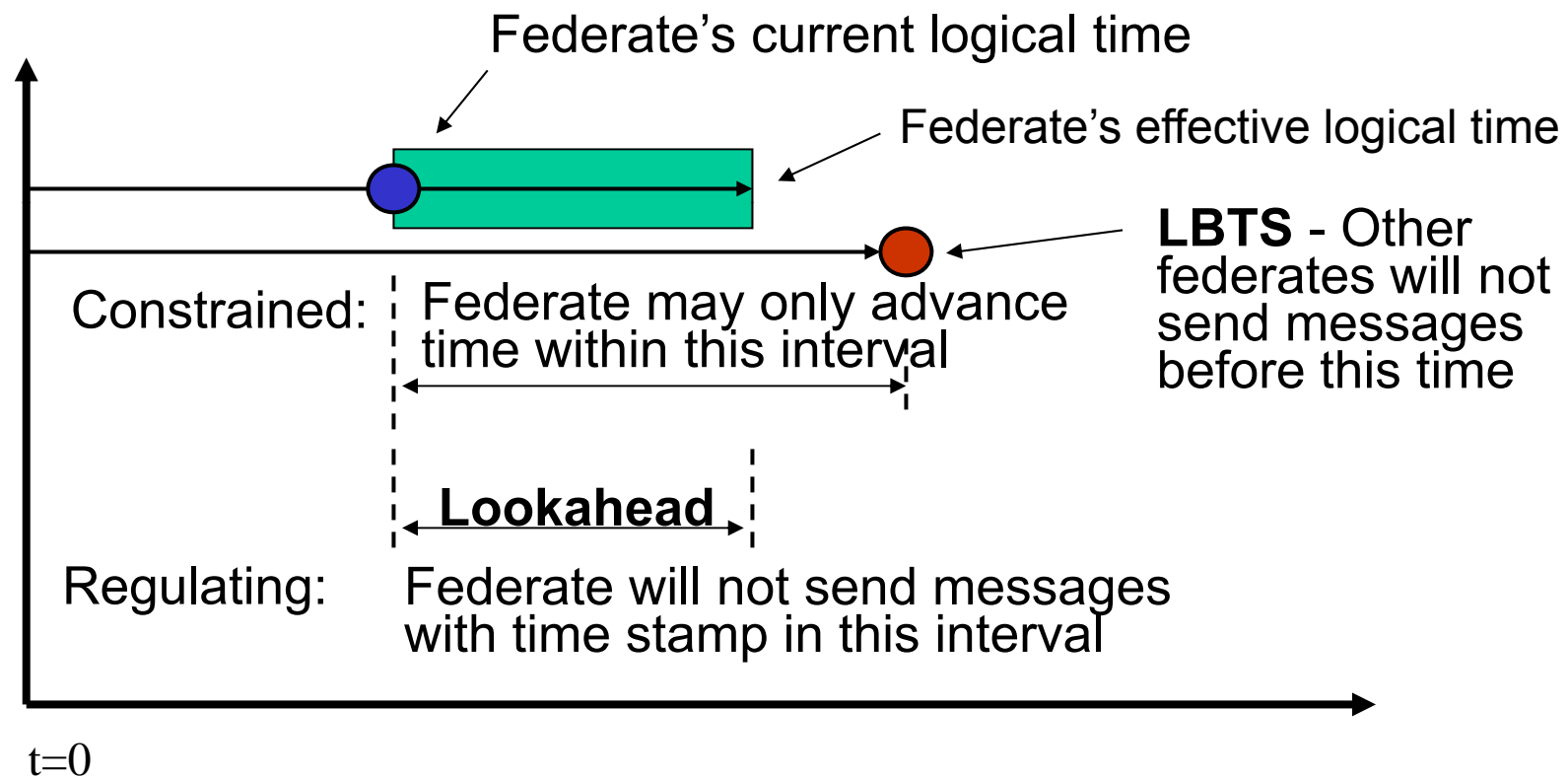


Regulating Federates

- ◆ Time Regulating Federates send TSO data, with messages delivered in order of time stamps



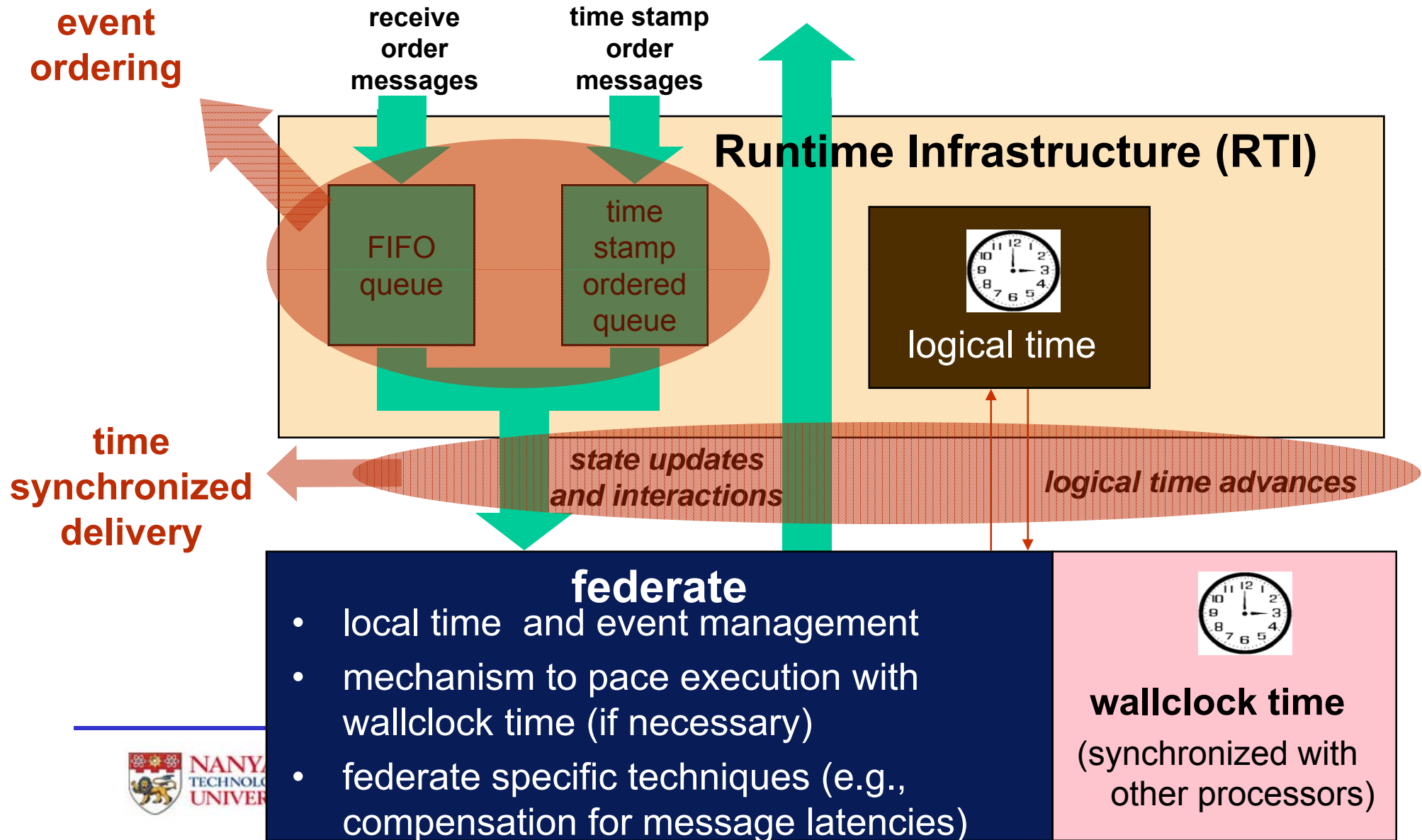
Regulating and Constrained



Advancing Logical Time

- ◆ Both Time Regulating and Time Constrained federates must request the RTI to advance their logical time
 - Coordination of logical time depends on the requests to time advance time, not the sending or receiving of events
 - A federate advances time by invoking a request and waiting for a callback granting a new logical time
 - The federate's logical time advances when a new time is granted, not when it is requested
 - For a time constrained federate, the logical time granted may be smaller than the logical time requested (to avoid the possibility of causality errors)
 - If the federate is not time constrained, the advance will be granted immediately with the logical time requested

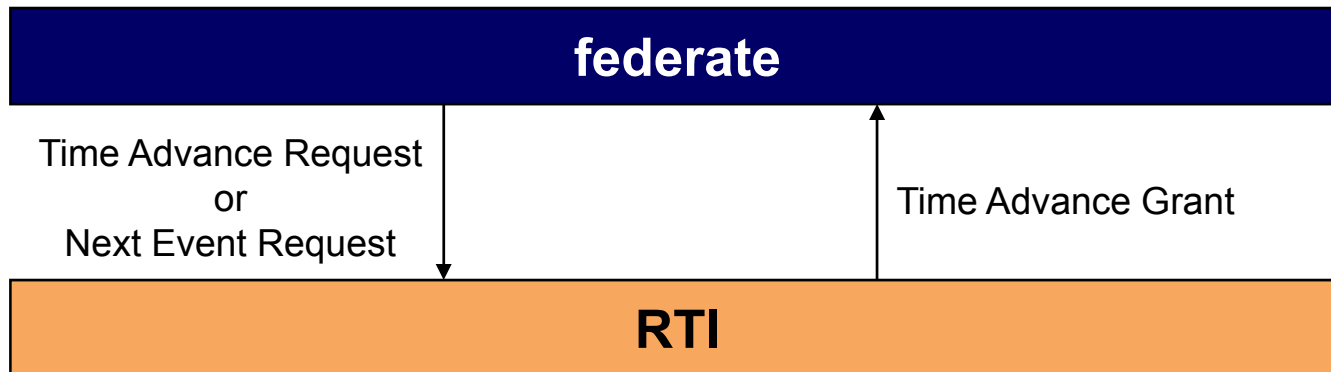
Time Management Services



Time Management Services

HLA Time Management Services define a protocol for federates to advance logical time – logical time only advances when that federate explicitly requests an advance

- ◆ Time Advance Request: *time stepped federates*
- ◆ Next Event Request: *event driven federates*
- ◆ Time Advance Grant: RTI invokes to acknowledge logical time advances



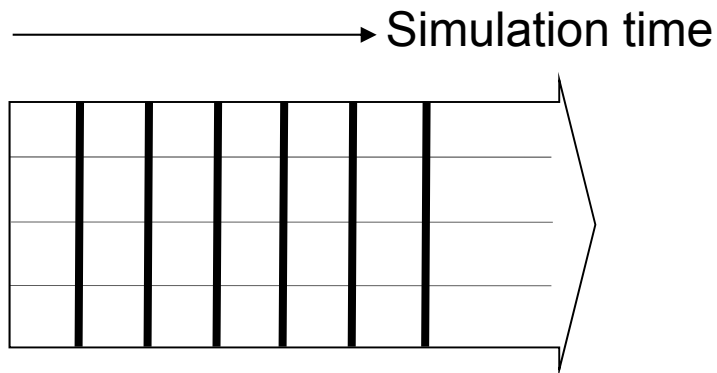
If the logical time of a federate is T , the RTI guarantees no more TSO messages will be passed to the federate with time stamp $< T$

Federates responsible for pacing logical time advances with wallclock time in real-time executions

Time Advance Mechanisms

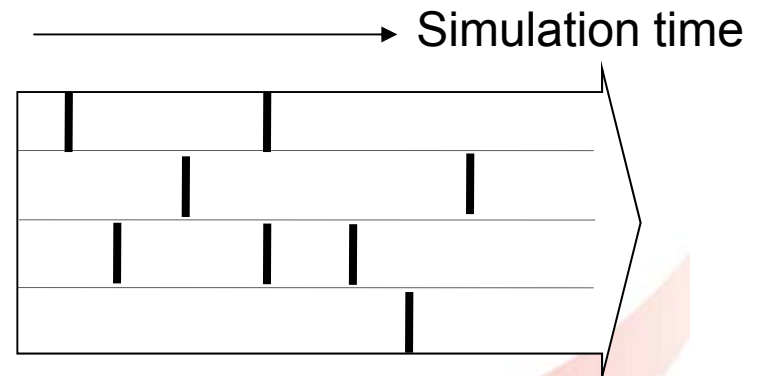
◆ Time Stepped

- All federates advance time with the same time increments
- Federates exchange messages for state updates



◆ Event Driven

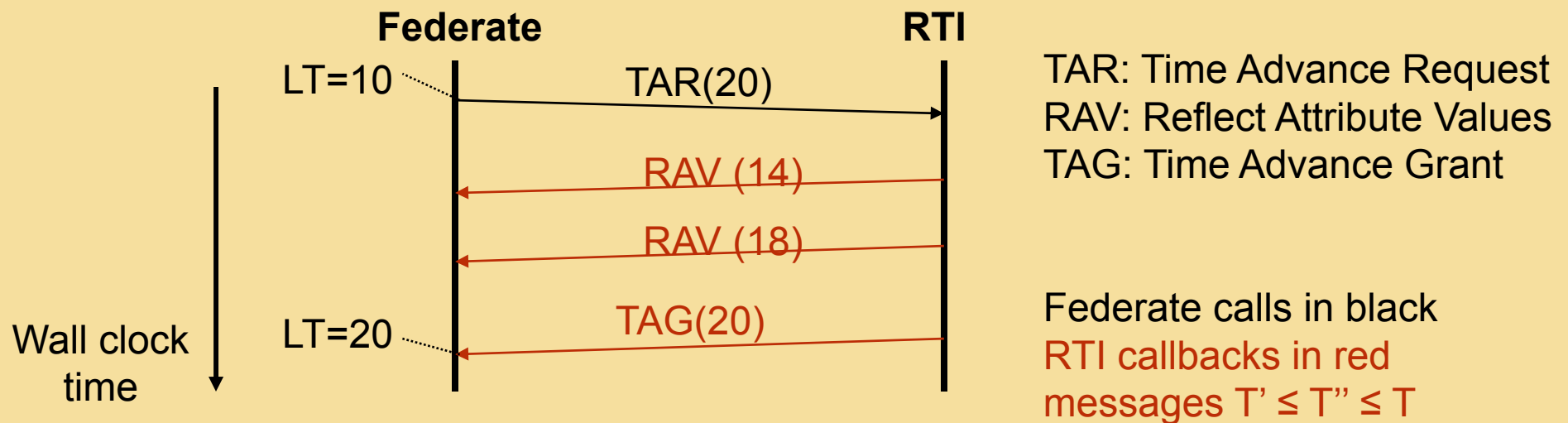
- State updates are scheduled at different times in the future
- Federates exchange events for state updates
- Events are executed in time stamp order



Time Stepped Simulation

- ◆ Federate is at logical time T' and invokes **Time Advance Request (T)** to request its logical time (LT) be advanced to T
- ◆ RTI delivers all TSO messages with time stamp $\leq T$
- ◆ RTI advances federate's time to T , invokes **Time Advance Grant (T)** when it can guarantee all TSO messages with time stamp $T'' \leq T$ have been delivered
- ◆ Grant time always matches the requested time

Typical execution sequence



Using “tick”

- ◆ In some RTIs **tick** is used to wait for service completion to pass control to the federate ambassador
- ◆ Arguments: **minimum** and **maximum waiting time**

- **timeAdvGrant** is initialized to false and set true by various callback functions when a new logical time has been granted

```
timeAdvGrant = FALSE;  
rtiAmb.enableTimeConstrained();  
while ( !timeAdvGrant ) rtiAmb.tick(0.01, 1.0);  
...  
timeAdvGrant = FALSE;  
rtiAmb.enableTimeRegulation ( logicalTime, lookahead);  
while ( !timeAdvGrant ) rtiAmb.tick(0.01, 1.0);
```

Federate Code

Sequential Simulator

```
t = initial simulation time;
while (!end of simulation) {
    update local simulation state;
    t = t + s; // s = step size
}
```

FedAmb functions called by RTI:

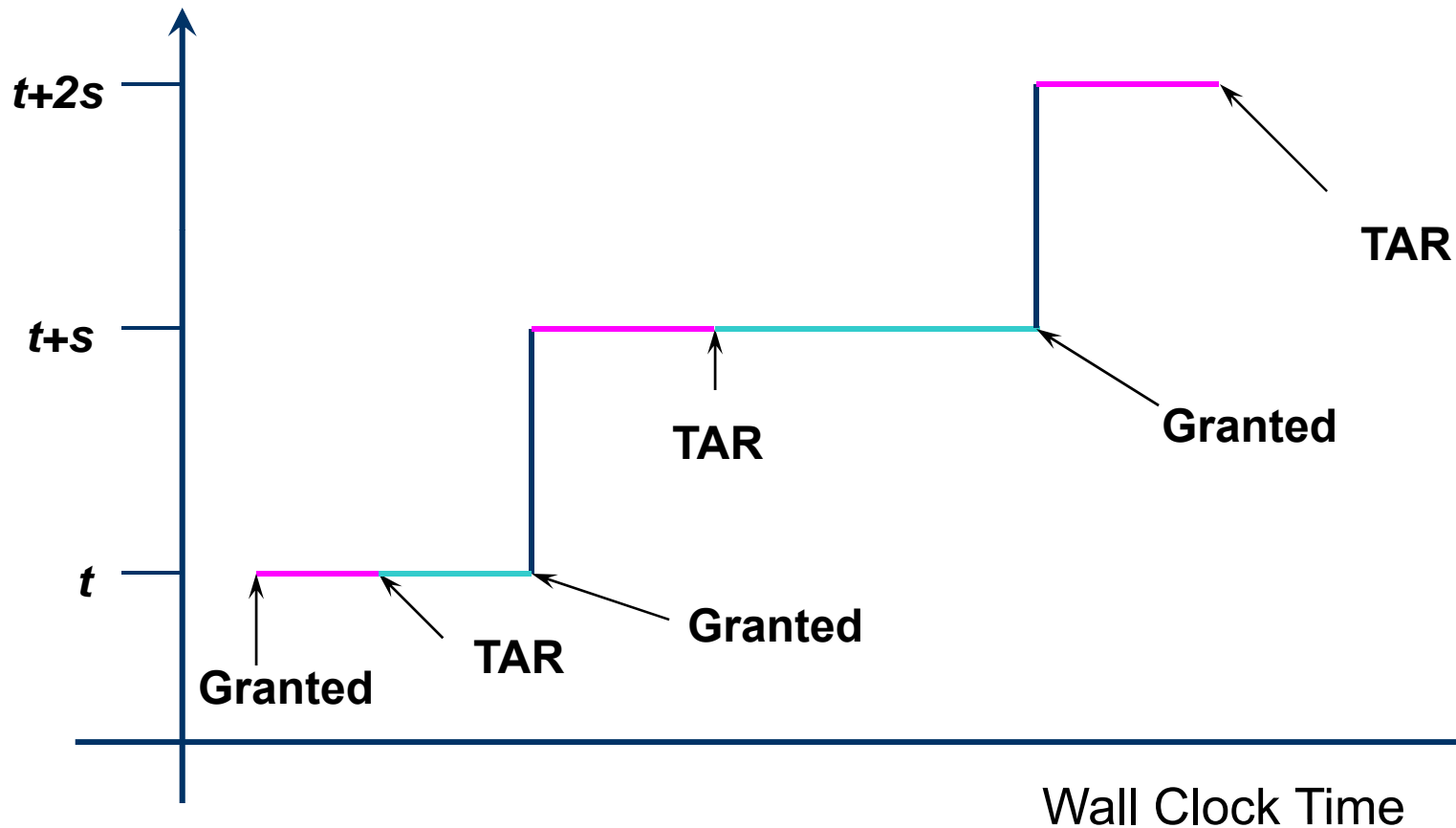
```
reflectAttributeValues †
    update local state
timeAdvanceGrant (...)
    timeAdvGrant = TRUE;
```

Federated Simulator

```
t = initial simulation time;
while (!end of simulation) {
    update local simulation state;
    updateAttributeValues( ... );
    timeAdvGrant = FALSE;
    timeAdvanceRequest( t + s );
    while( !timeAdvGrant )
        tick();
    t = t + s;
}
```

Time Stepped Simulation

Simulation Time

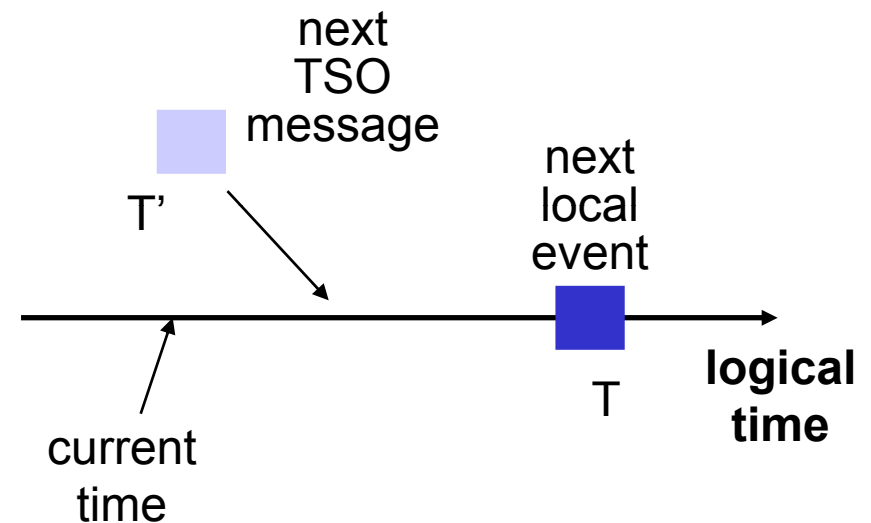
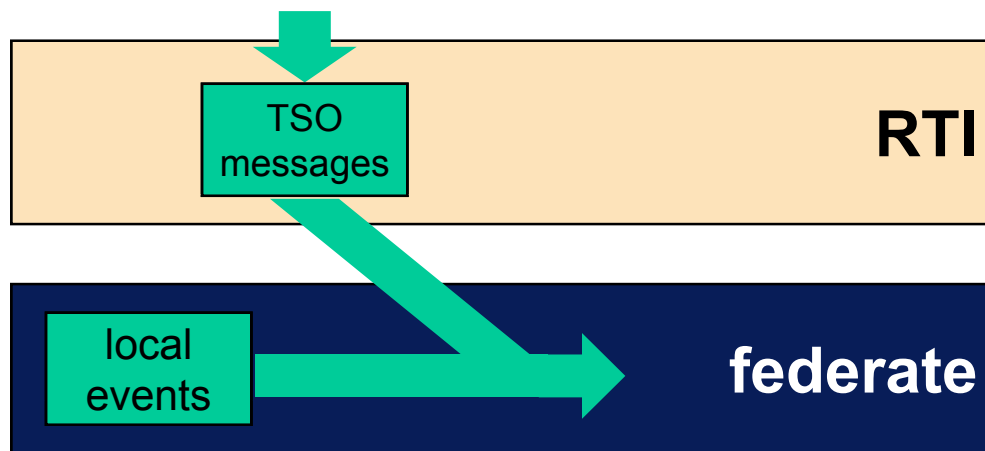


Event Driven Simulation

- ◆ An event driven federate has its own internal event list which holds future events that have been scheduled
 - The simulation proceeds by processing the next scheduled event (the one with the smallest time stamp)
 - Processing this event may schedule other events:
 - ◆ for the same federate (inserted in the event list)
 - ◆ for other federates (to be sent through the RTI)
 - The federate must merge internal events in its own event list with events received from other federates to obtain a single stream of events ordered by time stamp
 - ◆ The federation does not maintain a combined event list
 - Merging of events must ensure the federate does not receive an event in its past

Event Driven Simulation

- ◆ Goal: process all events (local and incoming TSO messages) in time stamp order



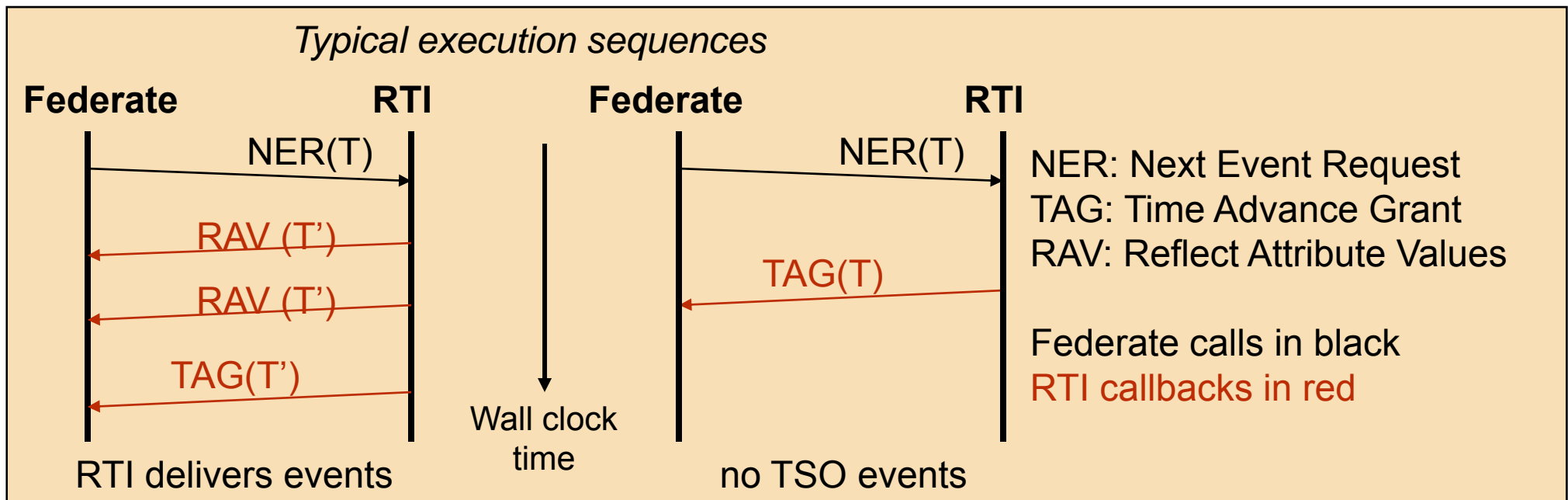
- ◆ Federate: next local event has time stamp T
- ◆ If no TSO messages with time stamp $< T$, advance to T , process local event
- ◆ If there is a TSO message with time stamp $T' \leq T$, advance to T' and process TSO message

Event Driven Simulation

- ◆ Federate invokes **Next Event Request (T)** to request its logical time be advanced to time stamp of next TSO message, or T, whichever is smaller
- ◆ If next TSO message has time stamp $T' \leq T$
 - RTI delivers next TSO message, and all others with time stamp T'
 - RTI issues **Time Advance Grant (T')**
- ◆ Else

RTI advances federate's time to T, invokes **Time Advance Grant (T)**

Typical execution sequences



Federate Code

Sequential Simulator

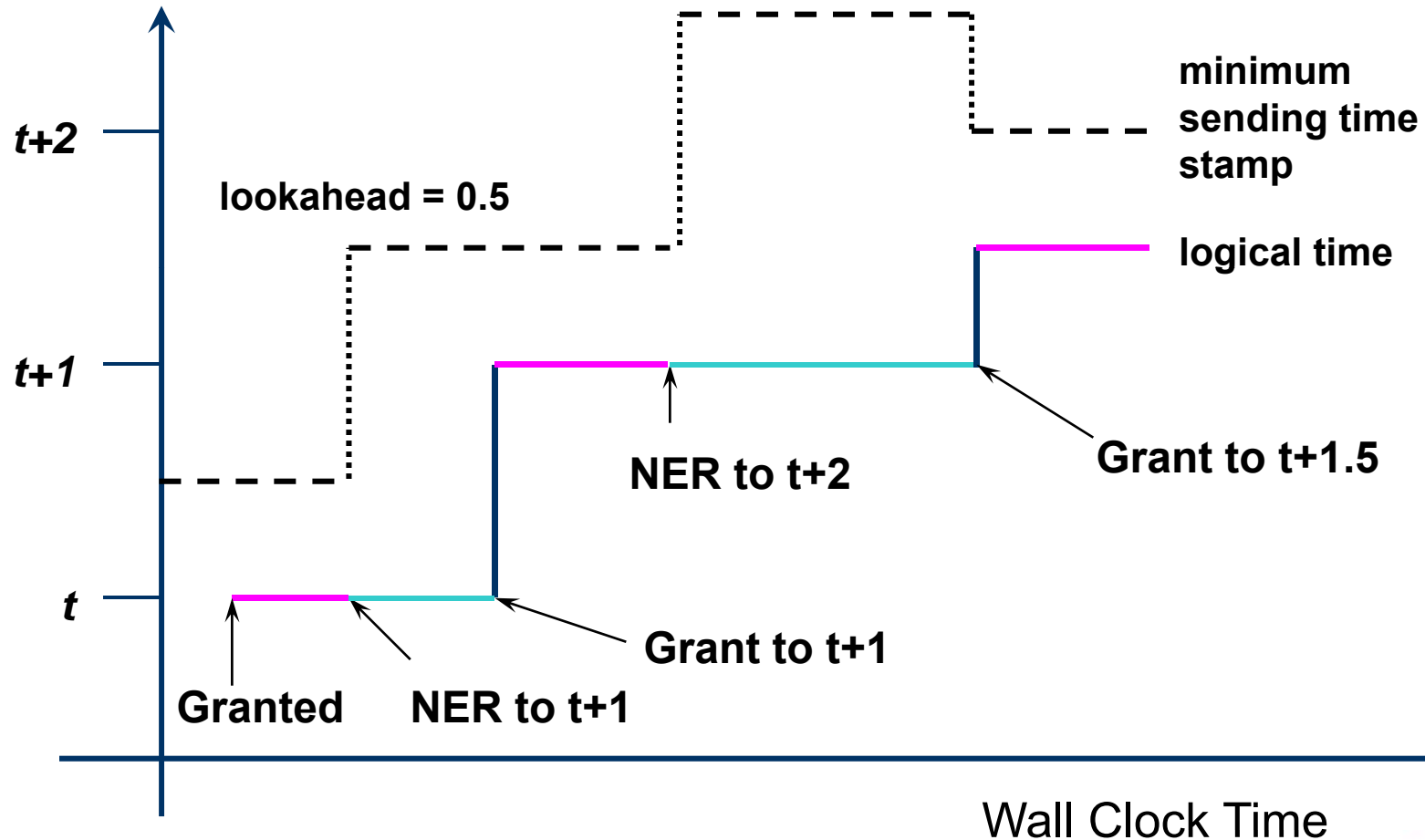
```
t = initial simulation time;
initialize event list (FEL);
while (!end of simulation) {
    t = time of next event in FEL;
    process next event;
    FedAmb functions called by RTI:
    reflectAttributeValues †
        insert event into FEL
timeAdvanceGrant (...)
    timeAdvGrant = TRUE;
```

Federated Simulator

```
t = initial simulation time;
initialize event list (FEL);
while (!end of simulation) {
    t = time of next event in FEL;
    timeAdvGrant = FALSE;
    nextEventRequest( t );
    while( !timeAdvGrant )
        tick();
    process next event in FEL;
}
```


Event Driven Simulation

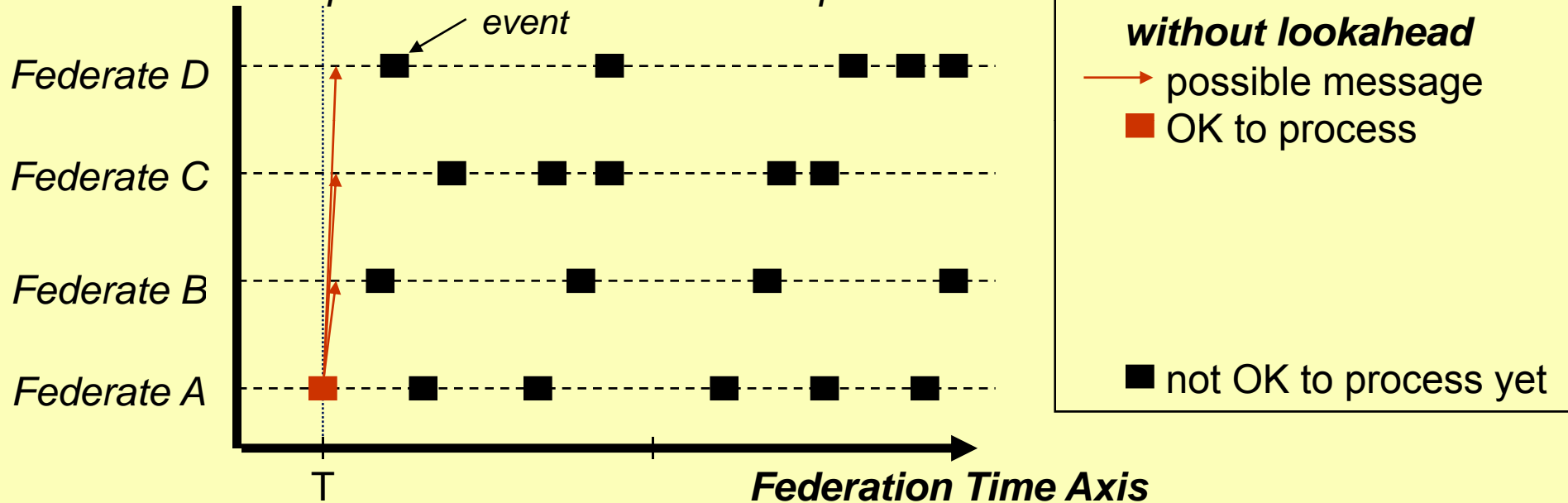
Simulation Time



Lookahead

NER: if no lookahead concurrency limited to events containing exactly the same time stamp

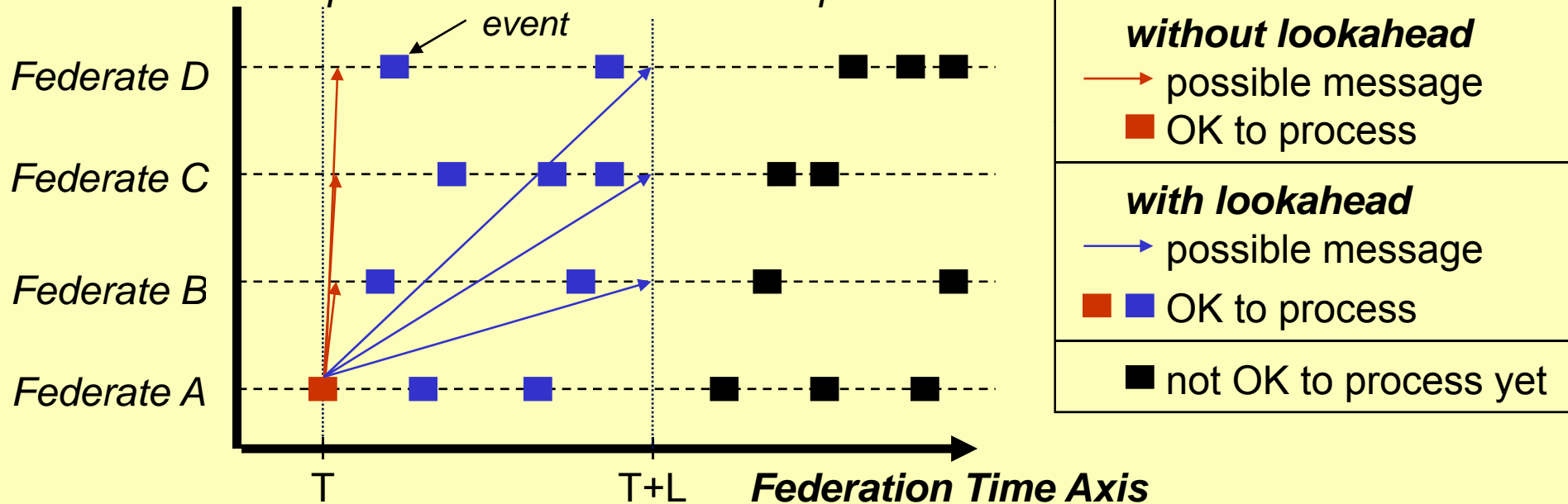
each federate must process events in time stamp order



Lookahead

NER: if no lookahead concurrency limited to events containing exactly the same time stamp

each federate must process events in time stamp order



Each federate using logical time declares a lookahead value L ; any TSO message sent by the federate must have a time stamp \geq the federate's current time + L

Lookahead is necessary to allow concurrent processing of events with different time stamps (unless optimistic event processing is used)

Lookahead

- ◆ No federate may generate an event with time stamp less than its current logical time plus its lookahead
- ◆ Choosing Lookahead:
 - **How quickly can one federate affect another?**
 - ◆ e.g., in a computer network simulation, there is a minimum simulation time required for transmitting a data packet
 - **How quickly can one federate react to another?**
 - ◆ e.g. in a manufacturing simulation, a machine has a minimum delay in responding to an operator's command

Federate and RTI Guarantees

- ◆ Federate at logical time T (with lookahead L)
- ◆ All outgoing TSO messages must have time stamp $\geq T+L$ ($L>0$)
- ◆ Time Advance Request (T)
 - Once invoked, federate cannot send messages with time stamp less than T plus lookahead
- ◆ Next Event Request (T)
 - Once invoked, federate cannot send messages with time stamp less than T plus the federate's lookahead unless a grant is issued to a time less than T
- ◆ Time Advance Grant (T) (after TAR or NER service)
 - All TSO messages with time stamp less than or equal to T have been delivered

RTI 1.3 vs. IEEE 1516

RTI 1.3

◆ LBTS: Lower Bound on Time Stamp

- Denotes the earliest time stamp of any event that a federate can receive anytime in the future
- Implies the federate can safely advance its local logical time to LBTS

IEEE 1516

◆ GALT: Greatest Available Logical Time

- Denotes the time to which a federate can safely advance its logical time
- No incoming (externally generated) events will have time stamps less than GALT
- More general definition than LBTS
- Can safely use LBTS for GALT

Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
- ◆ Data Distribution Management
 - **Concepts of Data Distribution**
 - **Data Distribution Management (DDM) and Services**
 - **Using Interactions with DDM**
 - **Using Objects with DDM**

Concepts of Data Distribution

- ◆ **Basic Question:** When a federate generates information (e.g., state updates) that may be of interest to other federates, who should receive the message?
- ◆ **Simple Approach:** Broadcast each message, receiver responsible for filtering (deleting) unwanted messages – adopted in SIMNET and DIS (initially)
 - **Not scalable:** can use large amount of communication bandwidth if a large number of federates
 - **Time spent receiving and filtering unwanted messages becomes a bottleneck**

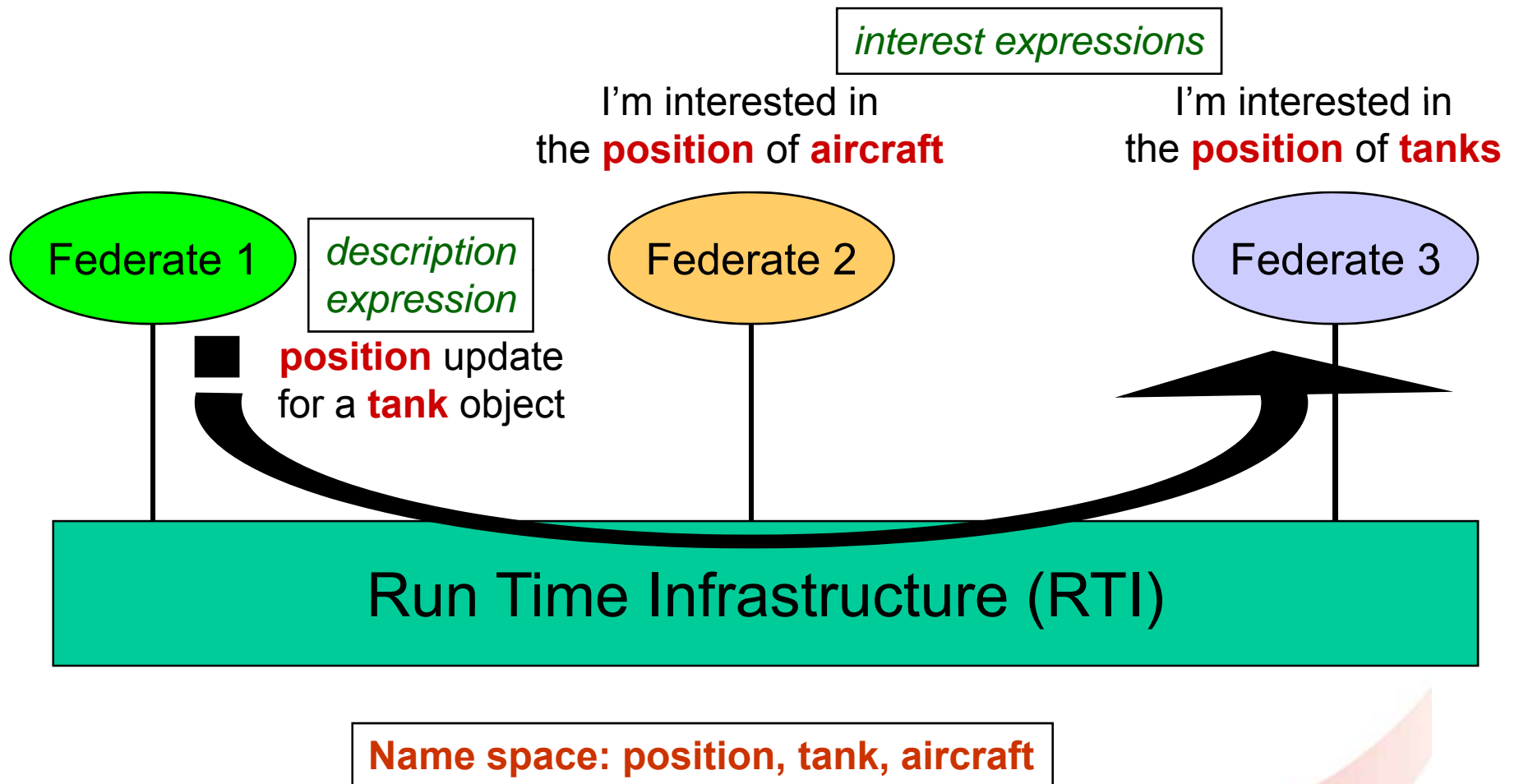
Concepts of Data Distribution

- ◆ Example: Moving vehicles in a virtual environment
 - Moving vehicle sends “update” messages indicating new position
 - Each vehicle that can “see” the moving vehicle should receive a message
 - How does the sender/RTI know which other federates should receive the message?
- ◆ A mechanism is required to reduce:
 - The message traffic over the network
 - The data to be processed by the receiving federate

Goal of Data Distribution

- ◆ **Goal:** route data produced by one federate to those federates that are interested in receiving the data (and ideally, not to federates that are not interested in receiving the data)
- ◆ This implies there is:
 - Some way for a federate to specify what data it is interested in receiving (**interest expressions**)
 - Some way to describe the data that is produced (**description expressions**)
 - A common language (vocabulary) to specify description and interest expressions (**name space**)

Example



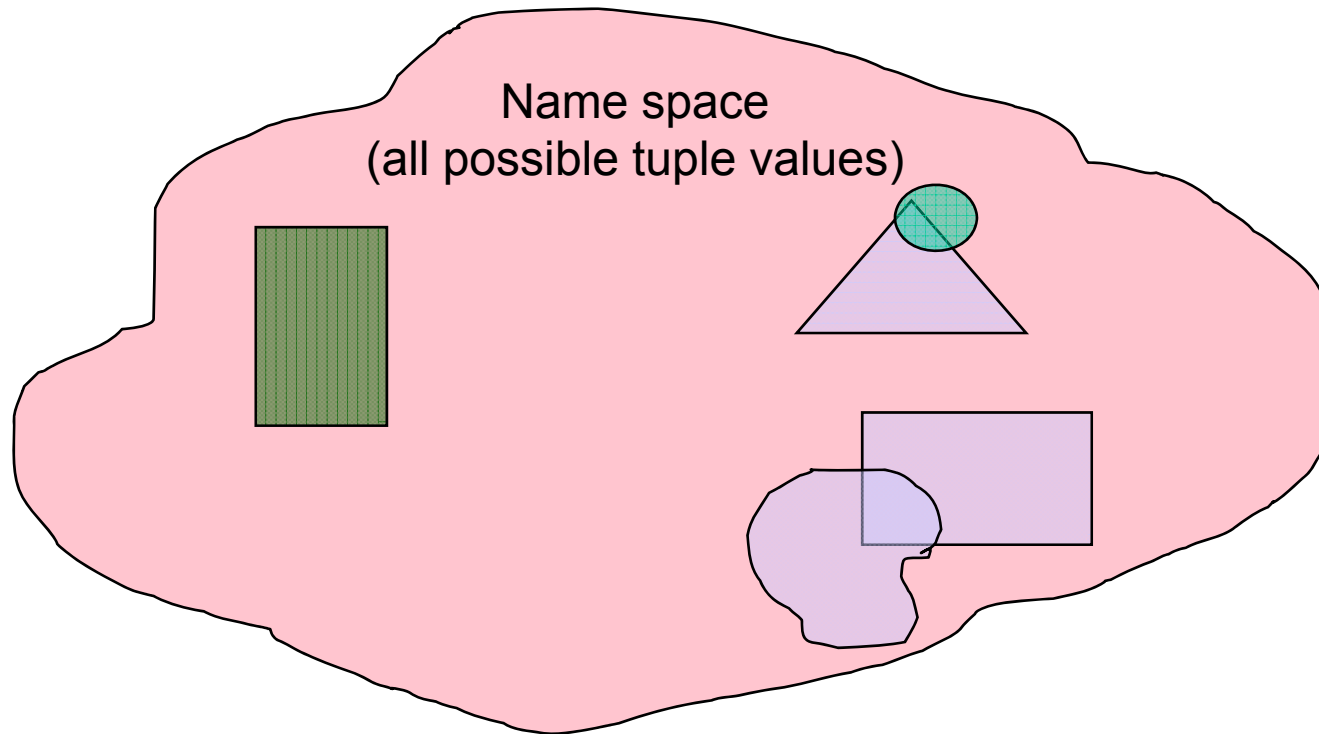
Vocabulary and Name Space




- ◆ Vocabulary used to create:
 - Interest expressions
 - Description expressions
 - May include **static** properties of objects (e.g., class names, attribute names) or **dynamic** properties (attribute values)
- ◆ Name space is a set of tuples (V_1, V_2, \dots, V_N) where V_i is a basic type or another tuple
 - **Example: (class, location)**
 - ◆ Class: enumerated type <tank, aircraft, ship>
 - ◆ Location: tuple (int: X-coordinate, int: Y-coordinate), where $0 < X < 1000$ and $0 < Y < 1000$
 - Values in name space: (tank,(30,200)); (aircraft,(10,20))

Interest and Description Expressions

- ◆ **Interest Expression:** subset of name space
 - Interested in all aircraft
 - ◆ (aircraft, (X, Y)) for any X and any Y
 - Interested in tanks that are “close by”
 - ◆ (tank, (X, Y)) where $10 < X < 20$, and $130 < Y < 150$
- ◆ **Description Expression:** subset of the name space
 - (tank, (15, 135))
 - (aircraft, (X, Y)) where $35 < X < 38$ and $98 < Y < 100$
- ◆ **Data routing**
 - A federate receives a message if the message's description expression overlaps with the federate's interest expressions

Interest and Description Expressions



-  Interest expressions, Federate 1
-  Interest expressions, Federate 2
-  Description expression for a message

The message is routed to Federate 2, but not to Federate 1

Static vs. Dynamic Data Distribution

◆ Static Data Distribution

- Name space only includes static properties that do not change during the execution
- Example: **Class-based** data distribution
 - ◆ Filter based on class types
 - ◆ “Give me updates to the position attribute of all objects of class tank”
- Cannot filter based on dynamically computed quantities

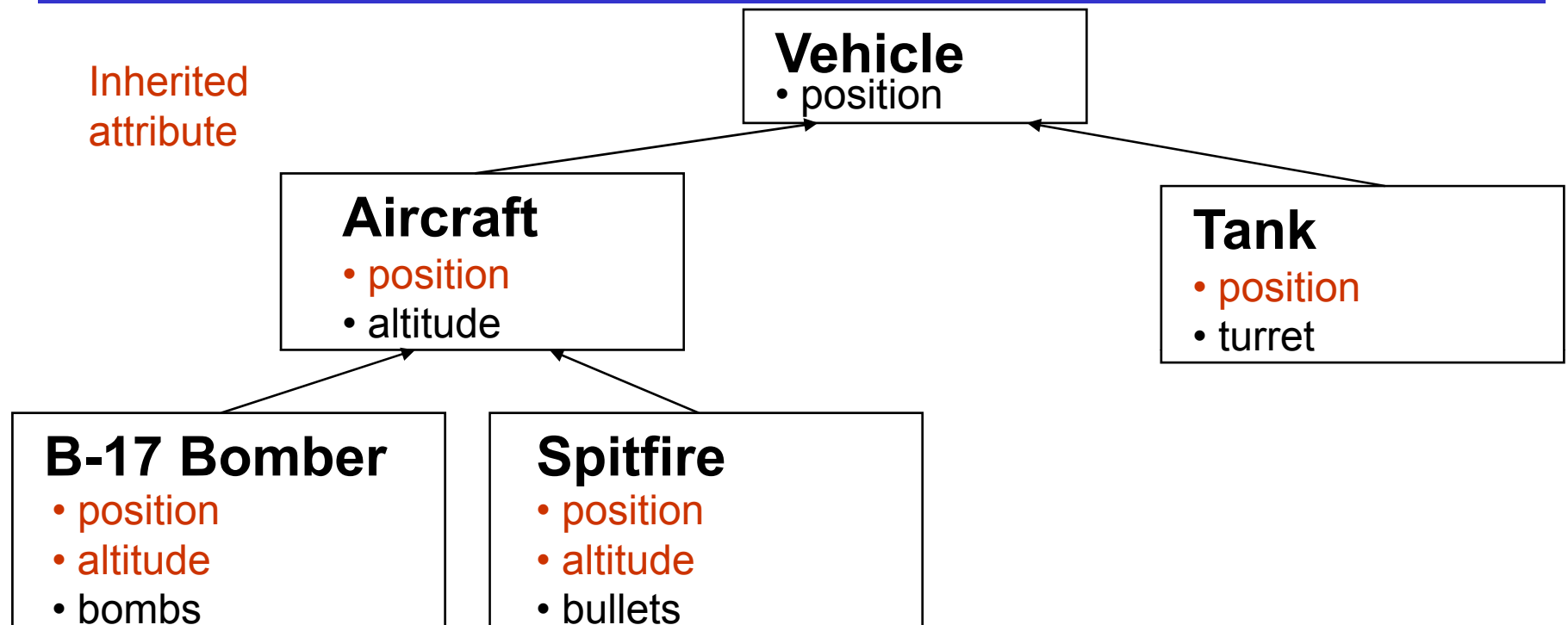
◆ Dynamic Data Distribution

- Name space includes dynamic quantities that may change during the execution
- Example: **Value-based** data distribution
 - ◆ “Give me updates to tank objects that are close to me”

Class-Based Data Distribution

- ◆ Declaration Management services in the HLA
- ◆ Federation Object Model (FOM) defines an interaction and object class hierarchy describing all data exchanged among federates
 - **Interaction Classes and Parameters**
 - **Object Classes and Attributes**
- ◆ New subclasses can be added without requiring modification to subscribers at higher levels in the class hierarchy
- ◆ Description expressions and interest expressions specify points in the interaction or object class hierarchy

Class Hierarchy Example

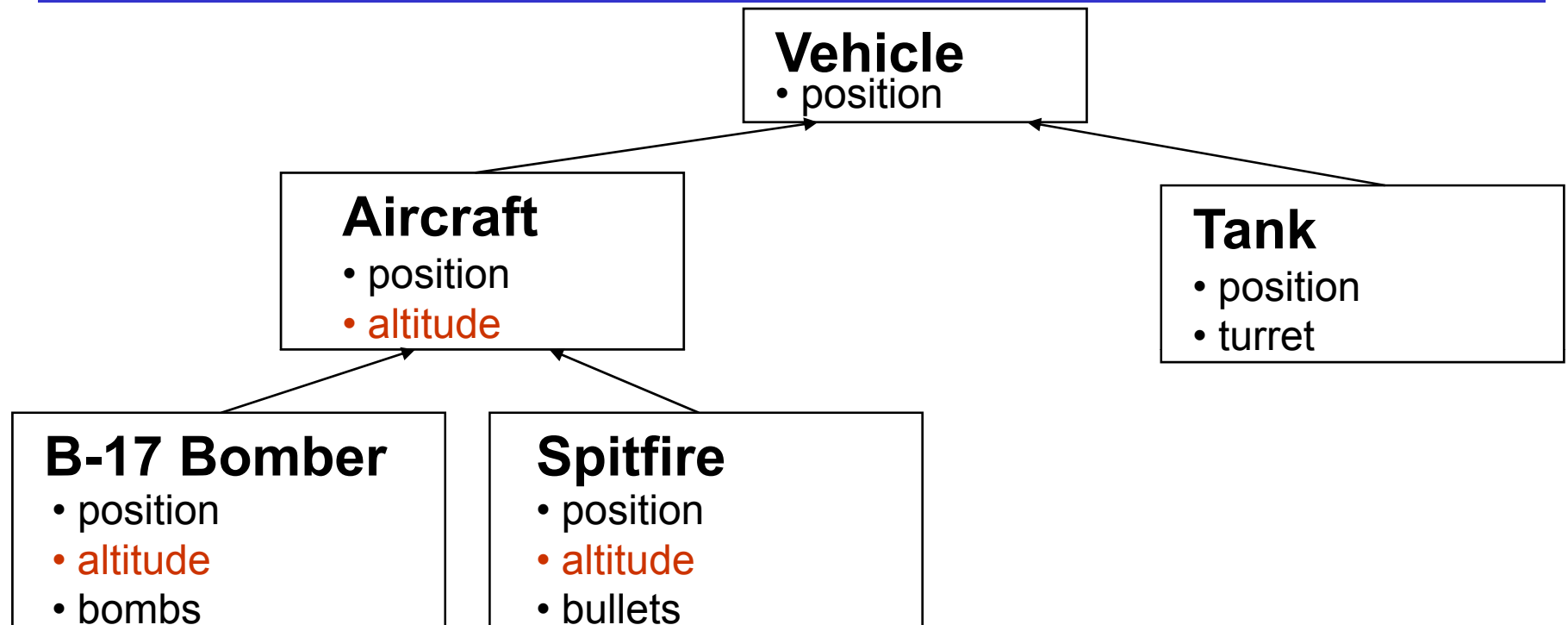


◆ Each class inherits attributes from parent class

◆ Name space: <class, attribute>

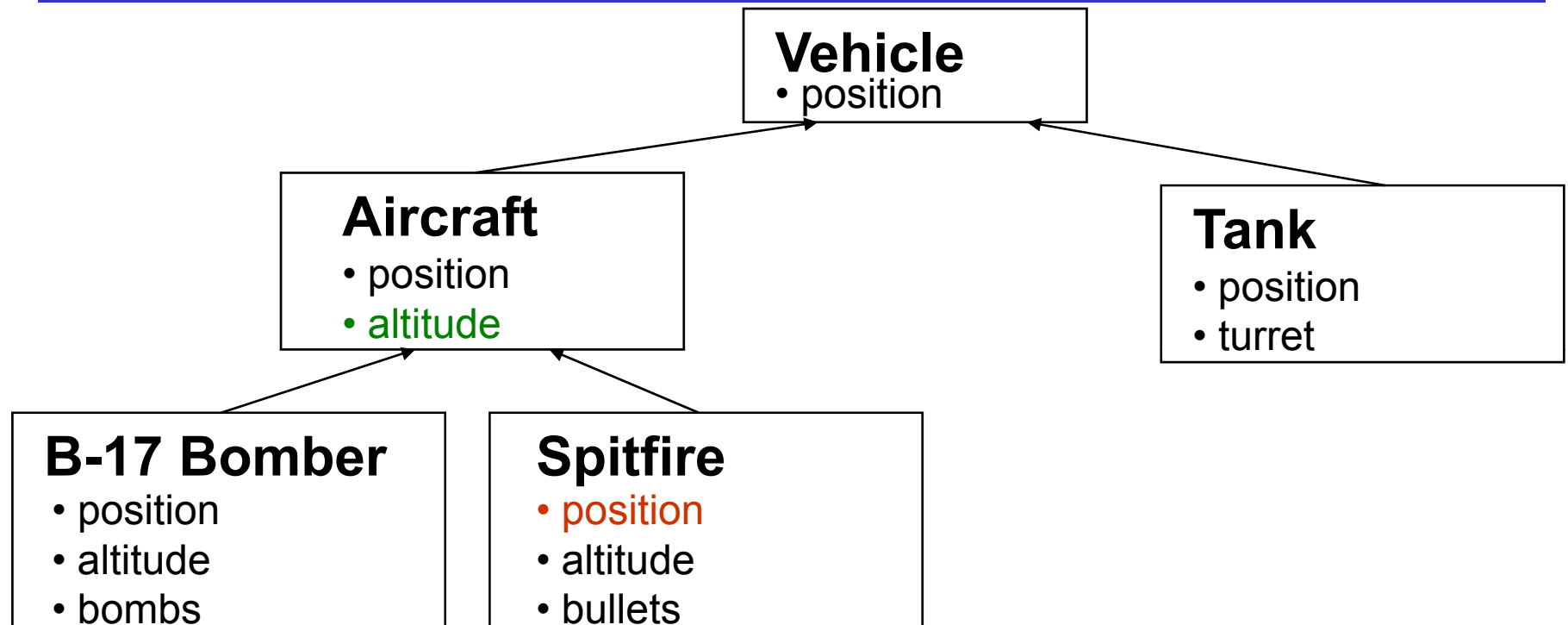
<Vehicle,position>, <Aircraft,position>, <Aircraft,altitude>, <Tank,position>,
<Tank,turret>, <B-17,position>, <B-17,altitude>, <B17,bombs>, <Spitfire,position>,
<Spitfire,altitude>, <Spitfire, bullets>

Interest Expressions



- ◆ Subscribe Object Class Attributes [class, attribute(s)]
- ◆ Interest Expression: Sub-tree rooted at subscription point
Subscribe (Aircraft, altitude): receive updates to altitude attribute of Aircraft, B-17, Spitfire objects
<Aircraft, altitude>, <B-17 Bomber, altitude>, <Spitfire, altitude>
In all cases, message appears as an update to an aircraft object

Description Expressions



- ◆ Update Attribute Values service sends a message
- ◆ Description Expression: an attribute of an object instance
Single <class attribute> point in the name space
Examples: <Spitfire, position> or <Aircraft, altitude>

Class Based vs. Value Based Data Distribution

◆ Class Based – Declaration Management (DM)

- A federate subscribing to attribute values of a class will get **all** updates of these values for all objects of that class currently existing in the federation
- A subscribing federate may stop delivery of attributes for classes of objects in which it has no interest
- DM is well suited for supporting small federations
- DM may not produce a sufficient reduction in traffic for federations with:
 - ◆ a large number of interaction and/or object classes
 - ◆ a large number of object instances per object class
 - ◆ frequent interactions and/or attribute updates

Class Based vs. Value Based Data Distribution

◆ Value Based – Data Distribution Management (DDM)

- Large federations require more refined filtering to improve their performance and scalability
- DDM provides detailed control over producer/consumer relationships
- DDM services allow a federate to receive the subscribed attributes **selectively** based on values of characteristics of the publishing federate

◆ DM vs. DDM

- DM describes the producer/consumer relationships in terms of interaction classes and object classes
- DDM allows the relationships to be redefined for individual interactions and instance attribute values

Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
- ◆ Data Distribution Management
 - **Concepts of Data Distribution**
 - **Data Distribution Management (DDM) and Services**
 - **Using Interactions with DDM**
 - **Using Objects with DDM**

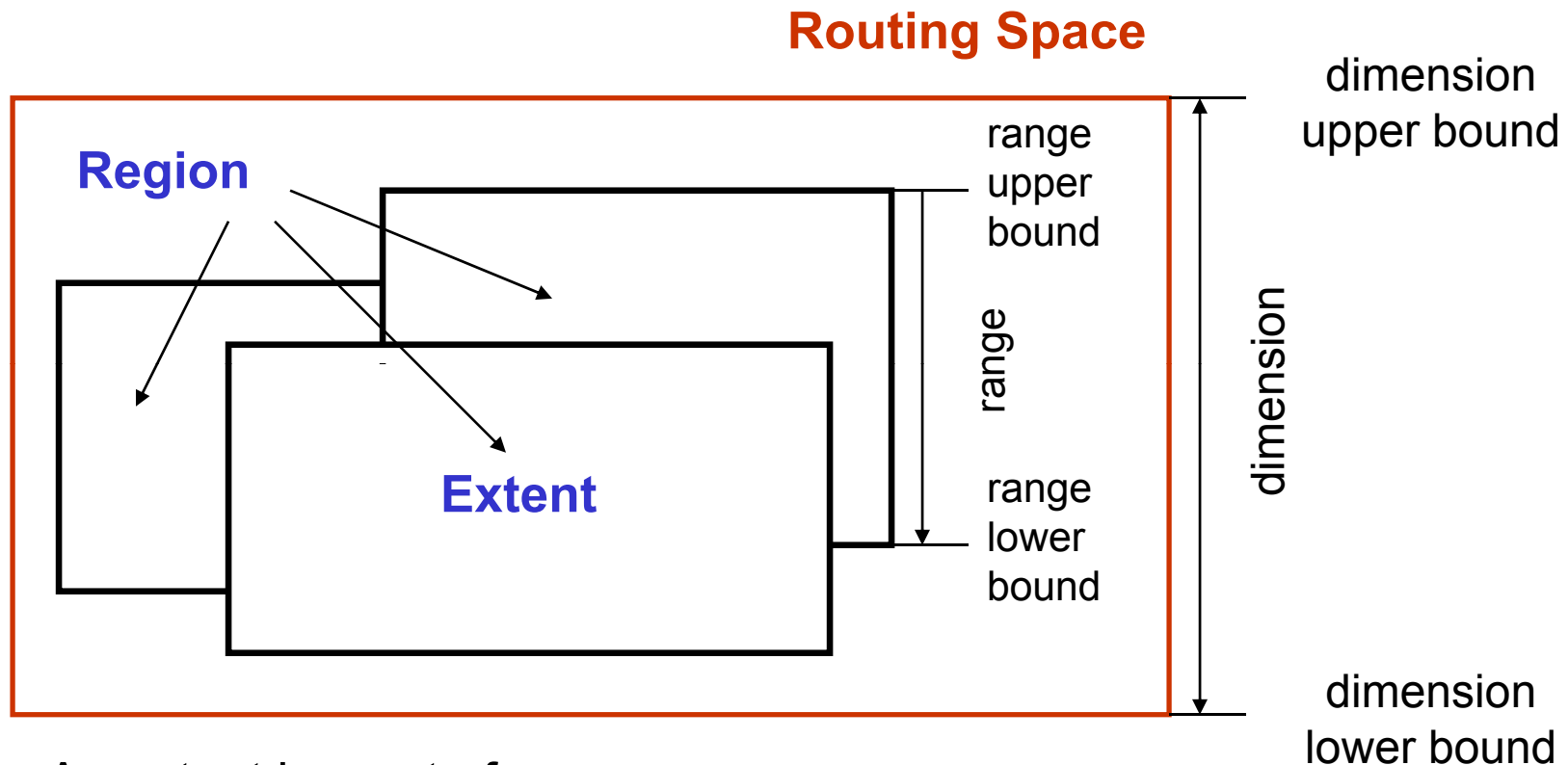
Data Distribution Management

- ◆ Name Space
 - **Routing Space:** N-dimensional coordinate system
 - **Separate from simulation state, used solely for routing**
- ◆ Interest Expressions
 - **Subscription Region:** N-dimension rectangle in routing space
 - **Associate region with subscription requests**
- ◆ Description Expressions
 - **Update Region:** N-dimensional rectangle in routing space
 - **Associate region with an **interaction** or an **object instance****
- ◆ A message is routed to a federate if:
 - **The federate is subscribed to the class/attribute, **and****
 - **The update region associated with the interaction or updated attribute overlaps with the federate's subscription region for that class/attribute**

Routing Spaces

- ◆ A **Routing Space** is a multi-dimensional coordinate system in which federates express an interest or intent for either receiving or sending data
- ◆ To use routing spaces, a federation must define the allowable routing spaces for its execution
- ◆ Routing spaces are specified in the FOM/FED with a name, number of dimensions and additional parameters
- ◆ The multi-dimensional routing space is subsetting to capture interest in receiving or sending data via **Subscription** and **Update Regions**

Routing Space and Regions



An extent is a set of ranges

A region is a set of extents (often simply one)

A region is a subset of the routing space

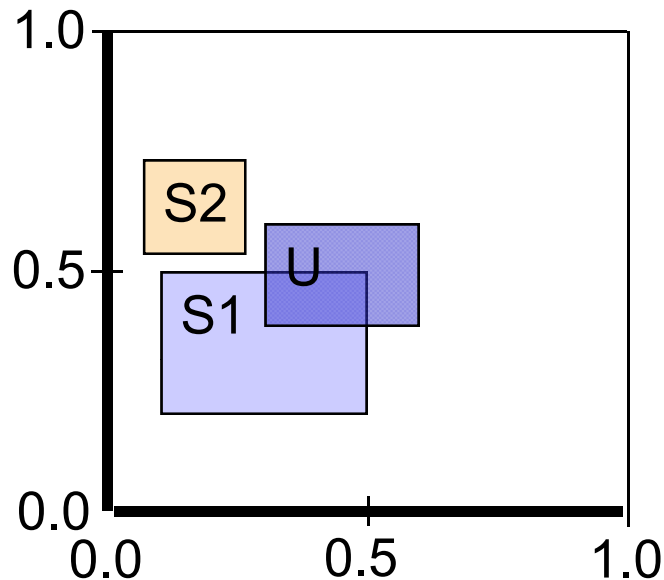
Subscription and Update Regions

- ◆ A **Subscription Region** is *associated* with a **subscribing federate**
 - The region describes the interests of a subscribing federate
 - The region may need to be modified as the interests of the federate change
- ◆ An **Update Region** is *associated* with **interactions and object instances** at the publishing federate
 - The region describes the data that is produced
 - The federate must guarantee that the correct region is associated with the interactions or instance attribute updates
 - The region may need to be modified to maintain this guarantee

Distributing Data

- ◆ The RTI uses the routing spaces and regions for distributing data, based on the **overlap** of subscription and update regions of different federates
 - **The RTI ensures that interactions and attribute updates associated with the update region are routed only to the federates whose subscription regions overlap the sender's update region**
 - **The subscribing federates each receive only the class attributes and interactions to which they subscribe**
- ◆ The overlap calculation establishes connectivity between producers and consumers and is performed once only (unless the region is modified), not for each attribute update or interaction

Data Distribution Management



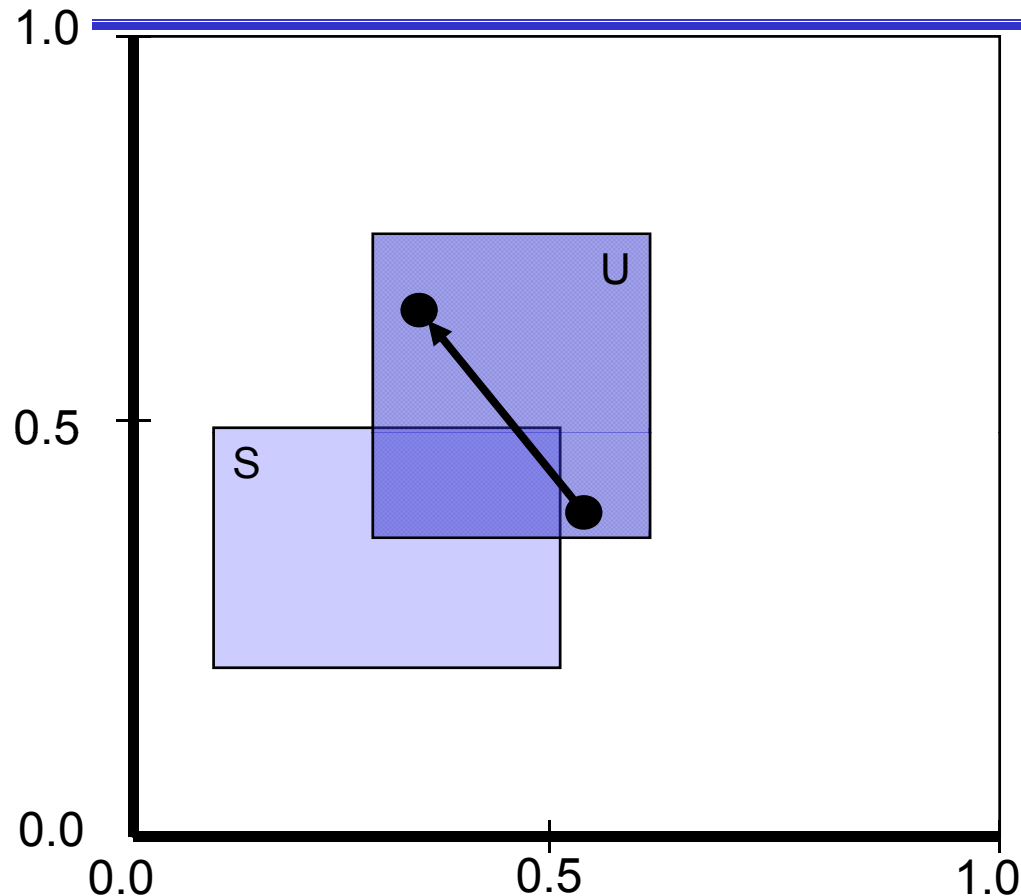
- Federate 1 (sensor): subscribe to S1
- Federate 2 (sensor): subscribe to S2
- Federate 3 (target): update region U

Update messages by target are sent to Federate 1, but not to federate 2

Description Expressions

- Update region in routing space (U)
- Associated an update region with each attribute update
- a federate receives a message if
 - It has subscribed to the attribute(s) being updated, and
 - Its subscription region overlaps with the update region

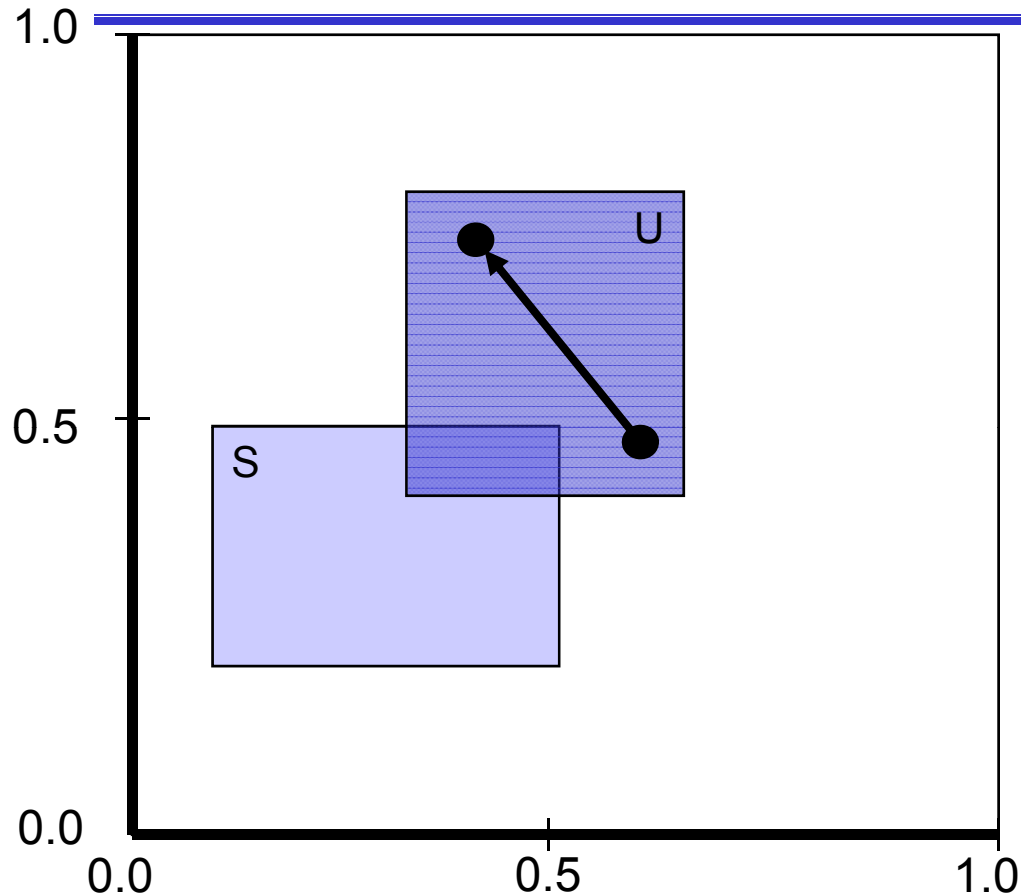
Update Regions vs. Points



- Routing space represents playbox
- Subscription region represents sensor
- Updates correspond to position of a moving vehicle

- ◆ Update points: Sensor not notified of vehicle
- ◆ Update regions: Sensor is notified of vehicle

Filtering Precision



- Vehicle out of range, but updates are still routed to sensor federate
- Messages must be filtered at the receiver
- Sensor range may not be rectangular

In general, DDM is a compromise:

- ◆ Filtering accuracy
- ◆ Implementation considerations
- ◆ Ease of use

DDM and the FOM

- ◆ Federates must agree about the meaning and use of Routing Spaces and their dimensions
 - **Routing Space names**
 - **Number of dimensions and names**
 - **Dimension type, range and units**
 - **Normalization functions**
 - **Interactions and Attributes to be routed through the routing space**
- ◆ Routing Space information is recorded in the FOM Routing Space table
- ◆ Interaction and Attribute information is recorded in the FOM Parameter and Attribute Tables

Normalization Functions

- ◆ Federates agree on the range and interpretation of dimensions and normalize values to the RTI axis (this is the same for all dimensions, e.g. $[0, 2^{31}]$).
 - ◆ Common normalization functions used by federates are:
 - **Linear:** a linear mapping from domain values to the RTI axis
 - **Linear Enumerated:** a linear mapping from enumerated values in the domain to the RTI axis
 - **Enumerated Set:** a one-to-one mapping from a set of enumerated values in the domain to a set of values on the RTI axis
 - **Logarithmic:** a logarithmic mapping from domain values to the RTI axis
-

DDM and other HLA Services

- ◆ DDM Services work in cooperation with Declaration (DM) and Object Management (OM)
- ◆ DDM, DM & OM Services for Interactions include:
 - **Creating Subscription and Update Regions (DDM)**
 - **Publishing Interaction Classes (DM)**
 - **Subscribing Interaction Classes (DM)**
 - ◆ with Region (DDM)
 - **Sending Interactions (OM)**
 - ◆ with Region (DDM)
 - **Receiving Interactions (OM)**
 - **Modifying and Deleting Regions (DDM)**

DDM and other HLA Services

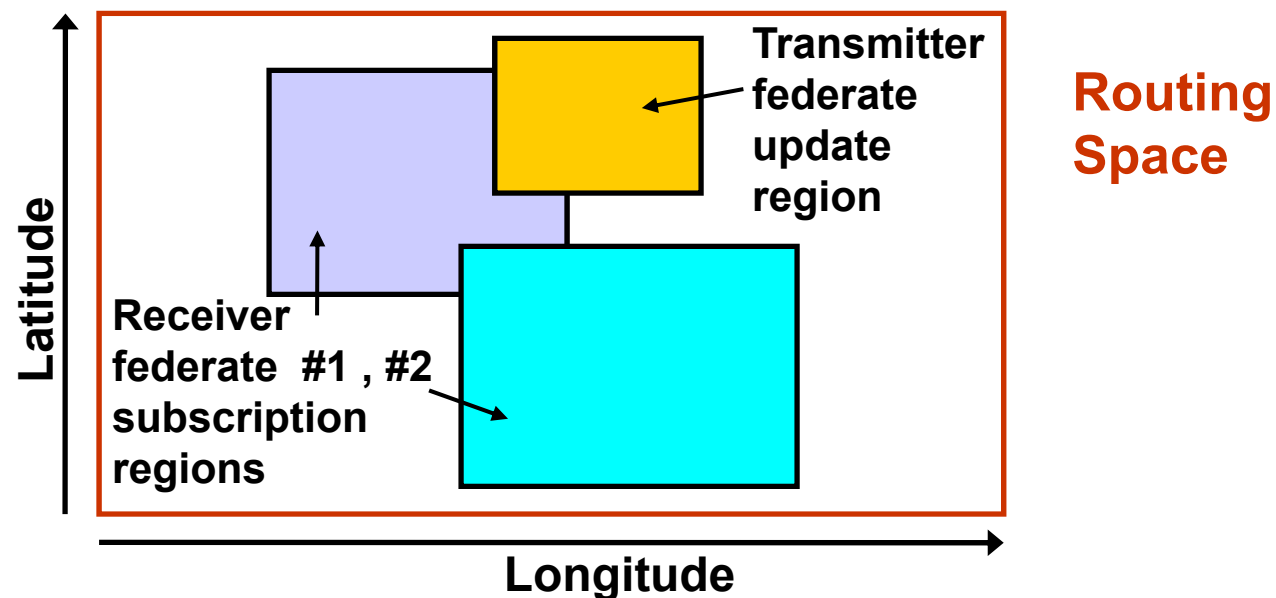
- ◆ DDM, DM & OM Services for Objects include:
 - **Creating Subscription and Update Regions (DDM)**
 - **Publishing Object Classes (DM)**
 - **Subscribing Object Classes (DM)**
 - ◆ with Region (DDM)
 - **Registering Object Instances (DM)**
 - ◆ with Region (DDM)
 - **Discovering Object Instances (OM)**
 - **Updating and Reflecting Attribute Values (OM)**
 - **Deleting & Removing Object Instances (OM)**
 - **Modifying and Deleting Regions (DDM)**

Outline – HLA And Distributed/Federated Simulation

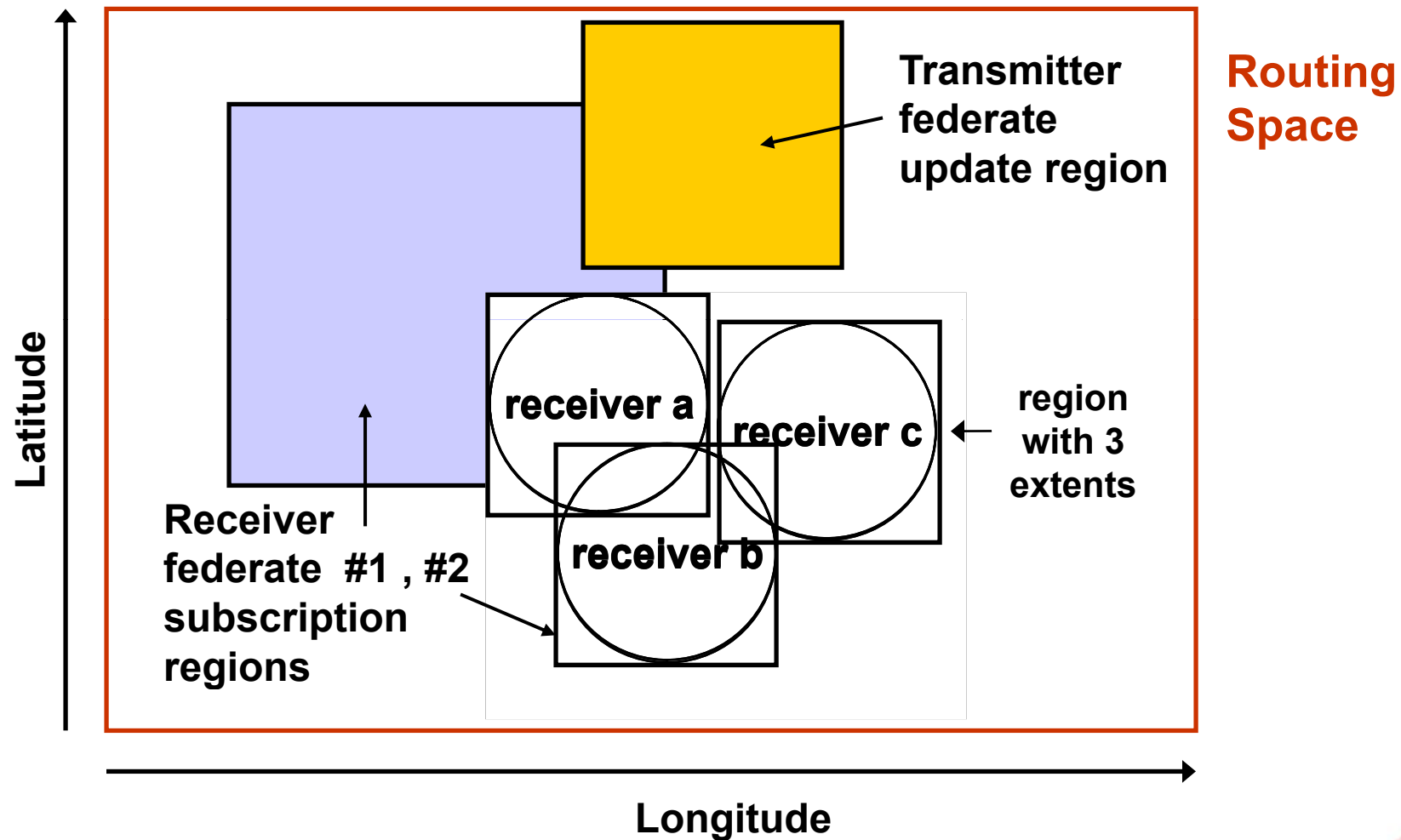
- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
- ◆ Data Distribution Management
 - **Concepts of Data Distribution**
 - **Data Distribution Management (DDM) and Services**
 - **Using Interactions with DDM**
 - **Using Objects with DDM**

Using Interactions with DDM: Radio Example

- ◆ Transmitter and Receiver federates have regions corresponding to geographical areas
- ◆ A Receiver federate can limit the messages received to those from Transmitters with an overlapping region



Routing Space and Regions: Radio Example



Normalization Function

◆ Normalization Function

- In general, a normalization function is needed to map from domain values to the RTI axis
- For this example, a linear normalization function can be used

$$f(v) = (((v - \text{MIN_D}) / (\text{MAX_D} - \text{MIN_D})) \\ * (\text{MAX_EXTENT} - \text{MIN_EXTENT}) \\ + \text{MIN_EXTENT})$$

where

- ◆ **v** is the domain value to be normalized
- ◆ **MIN_D** and **MAX_D** are minimum and maximum domain values
- ◆ **MIN_EXTENT** and **MAX_EXTENT** are minimum and maximum values on the RTI axis (defined by RTI implementation)

FOM: Radio Example

Routing Space Table					
Routing Space	Dimension Name	Dimension Type	Dimension Range/Set	Range/Set Units	Normalization Function
RadioArea	Latitude	Float	[0,180]	degrees	Linear
	Longitude	Float	[0,180]	degrees	Linear

Parameter Table				
Interaction	Parameter	Data Type	...	Routing Space
Message	sender	Short		RadioArea
	content	String		

Obtaining Handles

Routing Space Table					
Routing Space	Dimension Name	Dimension Type	Dimension Range/Set	Range/Set Units	Normalization Function
RadioArea	Latitude	Float	[0,180]	degrees	Linear
	Longitude	Float	[0,180]	degrees	Linear

- ◆ Obtain Handles for routing space and dimension name(s) using **getRoutingSpaceHandle** and **getDimensionHandle**

```
radioArealId = rtiAmb->getRoutingSpaceHandle ( "RadioArea" );
```

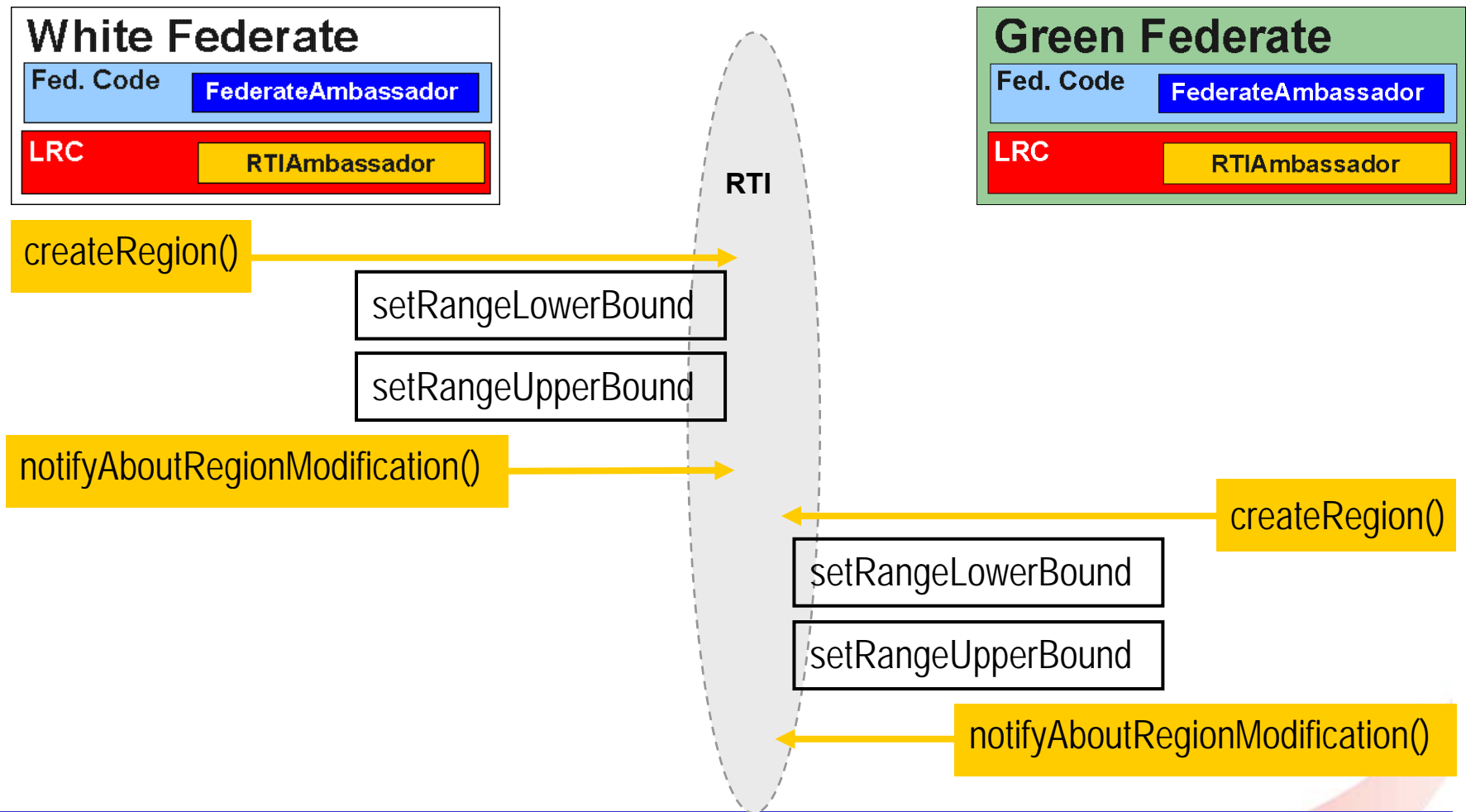
```
latitudeDimId =
```

```
    rtiAmb->getDimensionHandle ( "Latitude", radioArealId );
```

```
longitudeDimId =
```

```
    rtiAmb->getDimensionHandle ( "Longitude", radioArealId );
```


Creating Regions



Creating Regions

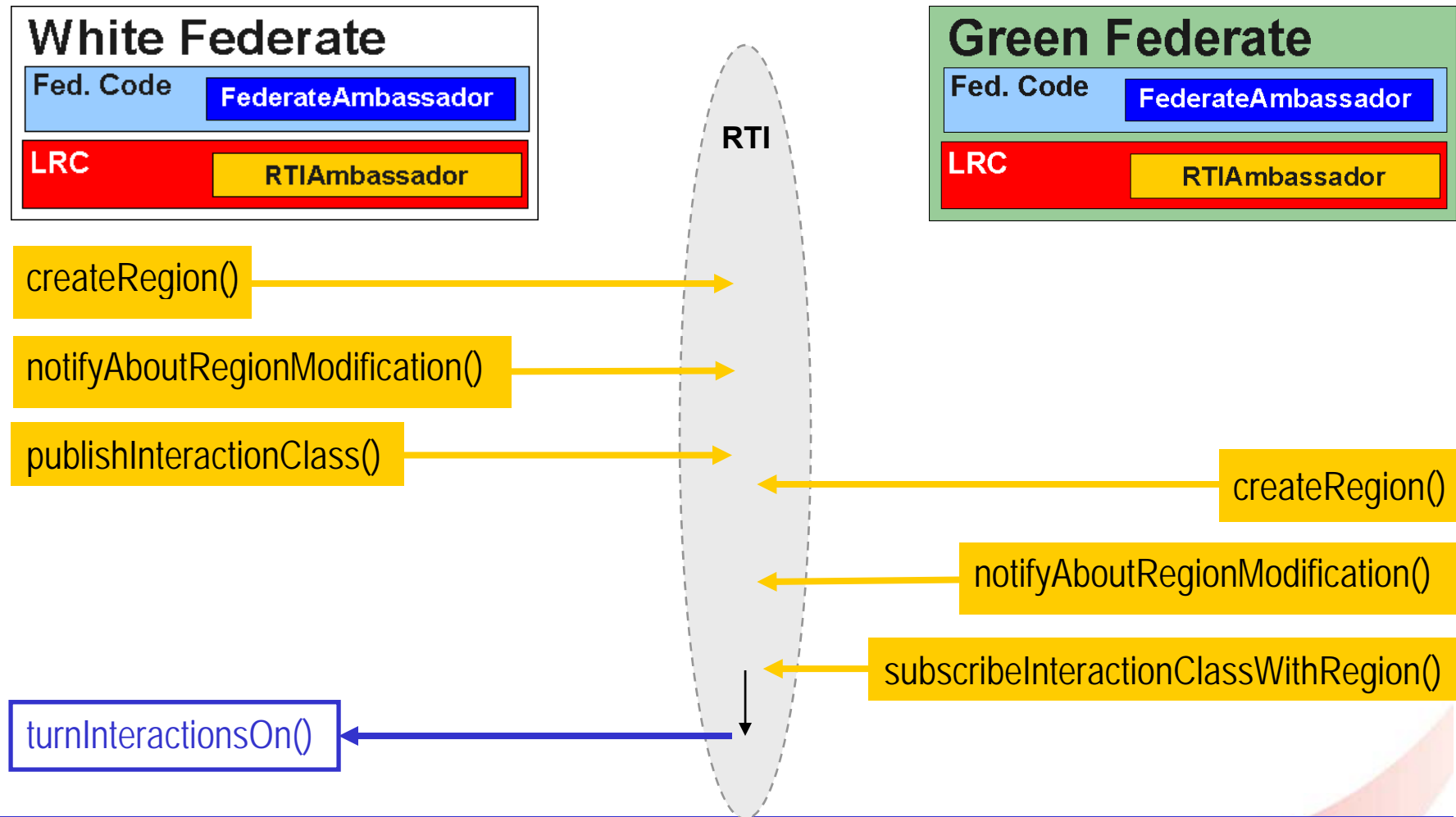
◆ createRegion

- **Arguments:** routing space handle, number of extents
`sRegion = rtiAmb->createRegion(radioAreald, 1);`
- **Returns a region object with functions to set range [Lower Bound, Upper Bound)**
- **Suppose the region has range [39.6N, 39.8N) latitude and [104.8W, 105.3W) longitude:**
`sRegion->setRangeLowerBound(0, latitudeDimId, f(39.6));`
`sRegion->setRangeUpperBound(0, latitudeDimId, f(39.8));`
`sRegion->setRangeLowerBound(0, longitudeDimId, f(104.8));`
`sRegion->setRangeUpperBound(0, longitudeDimId, f(105.3));`
- **The RTI must then be told that the range has been set:**
`rtiAmb->notifyAboutRegionModification (*sRegion);`

Creating Regions

- ◆ Subscription regions and update regions are created in exactly the same way
- ◆ Use of the region determines whether it is a subscription region or an update region
 - **A subscription region describes the interests of a subscribing federate**
 - **An update region describes the data that is produced**
- ◆ A federate can use the same region as both a subscription and an update region
- ◆ Federates can create multiple subscription and update regions

Using Interactions with DDM

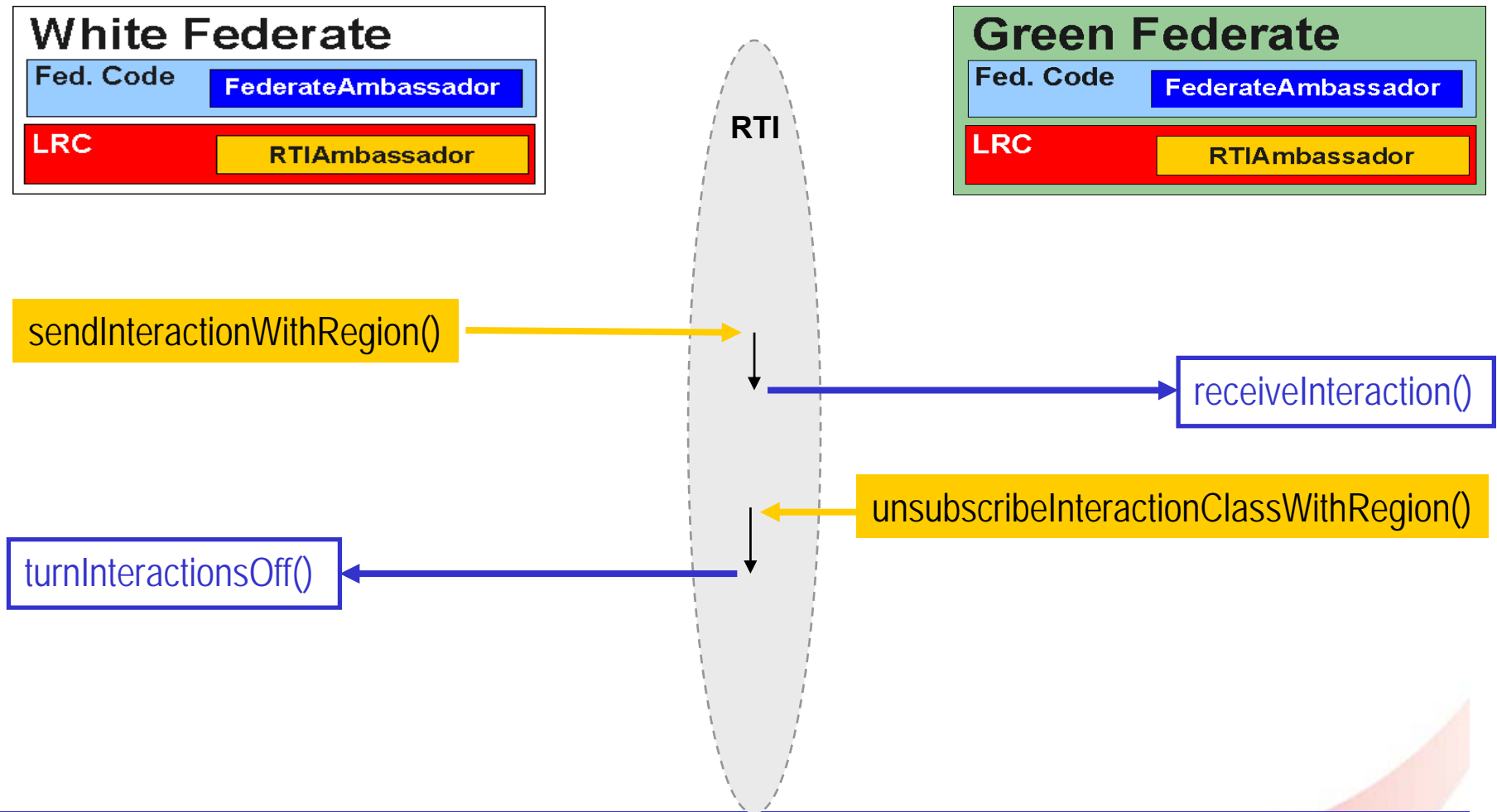


Subscribing Interactions with Regions

◆ subscribeInteractionClassWithRegion

- **Argument: interaction class handle, region, active flag**
`rtiAmb->subscribeInteractionClassWithRegion (`
`messageId, *sRegion);`
- **Using a subscription region means messages will be received only if they are sent with an update region which overlaps the subscription region**
- **Other arguments are exactly the same as the version without DDM**
- **Note that the region is associated with all parameters of an interaction class**
- **A newly subscribed region is added to the set of regions already subscribed for the interaction class**

Using Interactions with DDM



Sending Interactions with Regions

◆ **sendInteractionWithRegion**

- **Arguments:** **interaction class handle, parameters, timestamp, user tag, region**
`rtiAmb->sendInteractionWithRegion (messageld,
*messParams, timeStamp, tag, *uRegion);`
- **Other arguments are exactly the same as the version without DDM**
- **When an interaction is sent with regions, the RTI determines which federates should receive the message based on the **overlap** of update and subscription regions**
- **Messages are delivered by the RTI to the federate using the standard callback **receiveInteraction** †**

Outline – HLA And Distributed/Federated Simulation

- ◆ Historic Prospective
- ◆ HLA, RTI, and Federate
 - **What is HLA?**
 - **Federate vs. RTI**
- ◆ HLA/RTI Management Areas
- ◆ Time Management
- ◆ Data Distribution Management
 - **Concepts of Data Distribution**
 - **Data Distribution Management (DDM) and Services**
 - **Using Interactions with DDM**
 - **Using Objects with DDM**

Using Objects with DDM: HelloWorld

◆ HelloWorld is a **time-stepped** federate

- It simulates a country that periodically updates its population

$$P_{new} = P_{old} + k \times P_{old} \times \mathit{delt}$$

where

P_{old} is the population at time t

P_{new} is the population at time $t + \mathit{delt}$

delt is the time-step

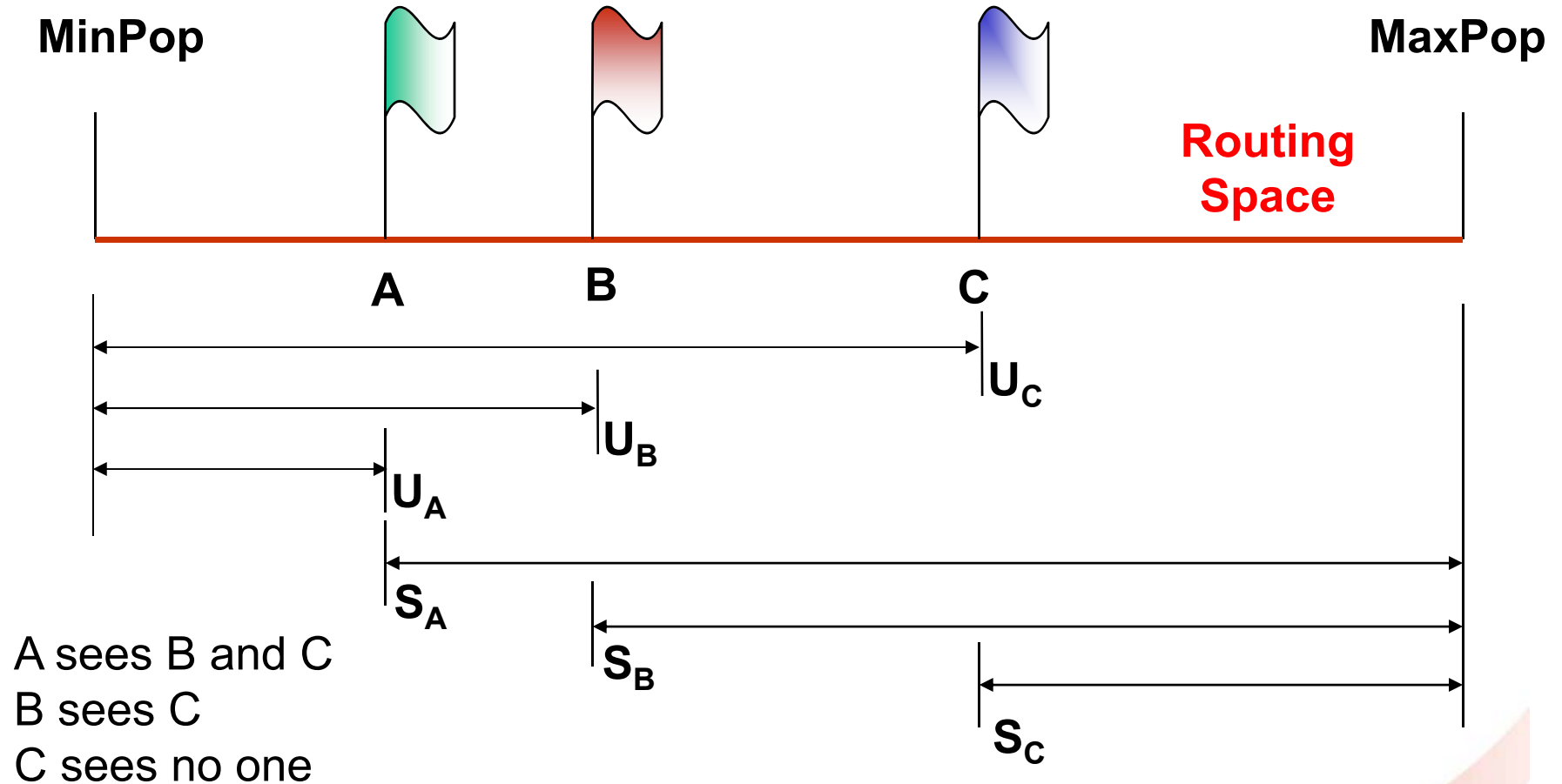
k is the growth rate

- It uses **updateAttributeValues()** to update its population to other federates

Using Objects with DDM: HelloWorld

- ◆ Suppose each country keeps a sorted table of population sizes
 - A country is only interested in receiving population attribute updates from other countries with larger populations in order to determine its own rank within the table
 - The routing space is called “PopulationRS” and is one dimensional
 - The dimension name is “populationSize” and it ranges from MIN_pop to MAX_pop
 - A linear normalization function is used as before
$$f(\text{pop}) = (((\text{pop} - \text{MIN_pop}) / (\text{MAX_pop} - \text{MIN_pop})) * (\text{MAX_EXTENT} - \text{MIN_EXTENT}) + \text{MIN_EXTENT})$$

Routing Space and Regions: Hello World



FOM: Hello World

Routing Space Table					
Routing Space	Dimension Name	Dimension Type	Dimension Range/Set	Range/Set Units	Normalization Function
PopulationRS	PopSize	Short	[MinPop, MaxPop]	N/A	Linear

Attribute Table				
Object	Attribute	Data Type	...	Routing Space
Country	Name	String		N/A
	Population	Short		PopulationRS

Obtaining Handles

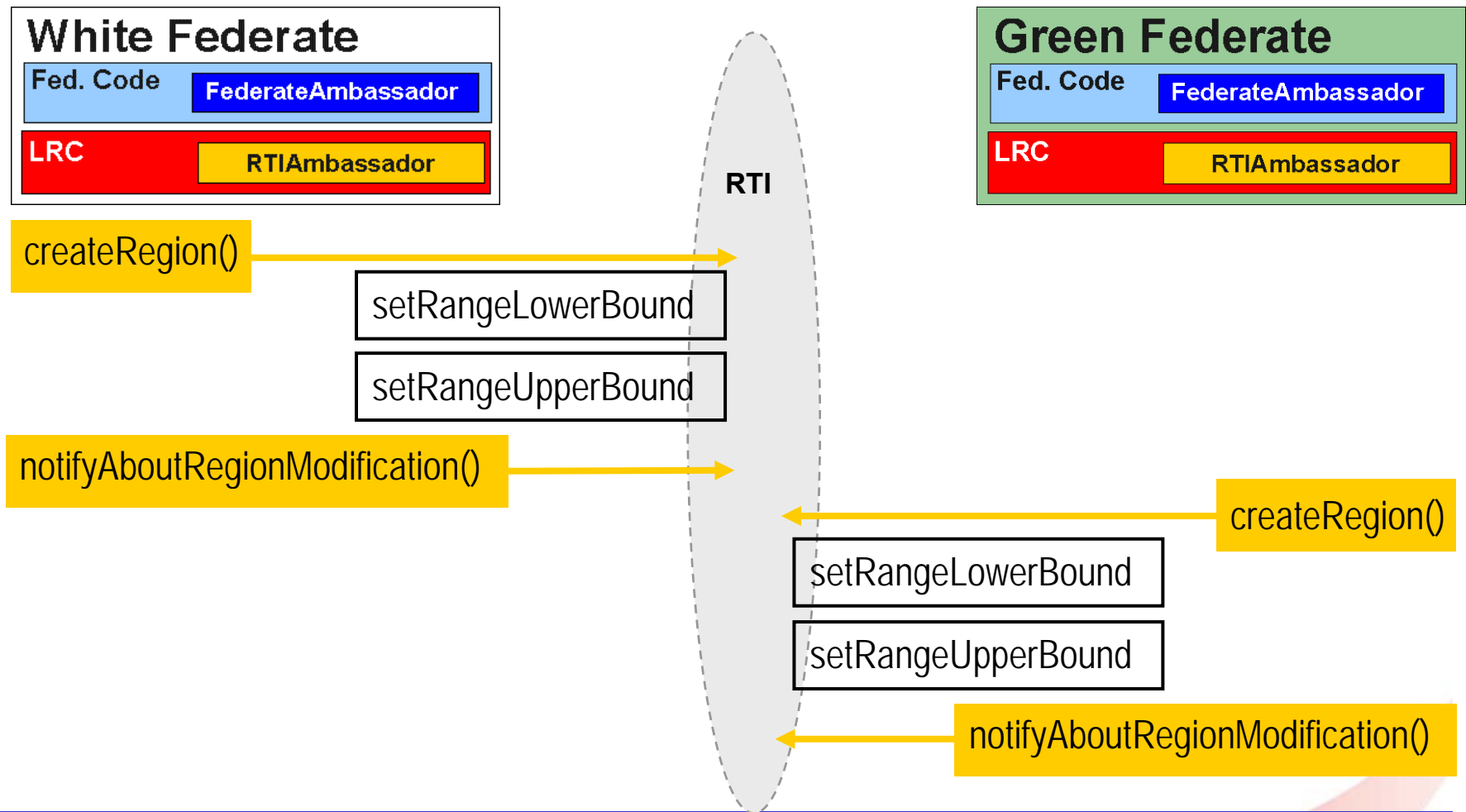
Routing Space Table					
Routing Space	Dimension Name	Dimension Type	Dimension Range/Set	Range/Set Units	Normalization Function
PopulationRS	PopSize	Short	[MinPop, MaxPop]	N/A	Linear

- ◆ Obtain handles for the routing space and dimension name using **getRoutingSpaceHandle** and **getDimensionHandle**

```
popRSId = rtiAmb->getRoutingSpaceHandle(  
    "PopulationRS" );
```

```
popSizeDimId =  
    rtiAmb->getDimensionHandle ( "PopSize", popRSId );
```

Creating Regions



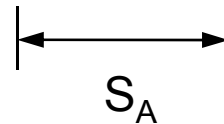
Creating Regions

◆ createRegion

- **Arguments:** routing space handle, number of extents

`sRegion = rtiAmb->createRegion (popRSId, 1);`

- **To create a region with range** $f(\text{myPop}, \text{maxPop})$



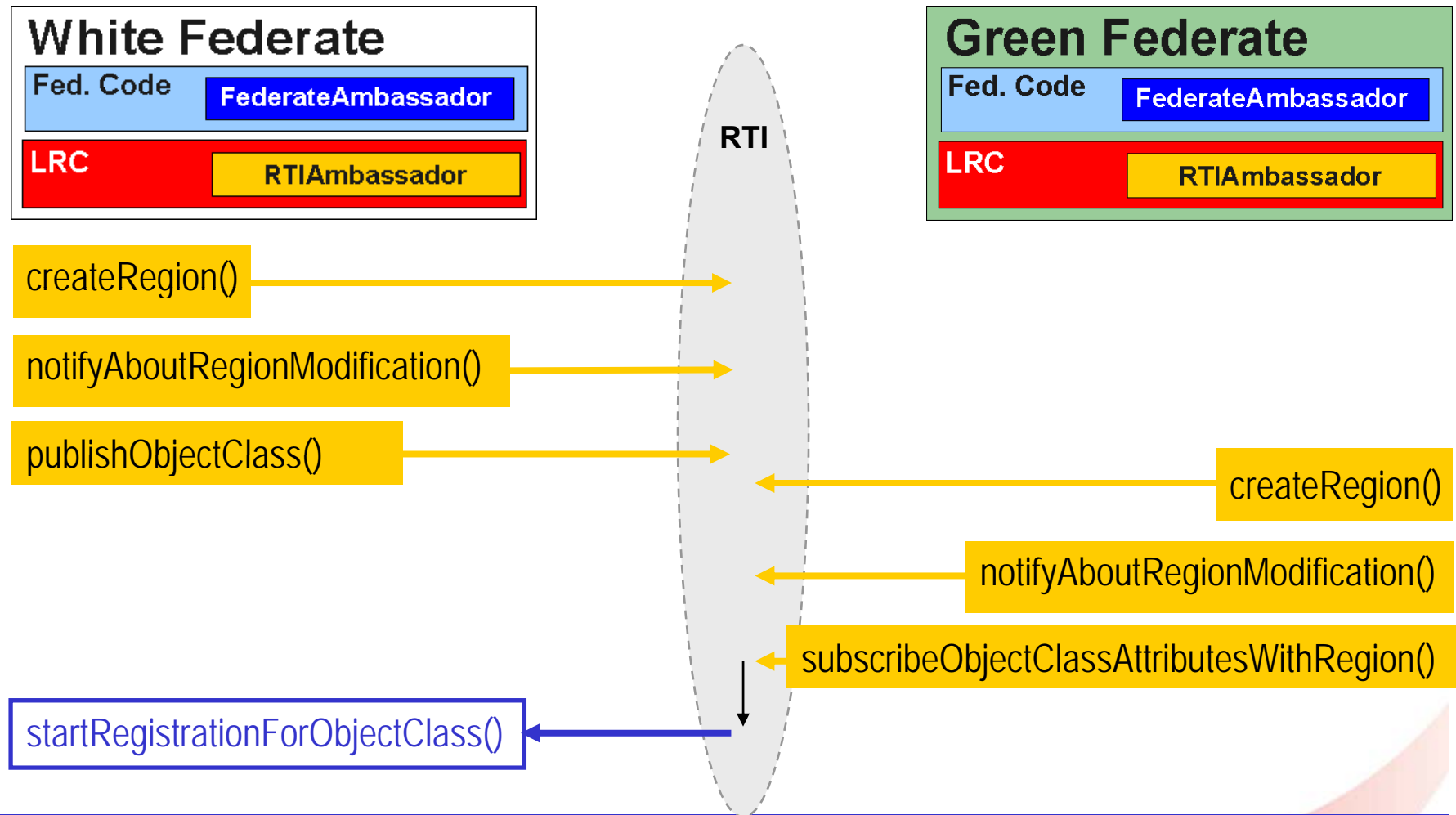
`sRegion->setRangeLowerBound(0, popSizeDimId, f(myPop));`

`sRegion->setRangeUpperBound(0, popSizeDimId, f(maxPop));`

- **The RTI must then be told that the range has been set:**

`rtiAmb->notifyAboutRegionModification (*sRegion);`

Using Object Classes with DDM



Subscribing Object Classes with Regions

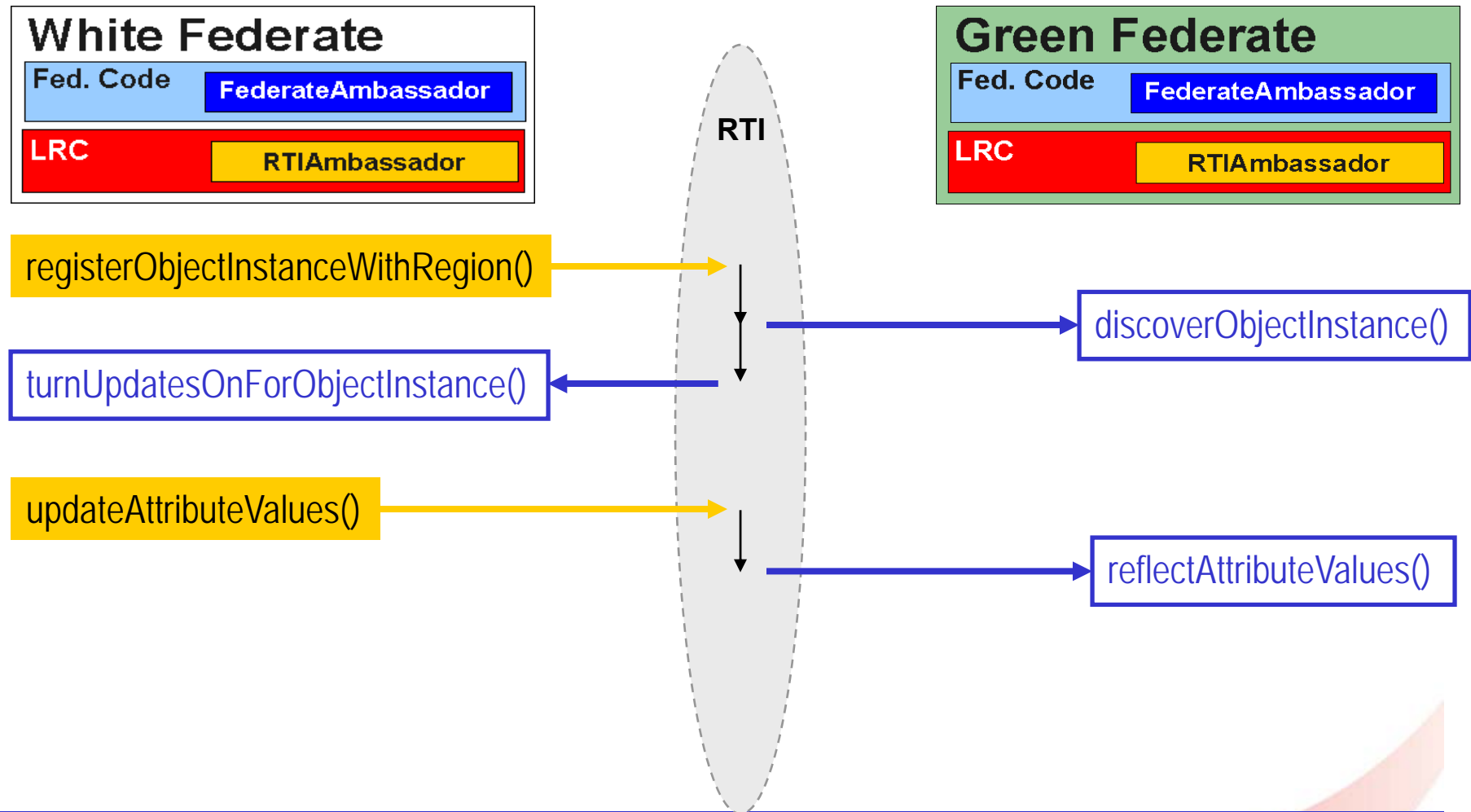
◆ subscribeObjectClassAttributesWithRegion

- **Arguments:** object class handle, region, attributes, active flag

```
rtiAmb->subscribeObjectClassAttributesWithRegion (  
                                countryTypeId, *sRegion, *attrSet);
```

- Using a subscription region means attribute updates will be received only if they are associated with an update region that overlaps the subscription region
- Different regions may be used for different attributes of the same object class and some attributes may be subscribed with no region
- Other arguments are exactly the same as the version without DDM
- A newly subscribed region is added to the set of regions already subscribed for the interaction class

Registering Object Instances with Regions



Registering Object Instance with Regions

◆ registerObjectInstanceWithRegion

- **Arguments:** object class handle, object instance name (optional), attribute list, region list, number of handles

AttributeHandle attrList[] = {popAttrId};

Region * regList[] = {uRegion};

instanceld =

rtiAmb->registerObjectInstanceWithRegion (
countryTypeld, attrList, regList, 1);

- The lists for attribute handles and regions must match one-to-one and must follow the attribute/routing space binding in the FOM and FED
- Other arguments are exactly the same as the version without DDM

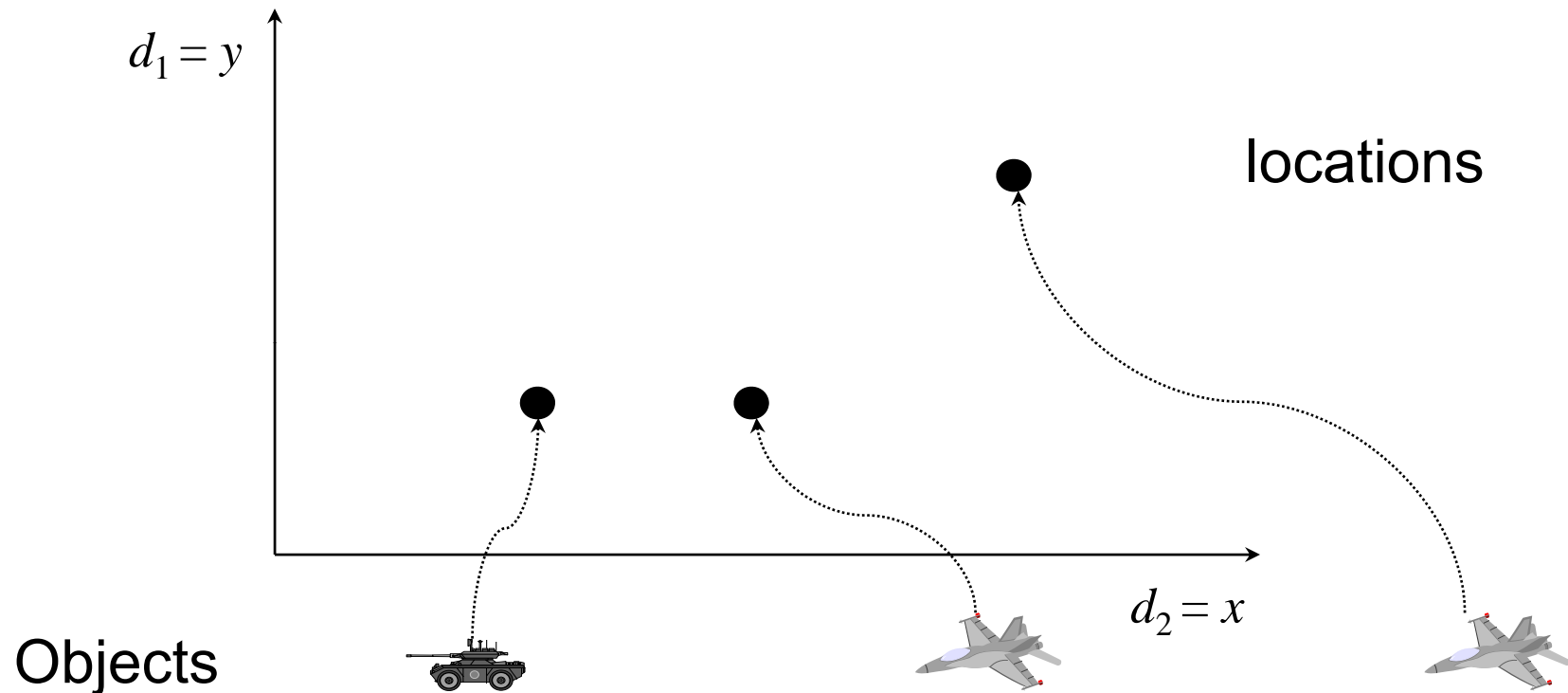
Registering Object Instances with Regions

- ◆ When an object instance is created using DDM, one or more update regions are associated with its relevant published attributes
- ◆ The RTI determines which objects should be discovered by which federates, based on the **overlap** of update and subscription regions
- ◆ Federates are notified by the RTI of objects that meet the federate's subscription requests using the standard callback **discoverObjectInstance** [†]
- ◆ Object discovery is provided only once by the RTI to a federate even when multiple locally defined subscription regions overlap an object's update region

Attribute Update and Reflection

- ◆ After registration, the federate may use the standard `updateAttributeValues` as required
- ◆ The attributes are reflected in each of the federates that have discovered the object using the standard callback `reflectAttributeValues` †
- ◆ These functions are used in exactly the same way as without DDM

DDM Processes

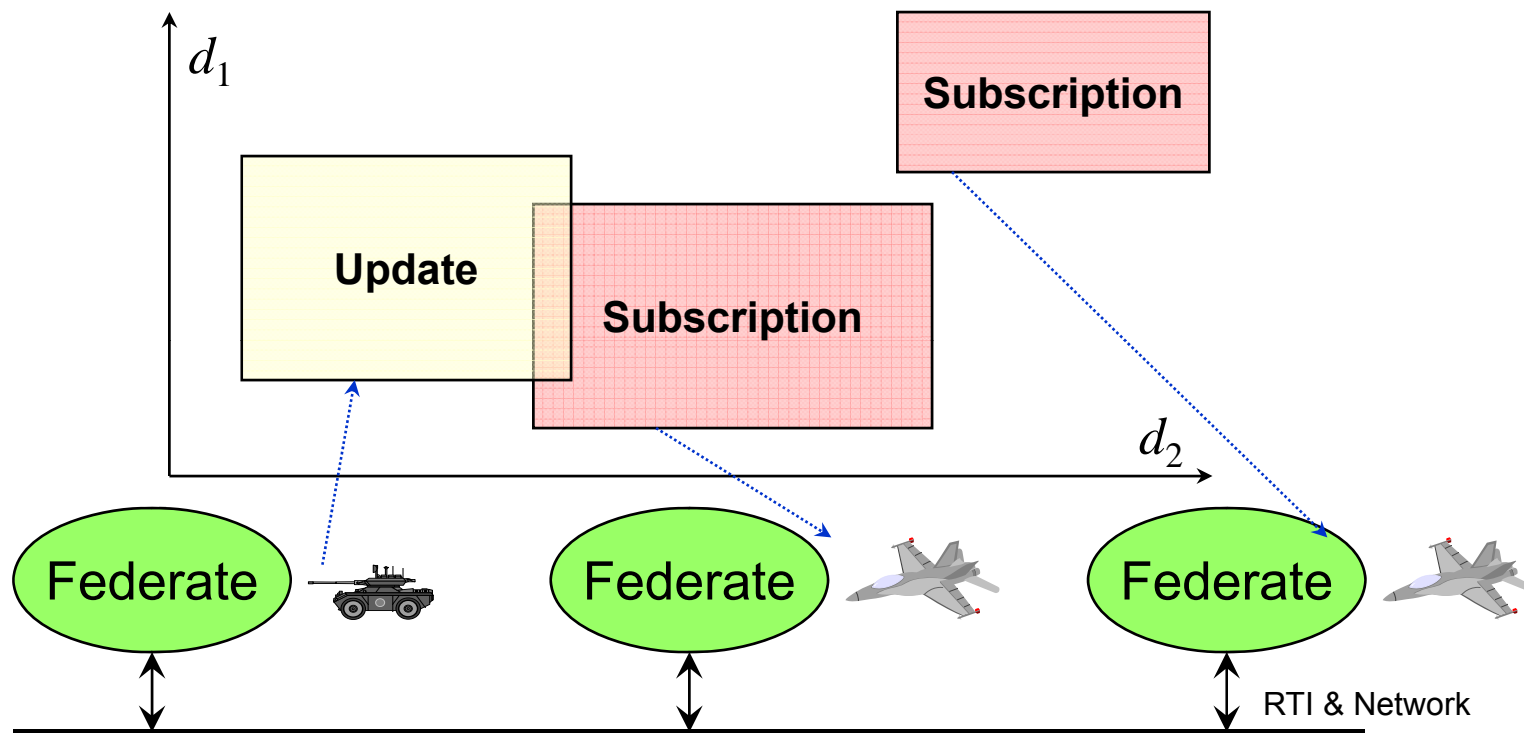


DDM coordinate system is based on a set of dimensions

Dimensions are usually associated with object attributes

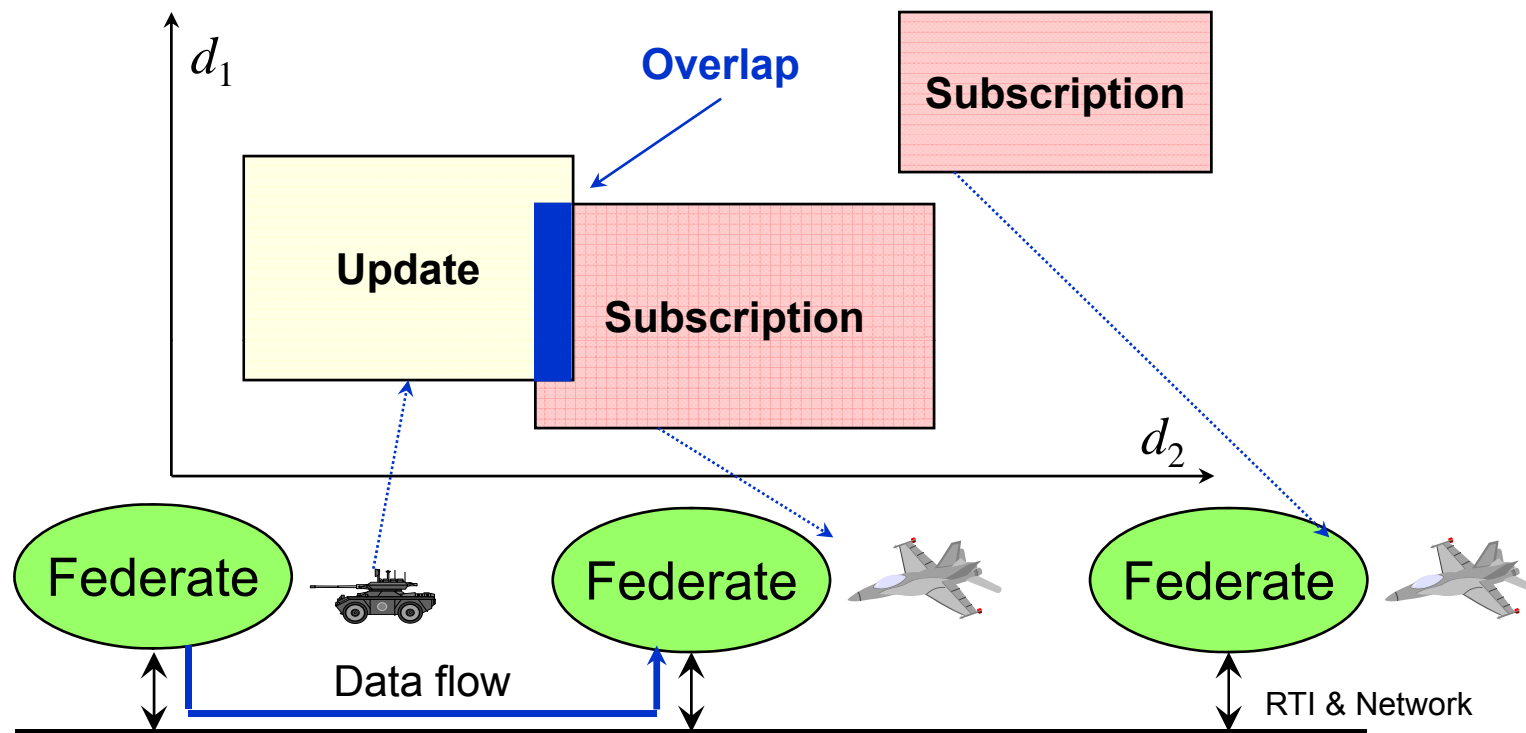
Dimensions and mapping functions for attributes defined at execution start

DDM Processes



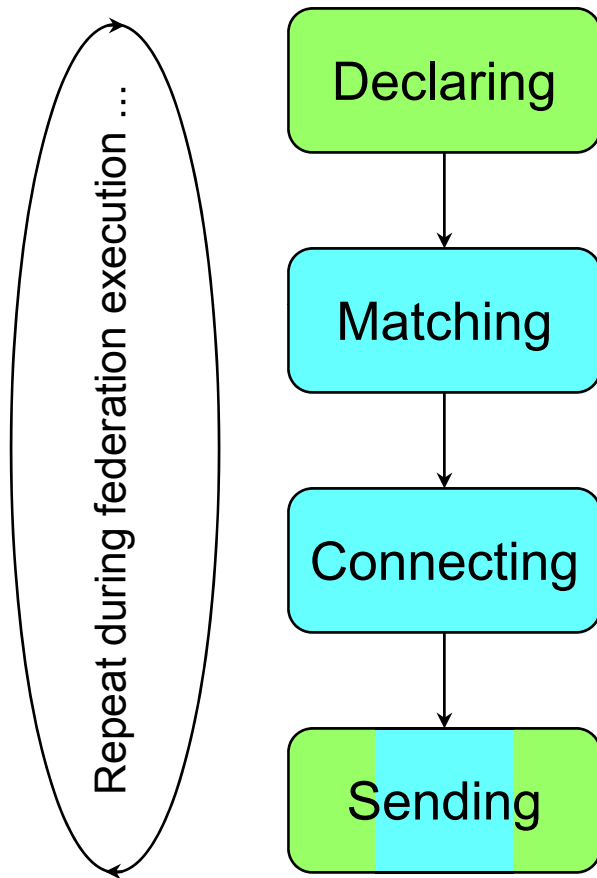
Federates define data production and interest as regions in coordinate system
Update region: data (attribute values) will correspond to region
Subscription region: federate wants data in region

DDM Processes



Update and subscription region overlap implies data flow between federates
RTI detects overlap, establishes connection, e.g. using multicast
Region configuration may change during execution

DDM Processes



Federates express the data they intend to produce (publish) or desire to receive (subscribe) using regions in a multi-dimensional coordinate system

RTI finds update regions that overlap subscription regions – for overlaps found, data must be sent from the publisher to the subscriber

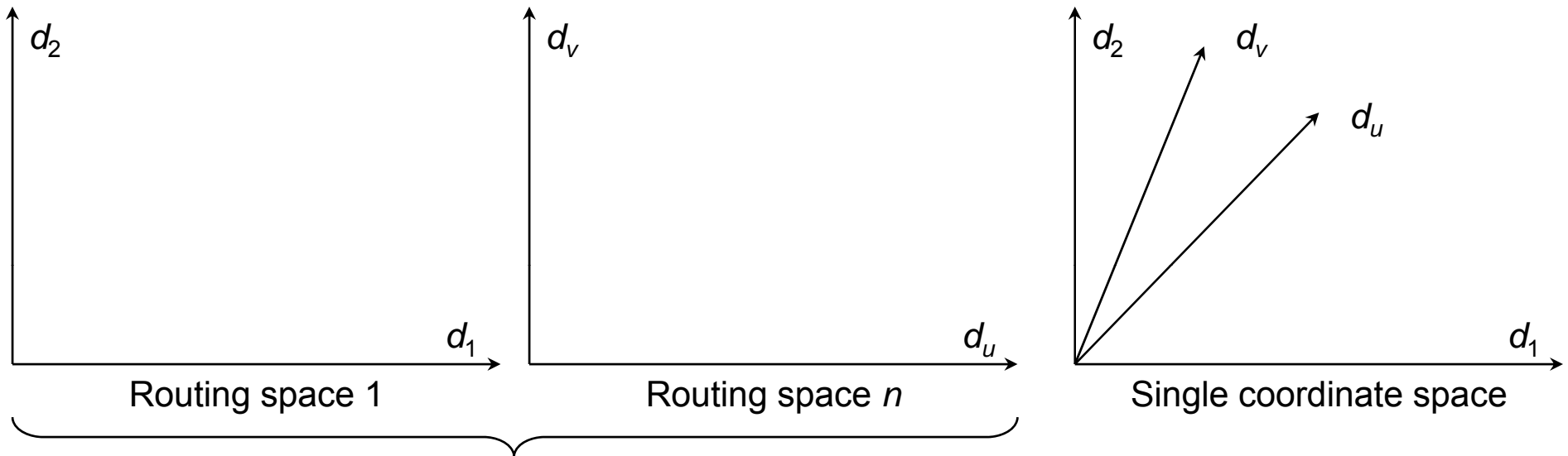
RTI establishes data flow connectivity between the sending and receiving federates – multicast is one method for doing this

Federates send data, which is transported by the **RTI** and the network infrastructure as per the connections, and received by **federates**

DDM 1.3 vs. DDM 1516

- ◆ Although the basic mechanism is the same, there are a number of important differences between DDM as defined in DMSO 1.3 Standard and DDM as defined in IEEE 1516 Standard
- ◆ Four main differences:
 - **Routing Space**
 - **Extent vs. Region**
 - **Region vs. Region Set**
 - **Overlap for Region/Region Set**
- ◆ These changes were introduced to simplify the standard

Routing Space



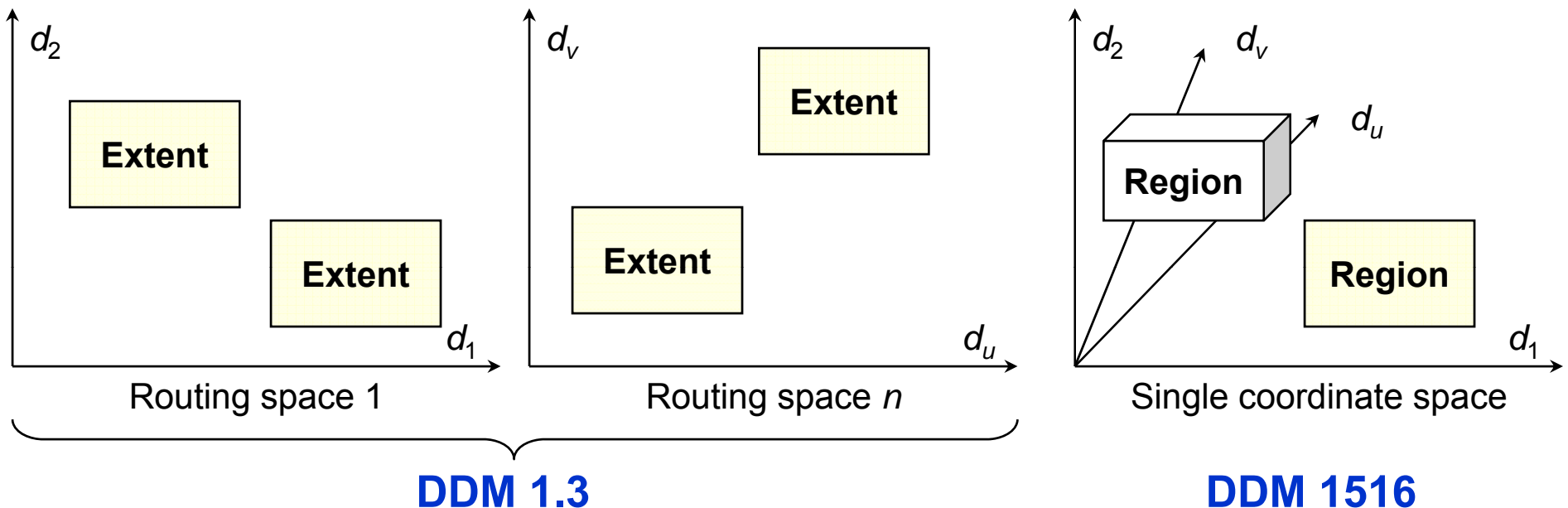
DDM 1.3

DDM 1516

Named subset of the available dimensions

- 1.3: *Routing space*; more than one may be defined
- 1516: No subset (i.e., no routing spaces); only a single coordinate space with all dimensions

Extent vs. Region

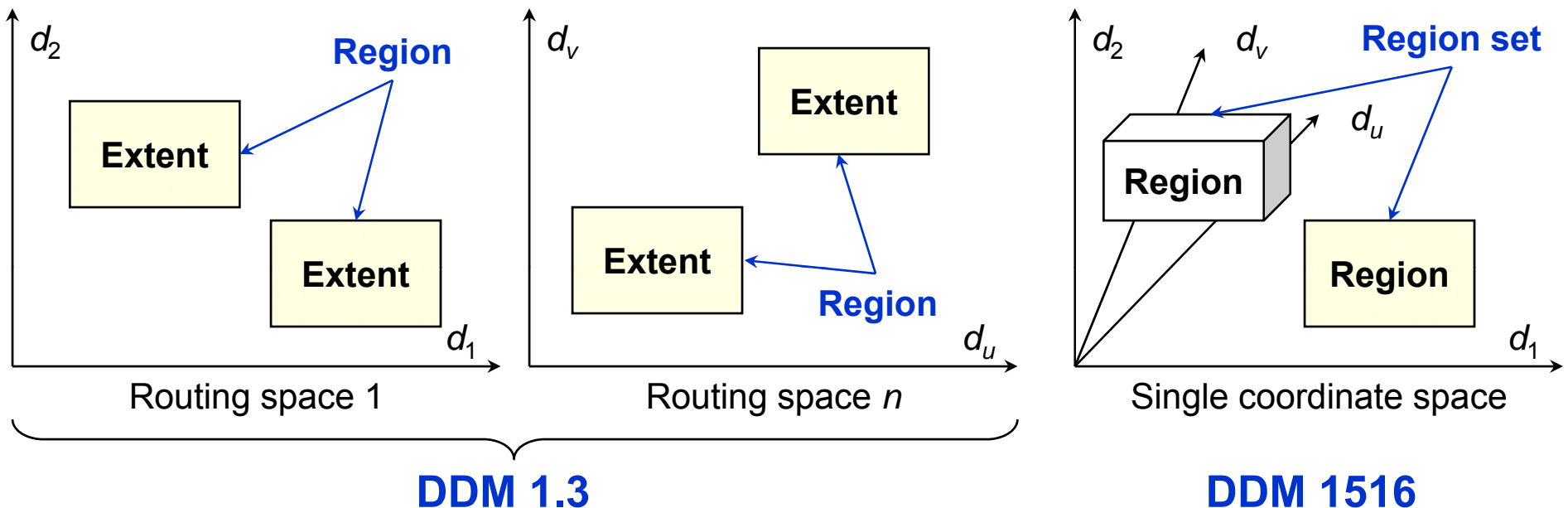


Rectangular subspace, defined by ranges on dimensions

1.3: *Extent*; has ranges for all dimensions in its routing space

1516: *Region*; has ranges on any subset of the dimensions

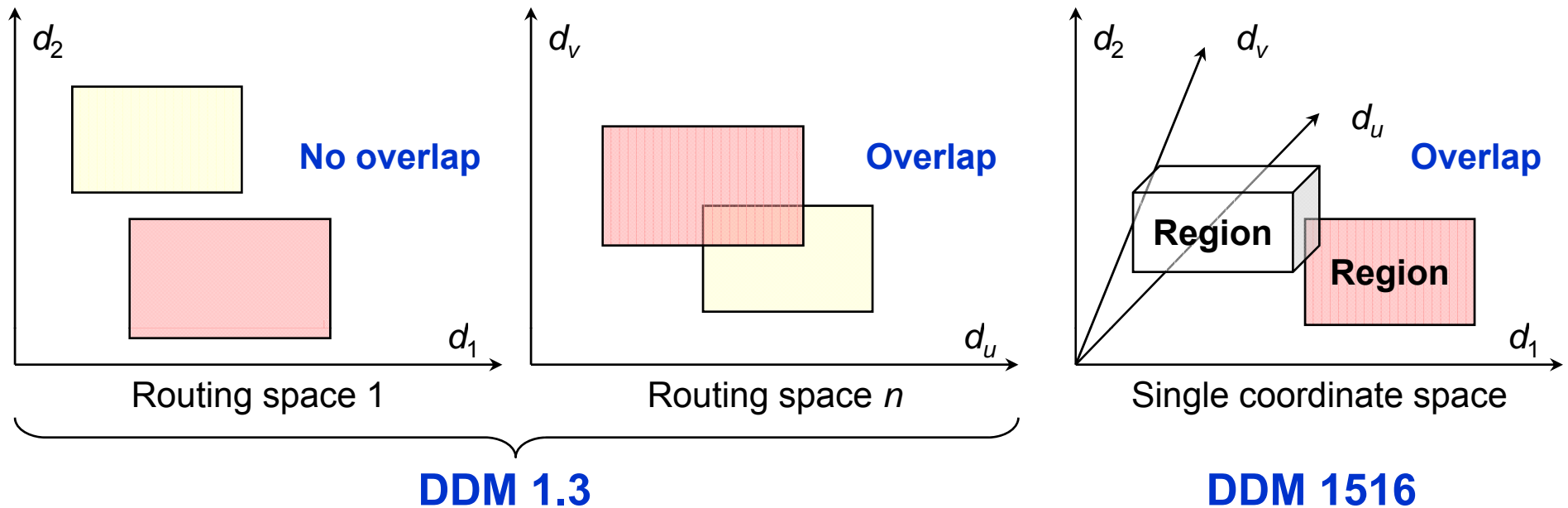
Region vs. Region Set



Associated collection of rectangular subspaces

- 1.3: *Region*; all extents in region must be in same routing space, thus will have the same dimensions
- 1516: *Region set*; regions in region set may have different dimensions

Overlap of Regions/Region Sets



Overlap of regions/region sets

- 1.3: Regions in same routing space (only) overlap iff two of their extents overlap; extents overlap iff their ranges overlap on all dimensions
- 1516: Region sets overlap iff two of their regions overlap; regions overlap iff they have at least one common dimension and they have overlapping ranges on all common dimensions

Summary

- ◆ The **HLA** is a flexible, reusable software architecture for creating component-based distributed simulations
- ◆ HLA provides a service-based framework for interoperating simulation models
- ◆ HLA applications: military trainings, acquisitions, distributed learning, gaming
- ◆ Terminology of HLA: federates, federation, RTI, FOM, object, interaction, attribute, parameter

Summary

- ◆ The HLA offers services in six areas, as defined by the **HLA Interface Specification**
 - ◆ **Federation Management** provides services for the creation and control of the federation execution
 - ◆ **Declaration Management** allows federates to specify the kind of data they will send and receive
 - ◆ **Object Management** provides the actual exchange of that data
 - ◆ **Ownership Management** allows federates to transfer ownership of object instance attributes
 - ◆ **Time Management** allows the proper ordering of events between federates
 - ◆ **Data Distribution Management (DDM)** provides detailed control over producer/consumer relationships
-

Summary

- ◆ HLA Time Management services designed to support interoperability of simulations with different time advance mechanisms
 - **Time stepped federates**
 - **Event driven federates**
- ◆ Time Management services include mechanisms to order messages (TSO delivery) and advance simulation time
- ◆ Time regulating/constrained used to “turn on” time management
- ◆ Each federate has its own lookahead value and different federates may have different lookahead values