Speculative High-Performance Simulation

Alessandro Pellegrini

A.Y. 2019/2020

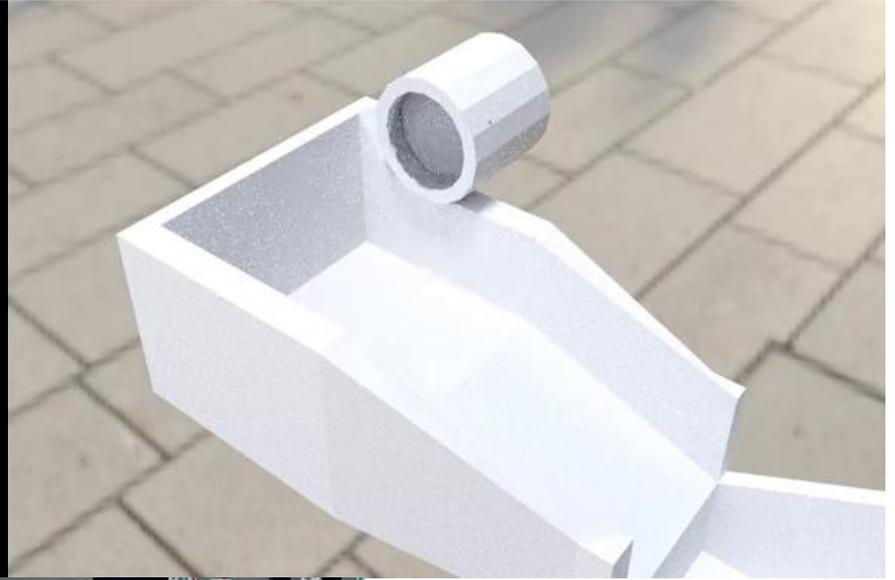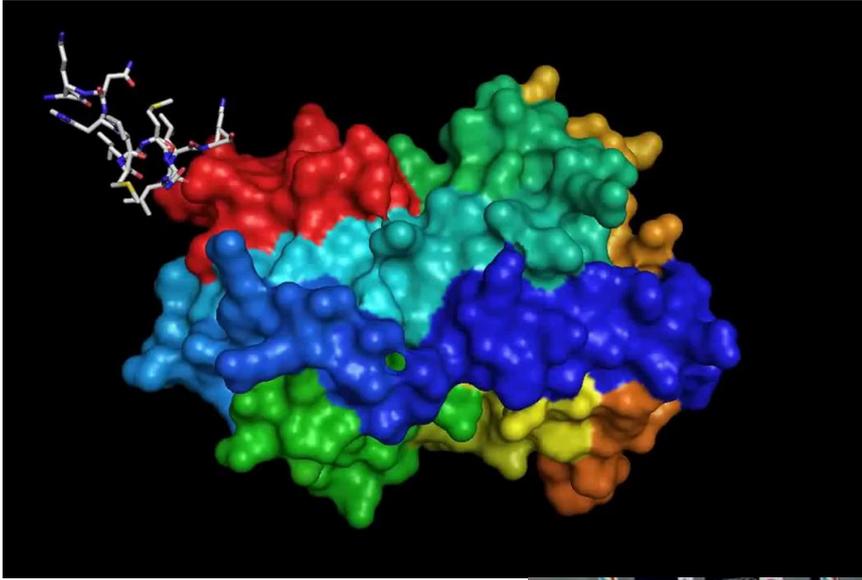# Simulation

- From latin *simulare* (to mimic or to fake)
- It is the imitation of a real-world process' or system's operation over time
- It allows to collect results long before a system is actually built (*what-if analysis*)
- It can be used to drive physical systems (*symbiotic simulation*)
- Widely used: medicine, biology, physics, economics, sociology, ...
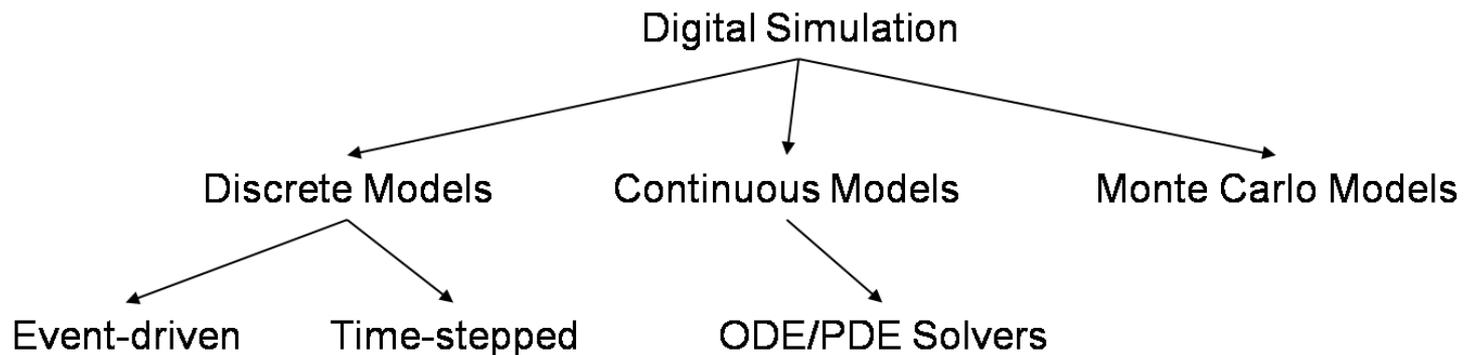
Some Examples

Main Categories of Simulation

- Continuous Simulation

- Monte Carlo Simulation

- Discrete-Event Simulation

Simulation Taxonomy

Wall-Clock Time vs Logical Time

- Two different notions of time are present in a simulation

- **Wall-Clock Time**: the *elapsed time* required to carry on a digital simulation (the shorter, the higher is the performance)

- **Logical Time**: the actual *simulated time*
  - Also referred to as *simulation time*

Continuous Simulation

- It is typically employed for modeling physical phenomena
  - Usually relies on a set of equations to be solved periodically

- Commonly physical phenomena are expressed via differential equations

- A continuous simulation involves repeatedly solving equations to update the *state* of the modeled phenomenon

An Example: Diffusion Equation

- Let's consider the Bidimensional Diffusion Equation Case:

$$\frac{\partial u}{\partial t} = \alpha \left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right)$$

- or, more compactly:

$$u_t = \alpha(u_{xx} + u_{yy})$$

An Example: Diffusion Equation

- We approximate *u(x, y, t)* by a discrete function $u_{i,j}^{(m)}$
  - *x = iΔx*
  - *y = iΔy*
  - *t = iΔt*
- This approximation is not enough for simulation: we must be able to compute a *future state* starting from the *current one*
- We use *finite difference* to transform it into a *recurrence relation*

Finite Difference

- A finite difference is a mathematical expression of the form *f(x+b) - f(x+a)*
  - Forward difference: $\Delta_h[f](x) = f(x+h) - f(x)$
  - Backward difference: $\nabla_h[f](x) = f(x) - f(x - h)$
  - Central difference: $\delta_h[f](x) = f(x + \frac{1}{2}h) - f(x - \frac{1}{2}h)$
- Using finite difference, the *finite-difference method* can be applied to solve differential equations
- Finite differences are used to *approximate derivatives*: it is a discretization method

An Example: Diffusion Equation

- Applying finite (forward) difference approximations to the derivatives we obtain:

$$\frac{u_{i,j}^{(m+1)} - u_{i,j}^{(m)}}{\Delta t} = \alpha \left[\left(\frac{u_{i+1,j}^{(m)} - 2u_{i,j}^{(m)} + u_{i-1,j}^{(m)}}{(\Delta x)^2} \right) + \left(\frac{u_{i,j+1}^{(m)} - 2u_{i,j}^{(m)} + u_{1,j-1}^{(m)}}{(\Delta y)^2} \right) \right]$$

- To simulate, we transform it into:

$$u_{i,j}^{(m+1)} = u_{i,j}^{(m)} + \alpha \Delta t \left[\left(\frac{u_{i+1,j}^{(m)} - 2u_{i,j}^{(m)} + u_{i-1,j}^{(m)}}{(\Delta x)^2} \right) + \left(\frac{u_{i,j+1}^{(m)} - 2u_{i,j}^{(m)} + u_{1,j-1}^{(m)}}{(\Delta y)^2} \right) \right]$$

- This gives us an expression of $u_{i,j}^{(m+1)}$

Stability of the Simulation

- This is an *approximation* of a continuous system
- Is the result correct *independently* of the selected time step?
- Stability reflects the sensitivity of Differential Equation solution to perturbations
- If the solutions are *stable,* they converge and perturbations are damped out
- When we step from an approximation to the next, we land on a different solution from what we started from

An Example: Diffusion Equation

- In case of 2D Heat Simulation, we rewrite u_t as:

$$u_{j,k}^{(m)} = u^{(m)} e^{i(\beta j \Delta x + \gamma k \Delta y)}$$

- The resulting amplification factor becomes:

$$\rho(\beta, \gamma) = 1 + 2\alpha \frac{\Delta t}{\Delta x^2}(\cos(\beta \Delta x) - 1) + 2\alpha \frac{\Delta t}{\Delta y^2}(\cos(\gamma \Delta y) - 1)$$

- Neumann boundary conditions lead to:

$$-1 \leq \rho(\beta, \gamma) \leq 1$$

An Example: Diffusion Equation

- We know that $-2 \leq \cos(\beta\Delta x) - 1 \leq 0$ and $-2 \leq \cos(\gamma\Delta y) - 1 \leq 0$

- The right-hand inequality holds for all β and γ

- The left-hand inequality leads to:

$$2\alpha\frac{\Delta t}{\Delta x^2} + 2\alpha\frac{\Delta t}{\Delta x^2} \leq 1$$

$$\Delta t \leq \frac{1}{2\alpha}\left(\frac{\Delta x^2 \Delta y^2}{\Delta x^2 + \Delta y^2}\right)$$

How is this useful programmatically?

- Simulation is an approximation of reality

- We want our approximation to resemble reality as much as possible

- Setting a simulation time step such that:

$$\Delta t > \frac{1}{2\alpha} \left(\frac{\Delta x^2 \Delta y^2}{\Delta x^2 + \Delta y^2} \right)$$

gives a simulation which is *incorrect*

Initial and Boundary Conditions

- $u_{i,j}^{(m+1)}$ is derived using $u_{i,j}^{(m)}$
- Then, we must give a numerical value to $u_{i,j}^{(0)}$
- Furthermore, we must specify boundary conditions to the Laplacian
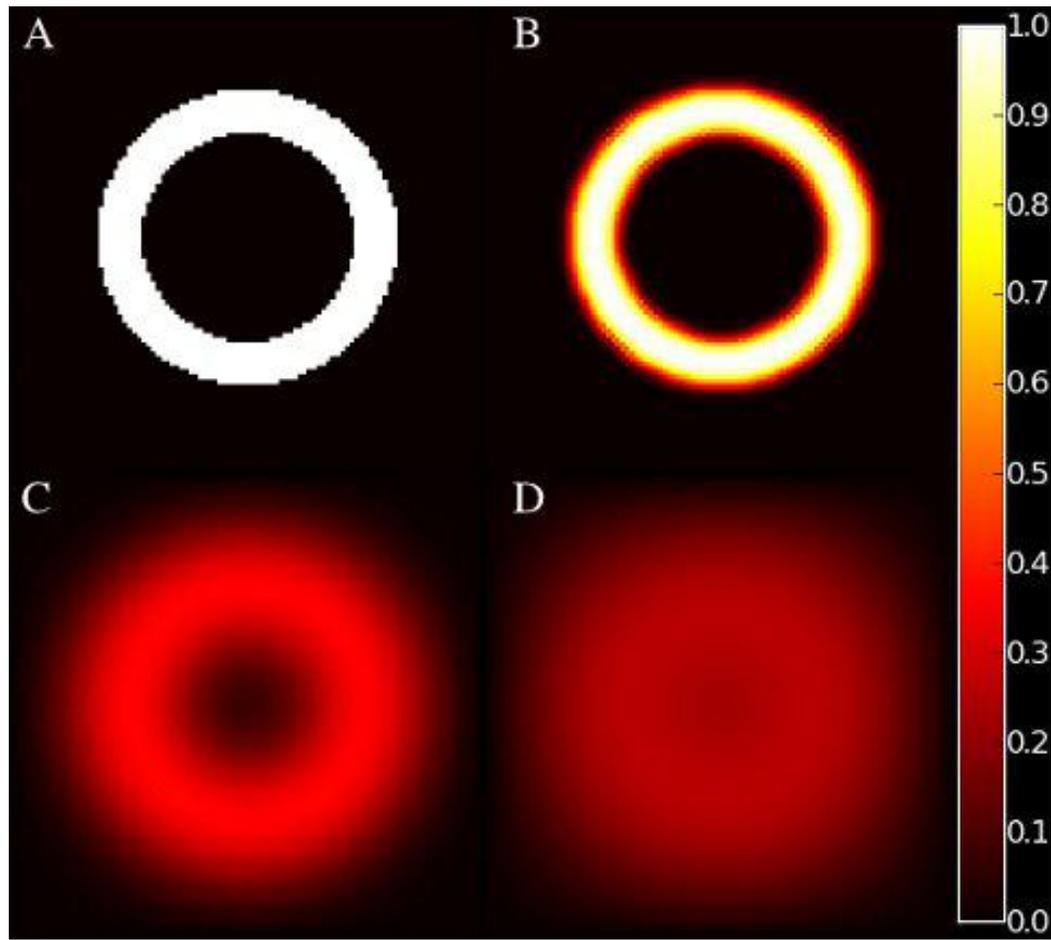 - We can arbitrarily set it to 0

ring at (0.5, 0.5)

$$u_{i,j}^{(0)} = \begin{cases} 1 & \text{if } 0.05 \leq (i\Delta x - 0.5)^2 + (j\Delta y - 0.5)^2 \leq 0.1 \\ 0 & \text{otherwise} \end{cases}$$

Evolution of the System

Coding the Problem

- We *repeatedly* solve the differential equations
- We rely on a *loop* to do this:

```python
for m in range(1, timesteps+1):
    recompute_u(u, ui)
```

- The code to update the state of the system looks like:

```python
1  def recompute_u(u, ui):
2      for i in range(1,nx-1):
3          for j in range(1,ny-1):
4              uxx = ( ui[i+1,j] - 2*ui[i,j] + ui[i-1, j] )/dx2
5              uyy = ( ui[i,j+1] - 2*ui[i,j] + ui[i, j-1] )/dy2
6              u[i,j] = ui[i,j]+dt*a*(uxx+uyy)
```

Coding the Problem: Initial Conditions

```python
1  dx=0.01 # Interval size in x-direction.
2  dy=0.01 # Interval size in y-direction.
3
4  nx = int(1/dx)
5  ny = int(1/dy)
6
7  ui = sp.zeros([nx,ny])
8  u = sp.zeros([nx,ny])
9
10 # Now, set the initial conditions (ui).
11 for i in range(nx):
12     for j in range(ny):
13         if ( ( (i*dx-0.5)**2+(j*dy-0.5)**2 <= 0.1)
14             & ((i*dx-0.5)**2+(j*dy-0.5)**2>=.05) ):
15                 ui[i,j] = 1
```

EXAMPLE SESSION

Heat Diffusion Simulation in Python

What Lessons Have we Learnt?

- Before going distributed, we must be sure that the *sequential* implementation is efficient

- Stability conditions are not only a mathematician's concern!

- Continuous simulation is actually an approximation of the continuous behaviour of a system

Monte Carlo Simulation

- It is generally used to evaluate some property that is time independent

- It tries to explore densely the whole space of parameters of the phenomenon
  - Monte Carlo simulations sample probability distribution for each variable to produce hundreds or thousands of possible outcomes

- It is used to find (approximate) solutions of mathematical problems involving a high number of variables that cannot be easily solved analytically
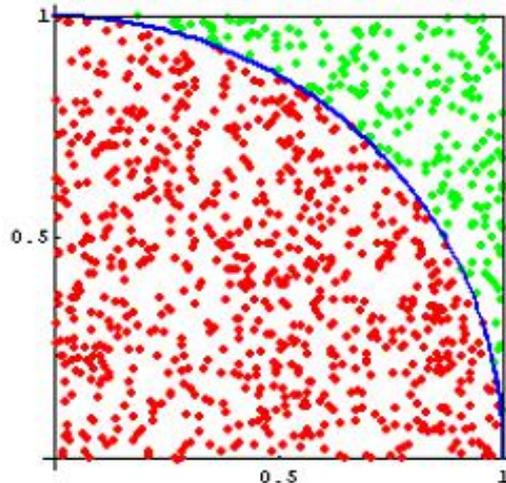
An Example: Computing π

- Let us consider a circle with *r = 1*

- The area of the circle is *πr² = π*

- The area of the sourrounding square is *(2r)² = 2² = 4*

- The ratio of the areas is:

$$\rho = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4} = 0.7853981633974483$$
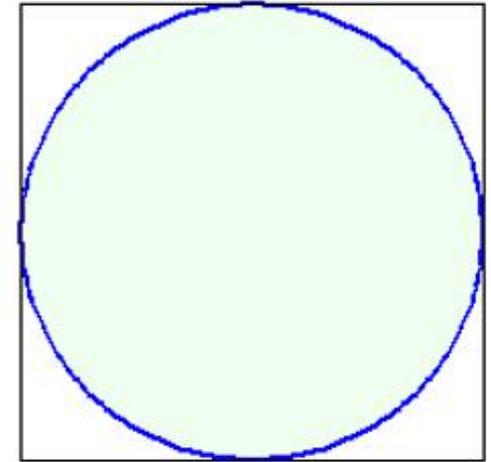
An Example: Computing π

- Randomly select points $\{(x_i, y_i)\}^n_{i=1}$ in the square
- Determine the ratio $\rho = \dfrac{m}{n}$
  - m is the number of points such that $x_i^2 + y_i^2 \leq 1$
- Since $\rho = \dfrac{\pi}{4}$, then $\pi = \rho \cdot 4$

EXAMPLE SESSION

Monte Carlo PI Approximation

Event-Driven Programming

- Event-Driven Programming is a programming paradigm in which the flow of the program is determined by *events*
  - Sensors outputs
  - User actions
  - Messages from other programs or threads
- Based on a *main loop* divided into two phases:
  - Event selection/detection
  - Event handling
- Events resemble what *interrupts* do in hardware systems

Event Handlers

- An event handler is an *asynchronous callback*
- Each event represents a piece of application-level information, delivered from the underlying framework:
    - In a GUI events can be mouse movements, key pression, action selection, . . .
- Events are processed by an *event dispatcher* which manages associations between events and event handlers and notifies the correct handler
- Events can be queued for later processing if the involved handler is busy at the moment

Discrete Event Simulation (DES)

- A *discrete event* occurs at an instant in time and marks a change of state in the system

- DES represents the operation of a system as a chronological sequence of events

- If the simulation is run on top of a parallel/distributed system, it's named Parallel Discrete Event Simulation (PDES)

DES Building Blocks

- **Clock**
  - Independently of the measuring unit, the simulation must keep track of the current simulation time
  - Being discrete, time *hops* to the next event's time

- **Event List**
  - At least the *pending event set* must be maintained by the simulation architecture
  - Events can arrive at a higher rate than they can be processed

- **Random Number Generators**

- **Statistics**

- **Ending Condition**

DES Skeleton

```
 1:  procedure INIT
 2:      End ← false
 3:      initialize State, Clock
 4:      schedule INIT
 5:  end procedure
 6:
 7:  procedure SIMULATION-LOOP
 8:      while End == false do
 9:          Clock ← next event's time
10:          process next event
11:          Update Statistics
12:      end while
13:  end procedure
```

Implementation of a DES Kernel

- *General-purpose Simulation* is easy for DES
  - No notion of *model* in the main-loop pseudocode!
- Only prerequisites:
  - The model must implement actual *handlers*
  - The model requires APIs to inject new events in the system and pass entities' states from the kernel
- Multiple models can be run on the same kernel
  - Core reuse
  - Model-independent optimization of the kernel

Data Structures for Simulation: Priority Queue

- Is an abstract data type similar to a regular queue
- Elements have a priority associated with each of them
- An element with a high priority is served before
- Operations:
  - insert with priority: add an element to the queue with associated priority
  - pull highest priority element: remove the element from the queue that has the highest priority, and return it
- Highest priority can be either minimum or maximum value
- It can be used to implement the FEL
  - What about the ordering of simultaneous events?

Data Structures for Simulation: Calendar Queue

- A fast priority queue implementation (Brown, 1988)
- Composed of n buckets, each of width (or covering time) *w*
- Notion of *current time*
- Items with priority *p* > current time go into bucket:

$$\left\lfloor \frac{p}{w} \right\rfloor \bmod n$$

- *n* and *w* should be chosen so as to have few elements per bucket
  - Double or halve *n* and change *w* if the number of items grows or shrinks too much

Data Structures for Simulation: Calendar Queue

- Changing n involves no more that 3 moves of each event in the worst case

- *w* should be the average separation between events

- Calendar Queue has amortized O(1) operations cost

| 0 | 1 | 2 | 3 | 4 |

5 buckets
width = 10
current time = 63

103
107   66   88   195

elements' priorities
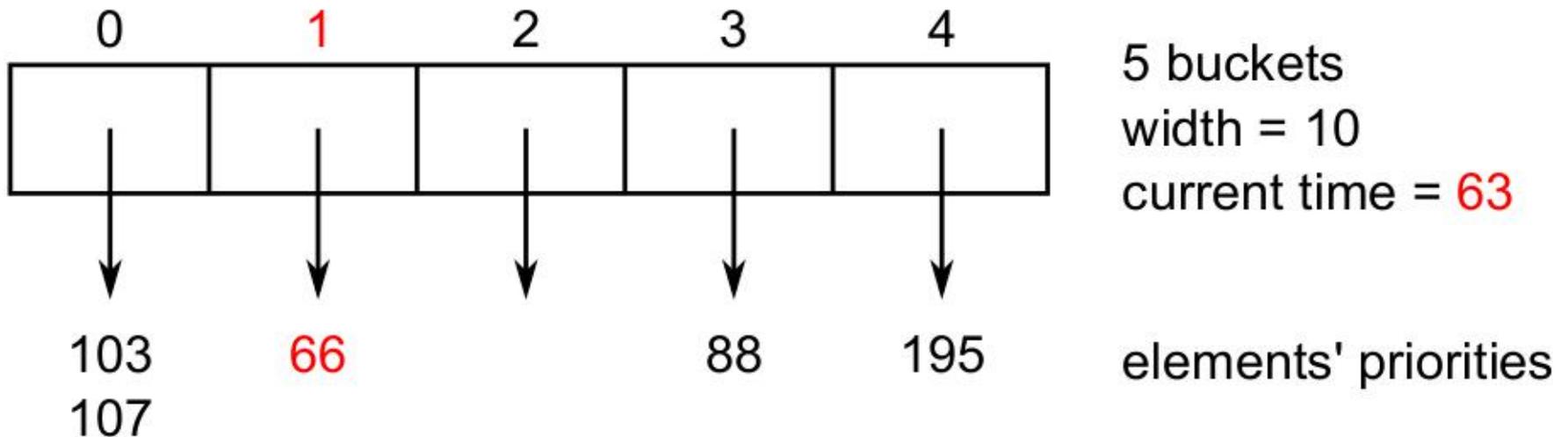
API to Schedule Events and Set State

```c
1  typedef struct _platform_event {
2      double timestamp;
3      unsigned int destination;
4      unsigned int event_type;
5      unsigned int size;
6      void *payload;
7  } platform_event;
8
9  static calqueue *fel;
10 static unsigned long long int processed_events = 0;
11 static double simulation_time = 0;
12 static unsigned int current_entity = UINT_MAX;
13 static void **simulation_states;
14
15 unsigned int num_entities;
```

API to Schedule Events and Set State

```c
16  extern void ScheduleNewEvent(unsigned int receiver, double
        timestamp, unsigned int event_type, void *event_content,
        unsigned int event_size) {
17      platform_event *e;
18
19      // Sanity checks
20      if(timestamp < simulation_time) {
21          fprintf(stderr, "Entity %d is trying to send events in the
                  past. Current time: %f, scheduled time: %f\n",
                  current_entity, simulation_time, timestamp);
22          exit(EXIT_FAILURE);
23      }
24
25      // Populate the message data structure
26      e = malloc(sizeof(platform_event));
27      bzero(e, sizeof(platform_event));
```

API to Schedule Events and Set State

```c
28      e->destination = receiver;
29      e->timestamp = timestamp;
30      e->event_type = event_type;
31      e->size = event_size;
32      e->payload = malloc(event_size);
33      memcpy(e->payload, event_content, event_size);
34
35      // Put the event in the Calendar Queue
36      calqueue_put(fel, timestamp, e);
37 }
38
39 extern void SetState(void *state) {
40      simulation_states[current_entity] = state;
41 }
```

Initialization and Main Loop

```c
1  int main(int argc, char **argv) {
2      // Initialize data structures to handle entities
3      num_entities = (int)strtol(argv[1], NULL, 10);
4      printf("Initializing %d entities...\n", num_entities);
5      simulation_states = malloc(sizeof(void *) * num_entities);
6
7      // Allocate and initialize FEL
8      fel = malloc(sizeof(calqueue));
9      calqueue_init(fel);
10
11     // Schedule INIT to entities
12     for(i = 0; i < num_entities; i++) {
13         current_entity = i;
14         ProcessEvent(i, 0, INIT, NULL, 0, NULL);
15     }
16
```

Initialization and Main Loop

```
17      // Main loop
18      while(!calqueue_empty(fel)) {
19          e = calqueue_get(fel);
20
21          // Update current entity and simulation clock
22          current_entity = e->destination;
23          simulation_time = e->timestamp;
24
25          ProcessEvent(current_entity, simulation_time, e->event_type,
                  e->payload, e->size, simulation_states[current_entity])
                  ;
26
27          processed_events++;
28
29
```

Initialization and Main Loop

```
31          if(e->payload != NULL)
32              free(e->payload);
33          free(e);
34      }
35
36      printf("Simulation complete after processing %llu events\n",
           processed_events);
37
38      free(simulation_states);
39      free(fel);
40  }
```

Personal Communication Service

- Networking System for mobile devices
- Interesting to study how different configurations behave
- Coverage area modeled as a set of adjacent hexagons
- Explicit modeling of channel allocation
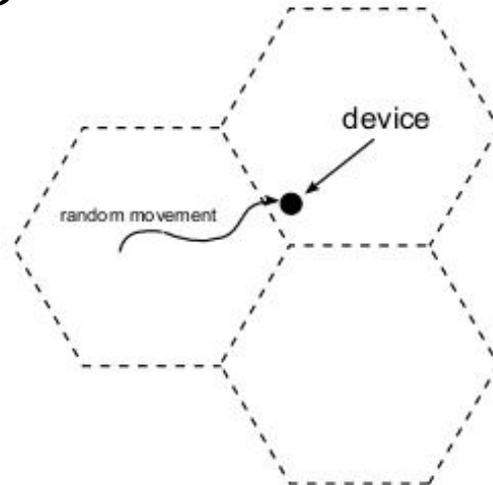
EXAMPLE SESSION

Personal Communication Service

Parallel Discrete Event Simulation

- To increase the overall performance, DES models can be run on top of multiple computing nodes
  - Distributed and/or concurrent simulation
- The main goal is *transparency*
- Simulation models should not be modified

Traditional PDES execution support
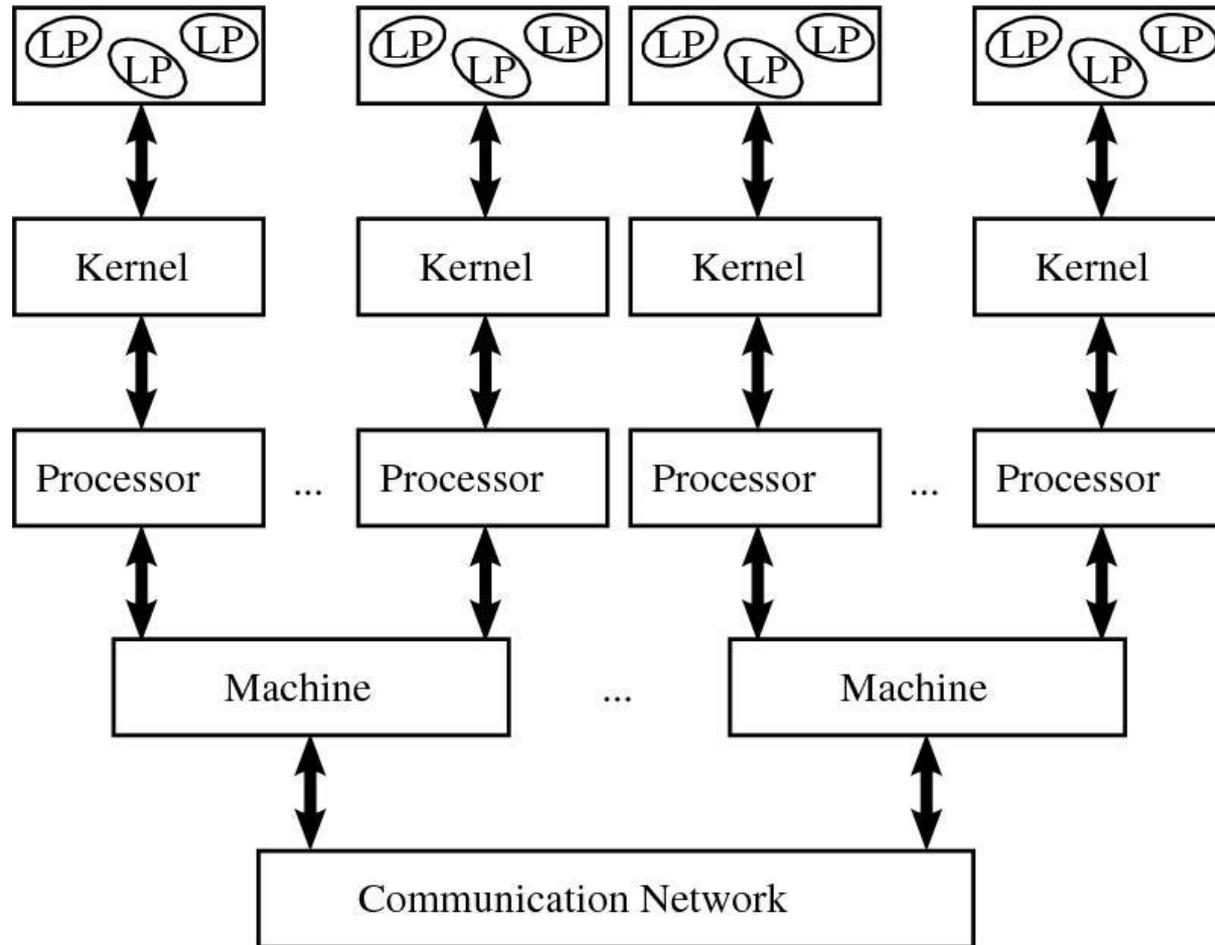
Why are multicores important?
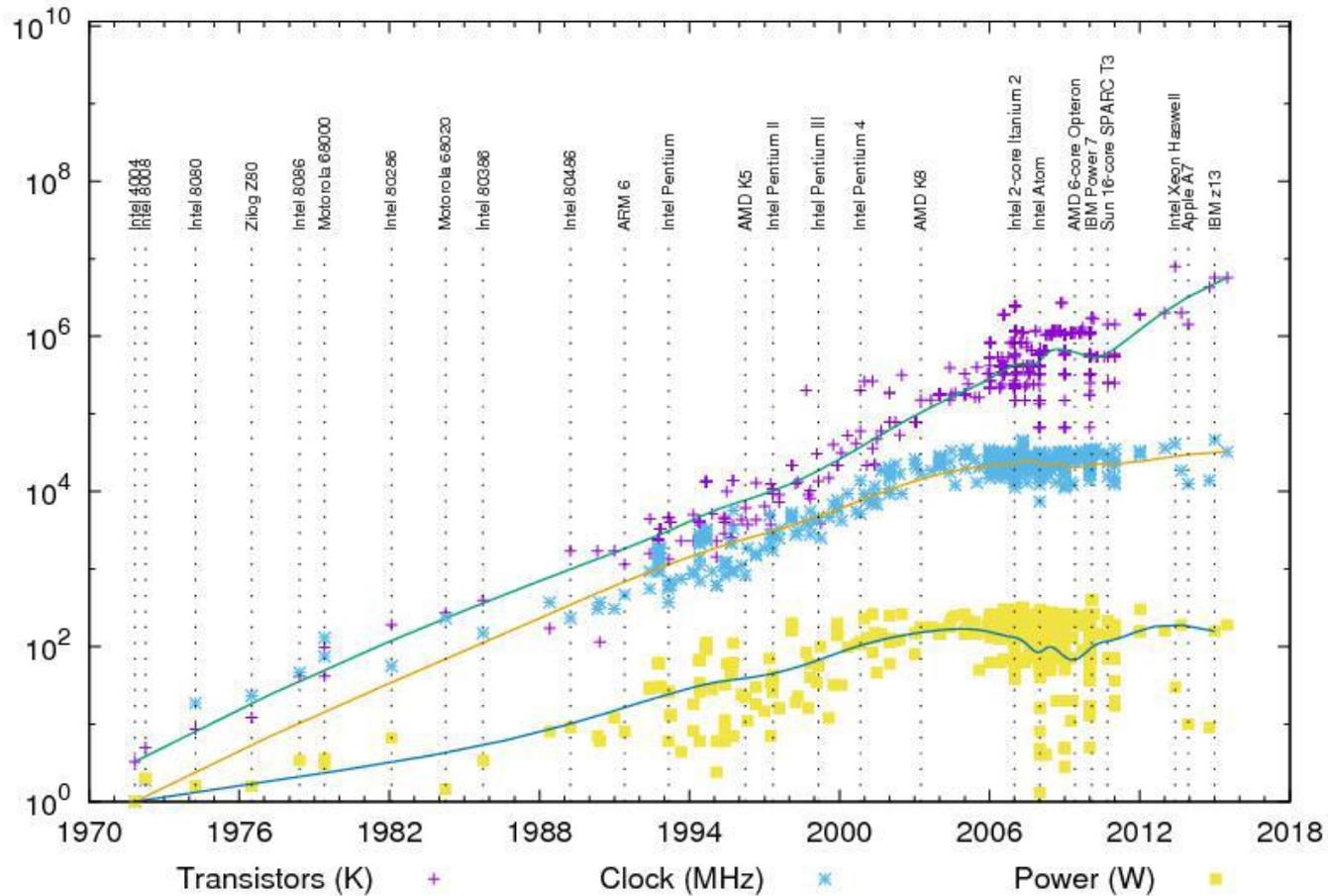
Revisited PDES Architecture
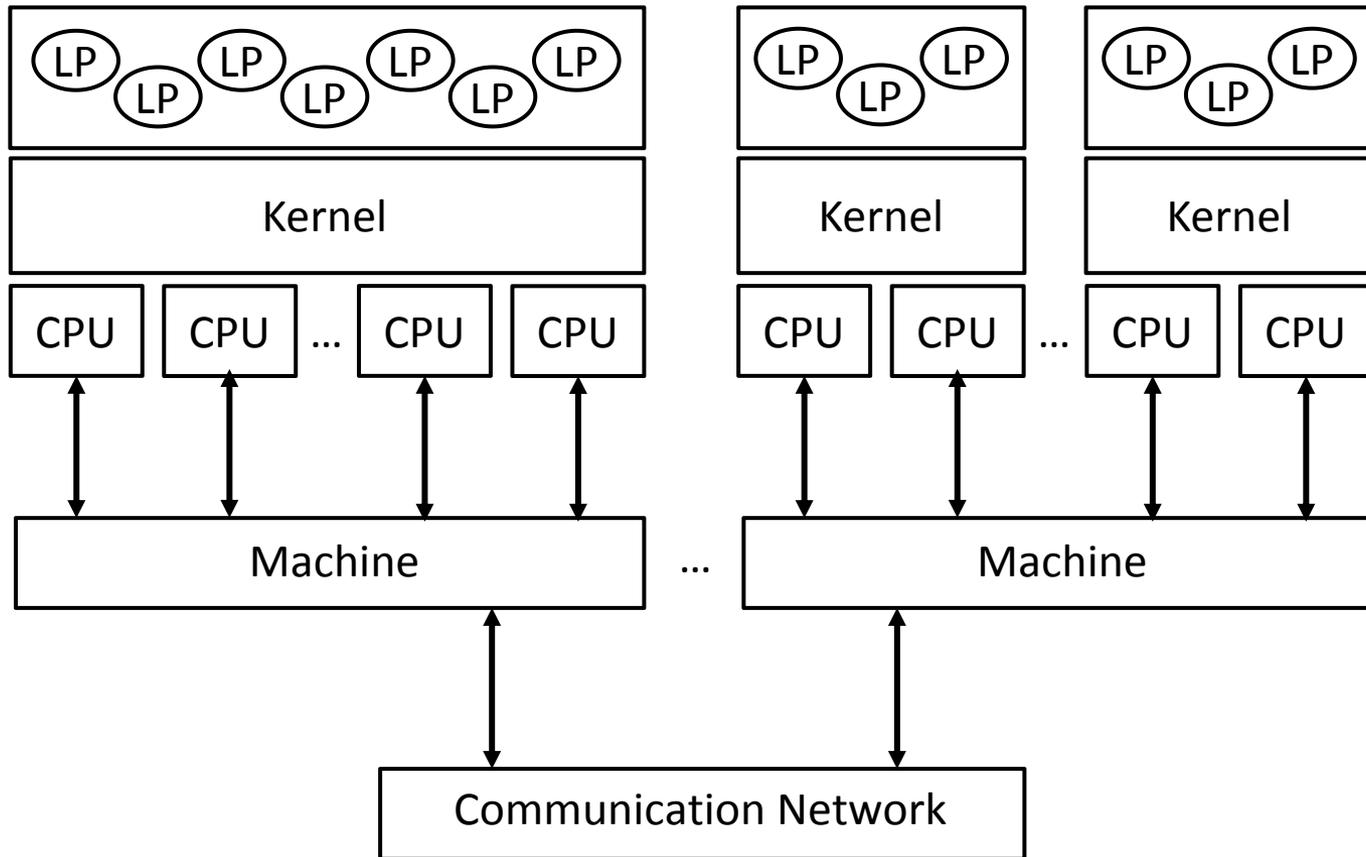
The Synchronization Problem

- Consider a simulation program composed of several *logical processes* exchanging timestamped messages

- Consider the *sequential execution*: this ensures that events are processed in timestamp order

- Consider the *parallel execution*: the greatest opportunity arises from processing events from different LPs concurrently

- Is *correctness* always ensured?
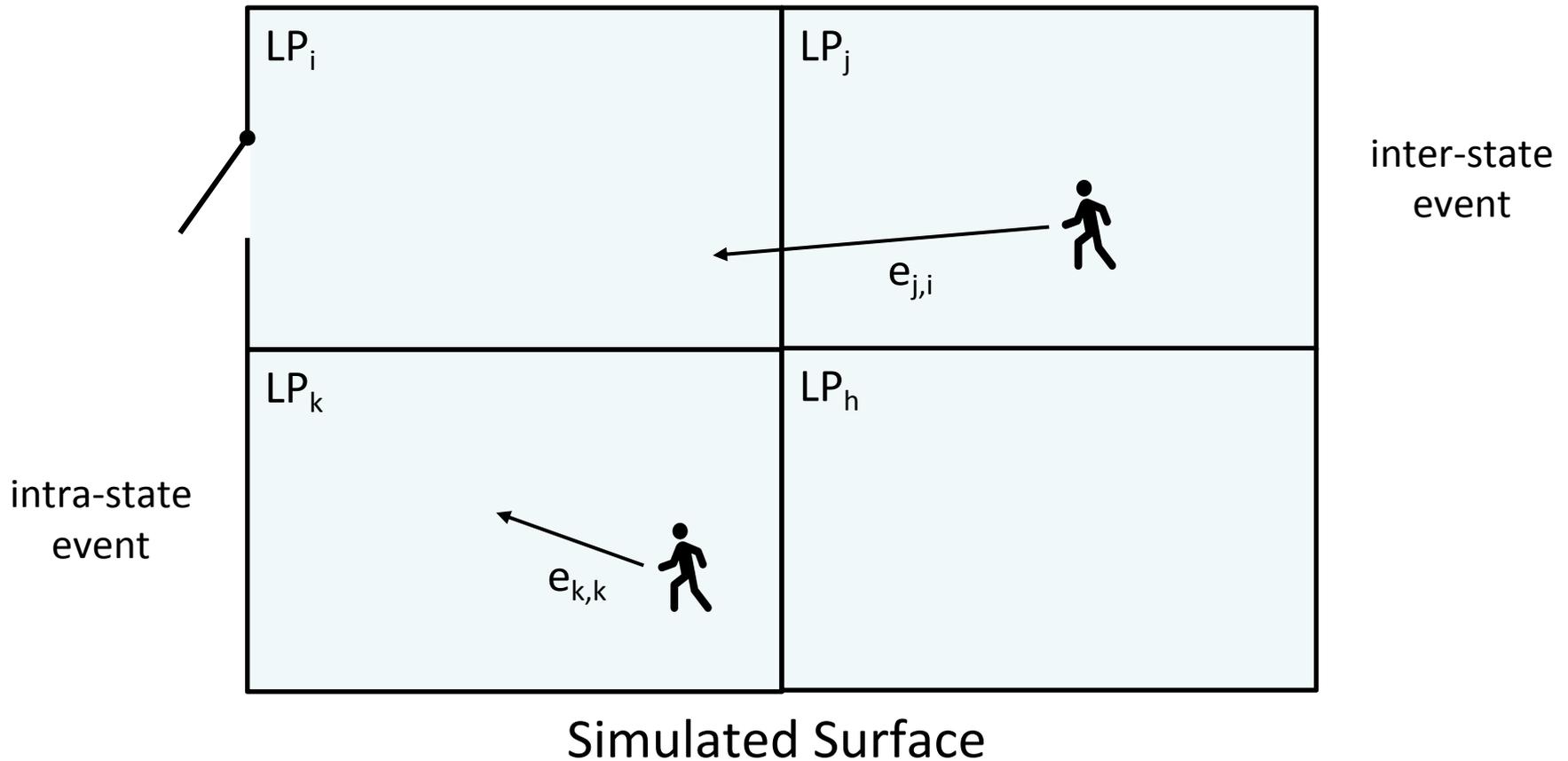
The Synchronization Problem

The Synchronization Problem

local virtual time (LVT)

LP$_i$

LP$_j$ ts = 2

ts = 4

e$_{j,i}$

inter-state event

CAUSALITY VIOLATION

LP$_k$

LP$_h$ ts = 7

intra-state event

e$_{k,k}$

Simulated Surface
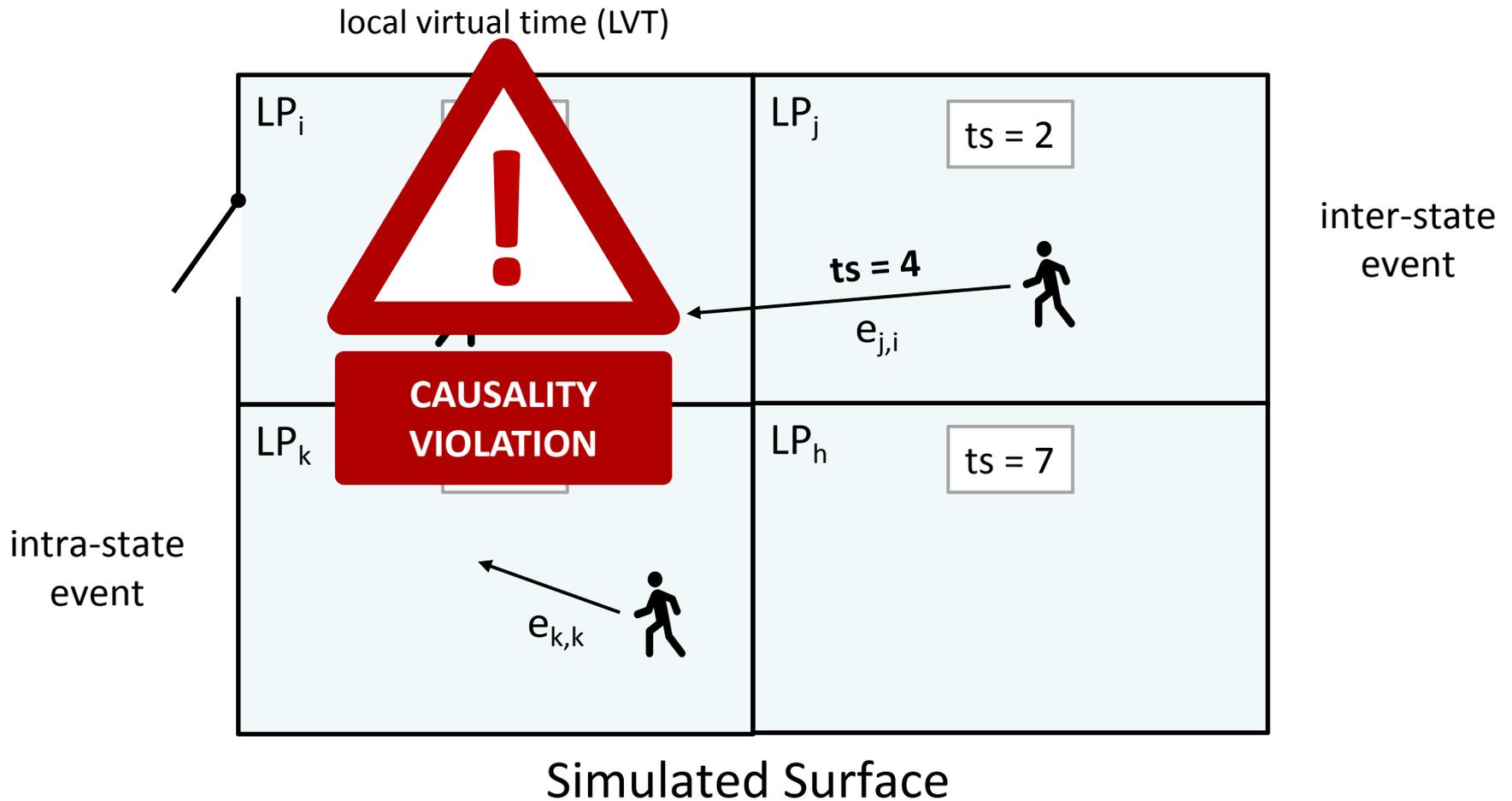
The Synchronization Problem
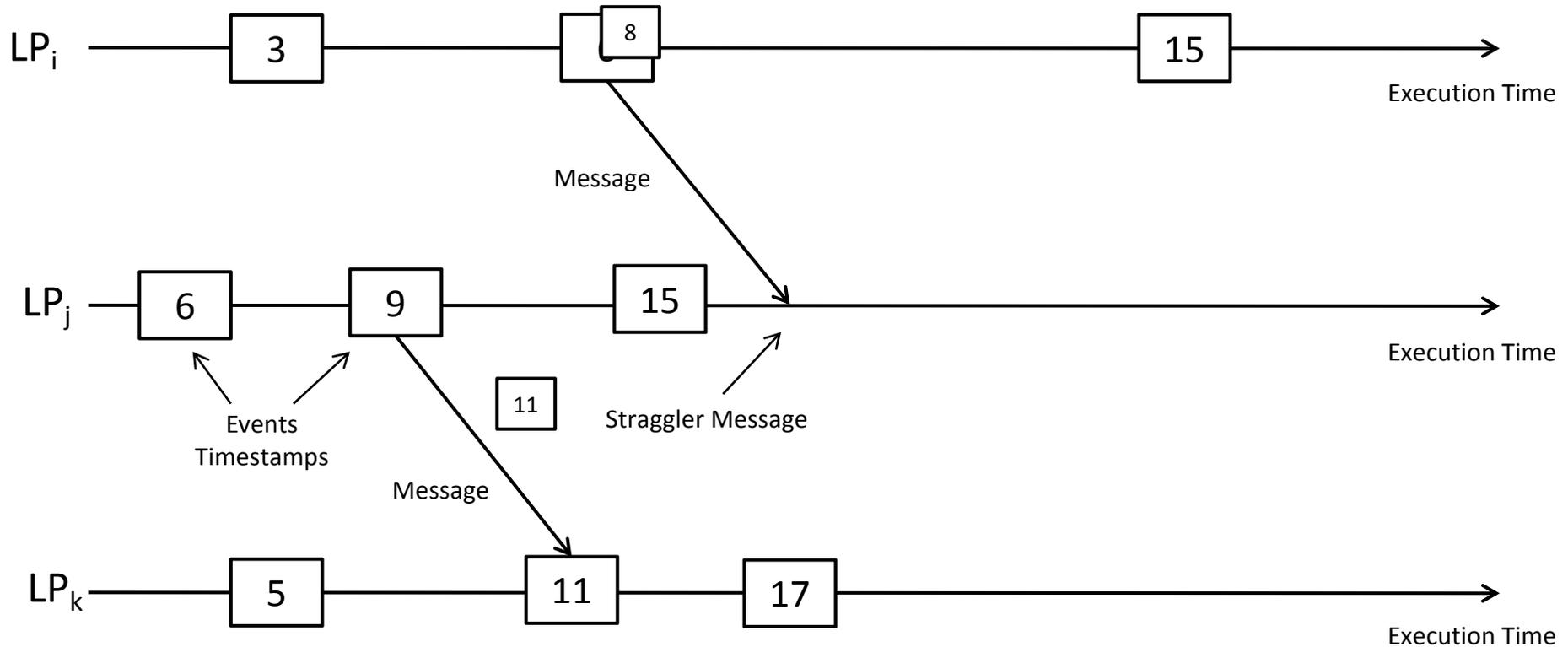
Conservative Synchronization

- Consider the LP with the *smallest* clock value at some instant T in the simulation's execution

- This LP could generate events relevant to every other LP in the simulation with a timestamp T

- No LP can process any event with timestamp larger than T

Conservative Synchronization

- If each LP has a *lookahead* of *L*, then any new message sent by al LP must have a timestamp of at least *T* + *L*

- Any event in the interval [*T, T* + *L*] can be safely processed

- *L* is intimately related to details of the simulation model

Optimistic Synchronization: Time Warp

- There are no state variables that are shared between LPs
- Communications are assumed to be reliable
- LPs need not to send messages in timestamp order
- **Local Control Mechanism**
  – Events not yet processed are stored in an *input queue*
  – Events already processed are not discarded
- **Global Control Mechanism**
  – Event processing can be *undone*
  – A-posteriori detection of causality violation

The Synchronization Problem

local virtual time (LVT)

LP$_i$ ts = **9**

LP$_j$ ts = 2

ts = 4

e$_{j,i}$

inter-state event

LP$_k$ ts = 5

LP$_h$ ts = 7

intra-state event

e$_{k,k}$

Simulated Surface
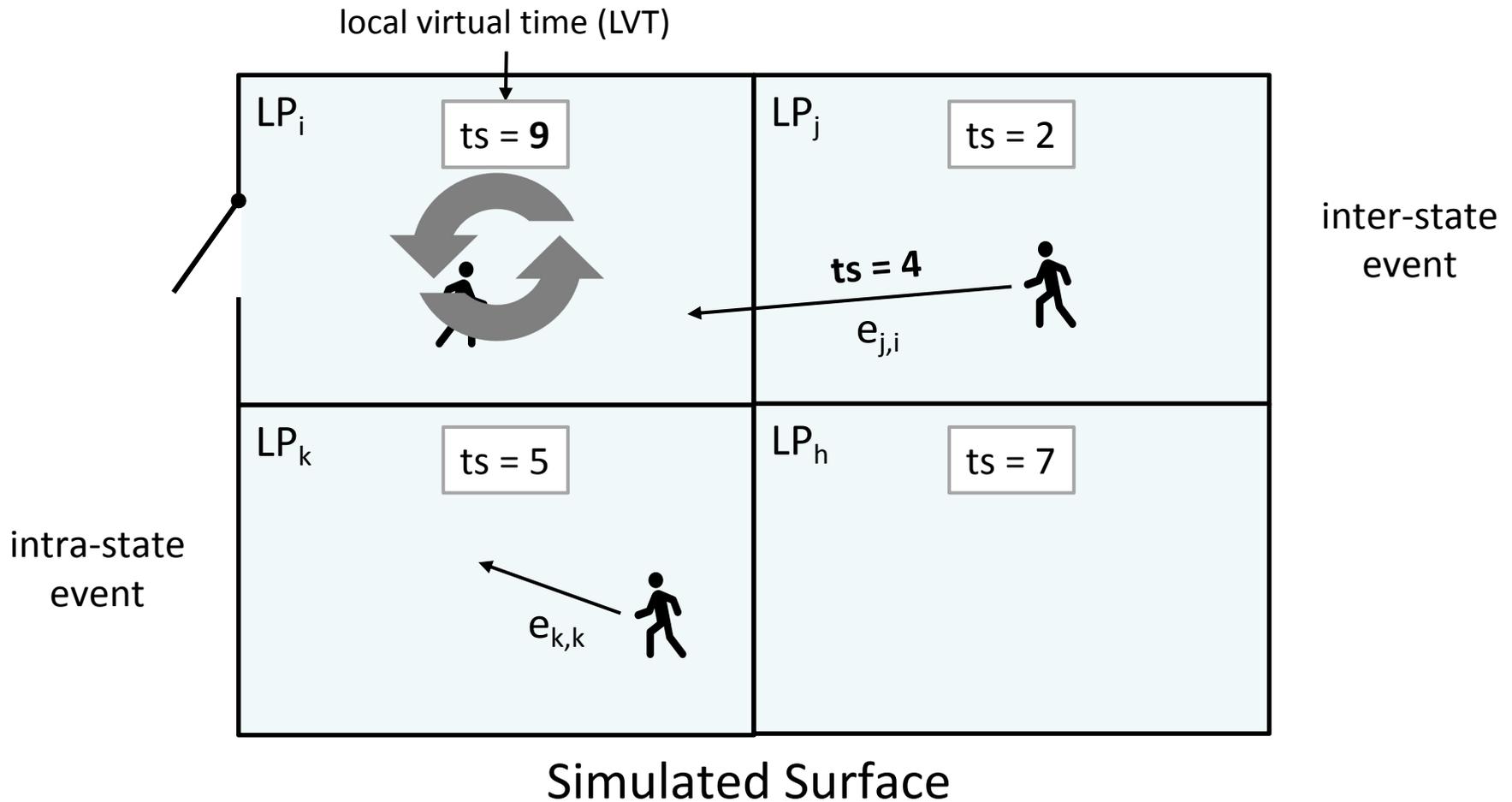
Time Warp: State Recoverability
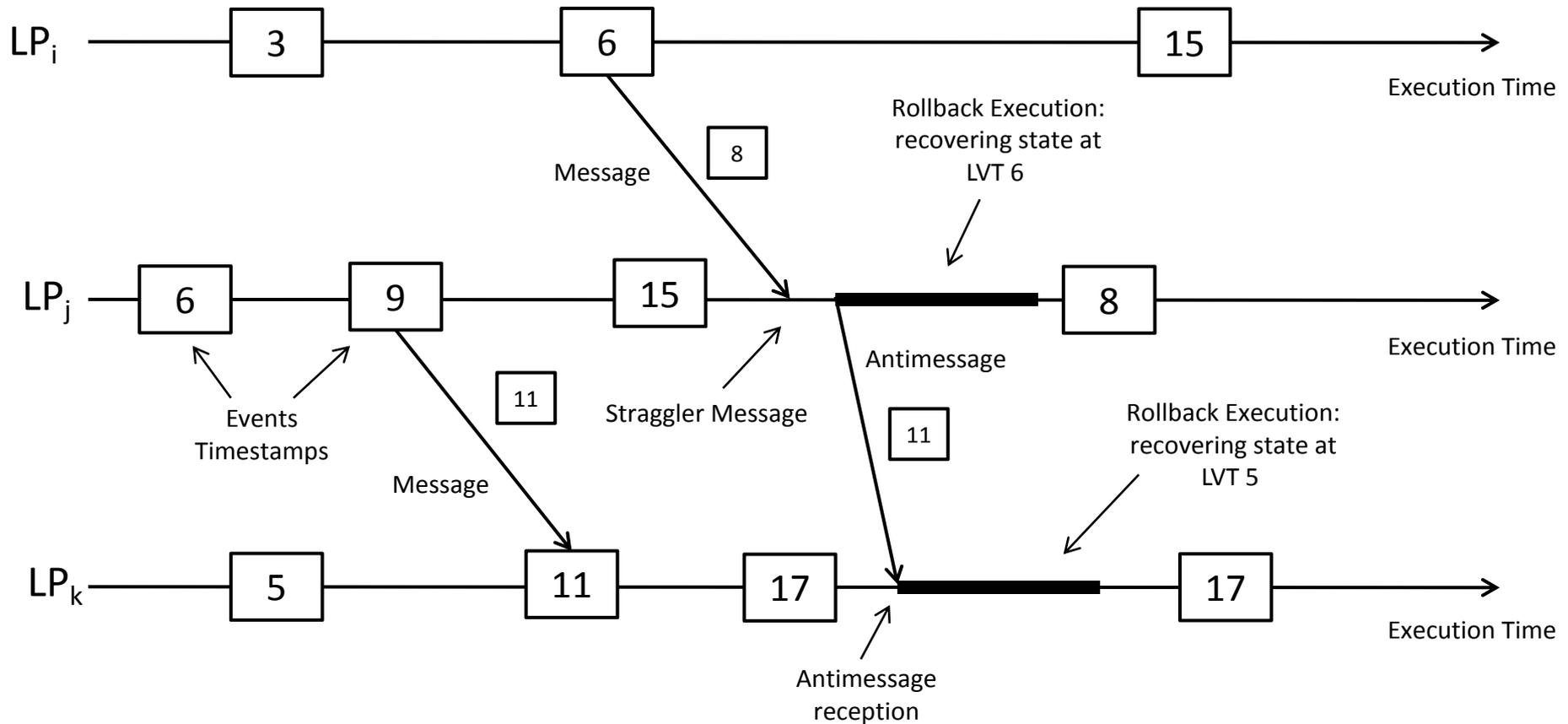
Rollback Operation

- The rollback operation is fundamental to ensure a correct speculative simulation

- Its *time critical*: it is often executed on the *critical path* of the simulation engine

- 30+ years of research have tried to find optimized ways to increase its performance

State Saving and Restore

- The traditional way to support a rollback is to rely on state saving and restore

- A state queue is introduced into the engine

- Upon a rollback operations, the "closest" log is picked from the queue and restored

- What are the *technological* problems to solve?

- What are the *methodological* problems to solve?

State Saving and Restore

State
Queue ————————————————————————————→

Simulation Time

Input
Queue ————————————————————————————→

Simulation Time

Output
Queue ————————————————————————————→

Simulation Time

State Saving and Restore

State
Queue
→ Simulation Time

Input
Queue
| 3 | 5.5 | 7 | 15 | 21 | 33 |
→ Simulation Time

Output
Queue
→ Simulation Time

State Saving and Restore

State
Queue

Simulation Time

bound

Input
Queue

| 3 | 5.5 | 7 | 15 | 21 | 33 |

Simulation Time

Output
Queue

Simulation Time

State Saving and Restore

State Queue

Simulation Time

Input Queue

bound

| 3 | 5.5 | 7 | 15 | 21 | 33 |

Simulation Time

Output Queue

| 3 | 3 | 3 | 3 |

Simulation Time

State Saving and Restore

State Queue

3

Simulation Time

bound

Input Queue

3   5.5   7   15   21   33

Simulation Time

Output Queue

3   3   3   3

Simulation Time

State Saving and Restore

State Saving and Restore

State Queue

3   5.5

Simulation Time

bound

Input Queue

3   5.5   7   15   21   33

Simulation Time
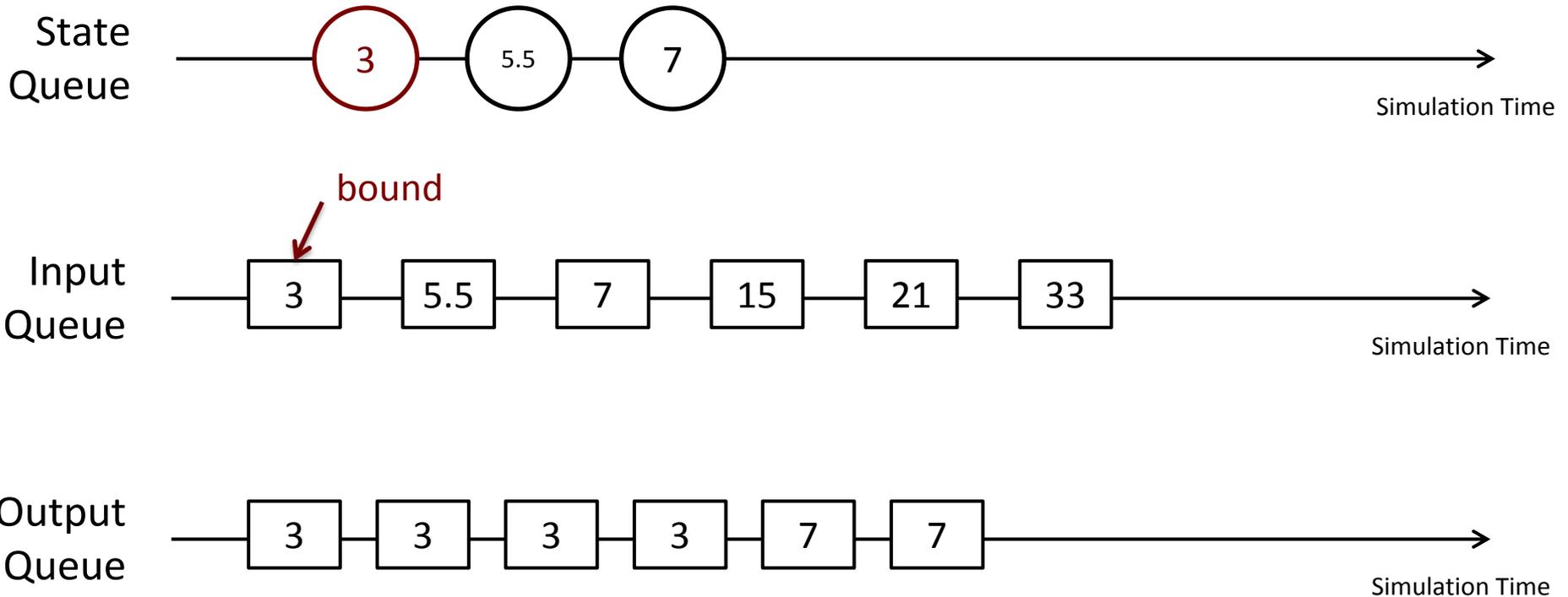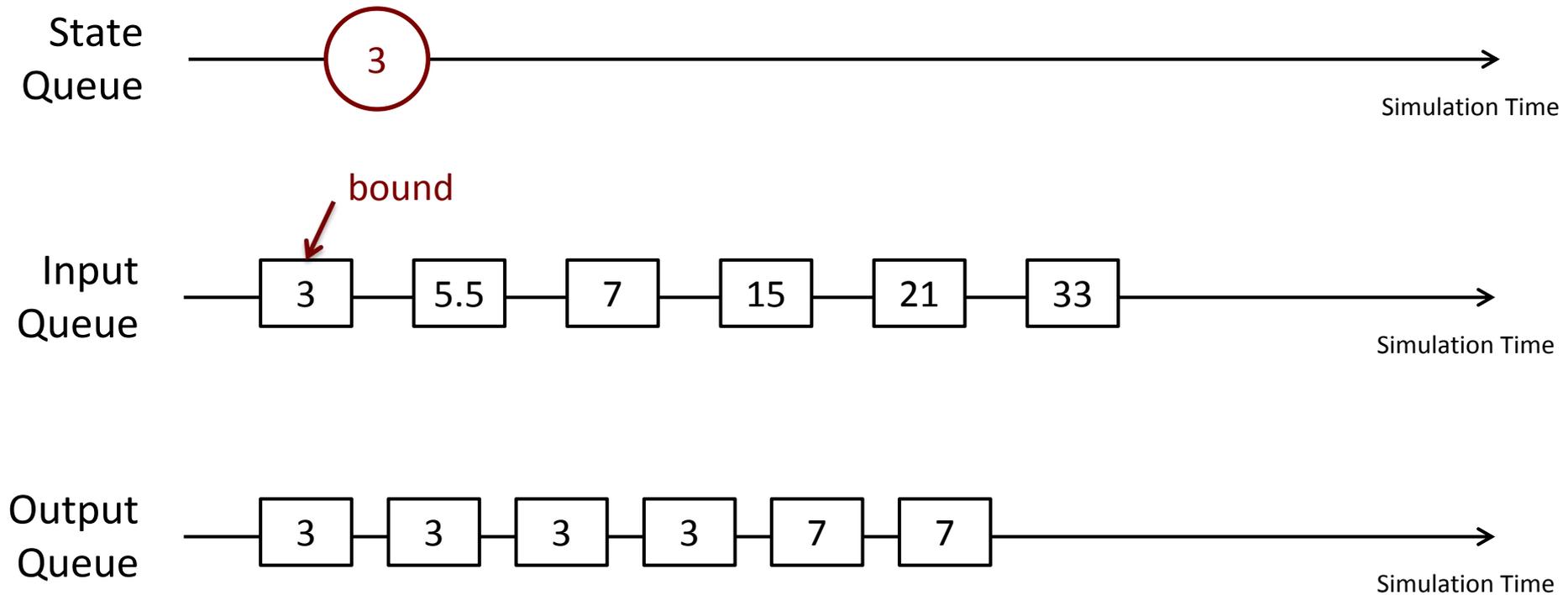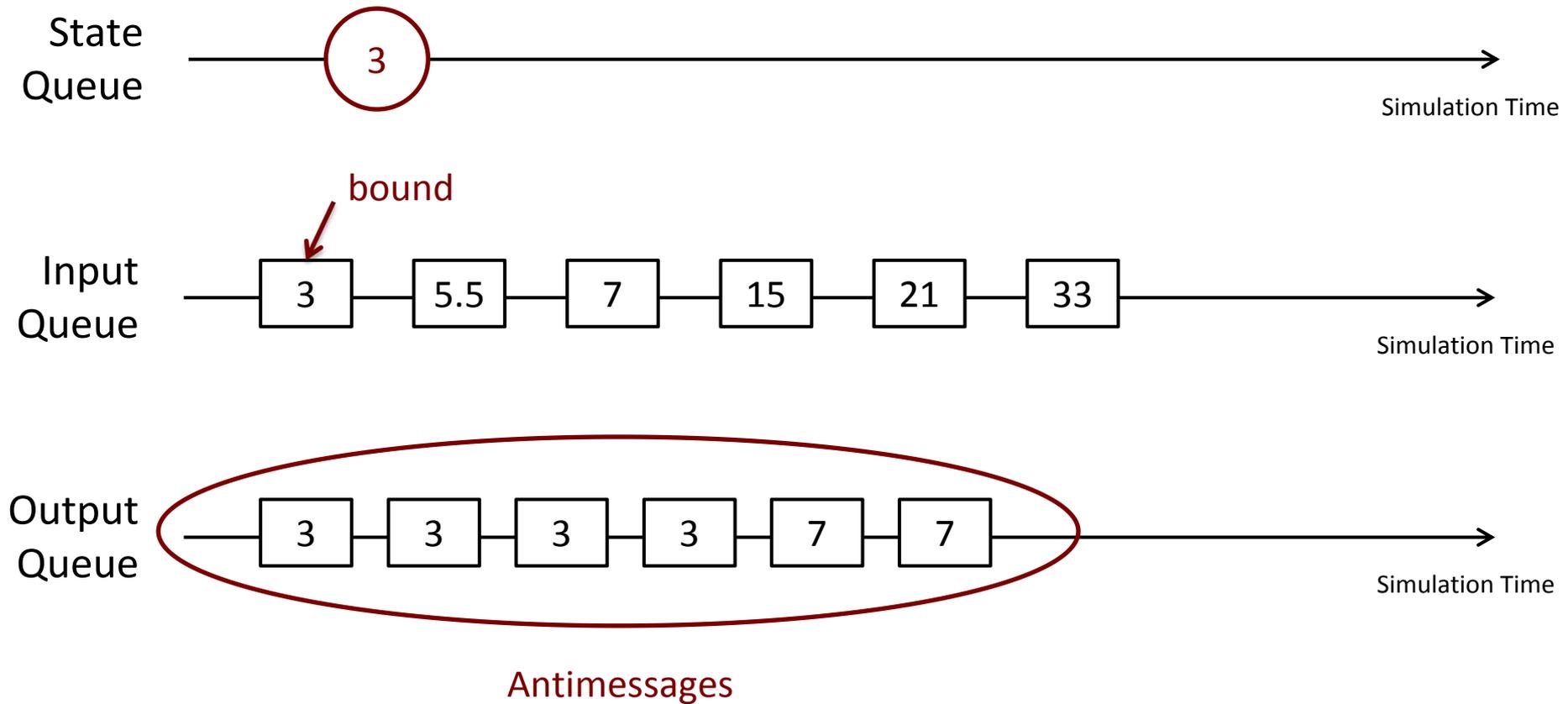
Output Queue

3   3   3   3

Simulation Time

State Saving and Restore

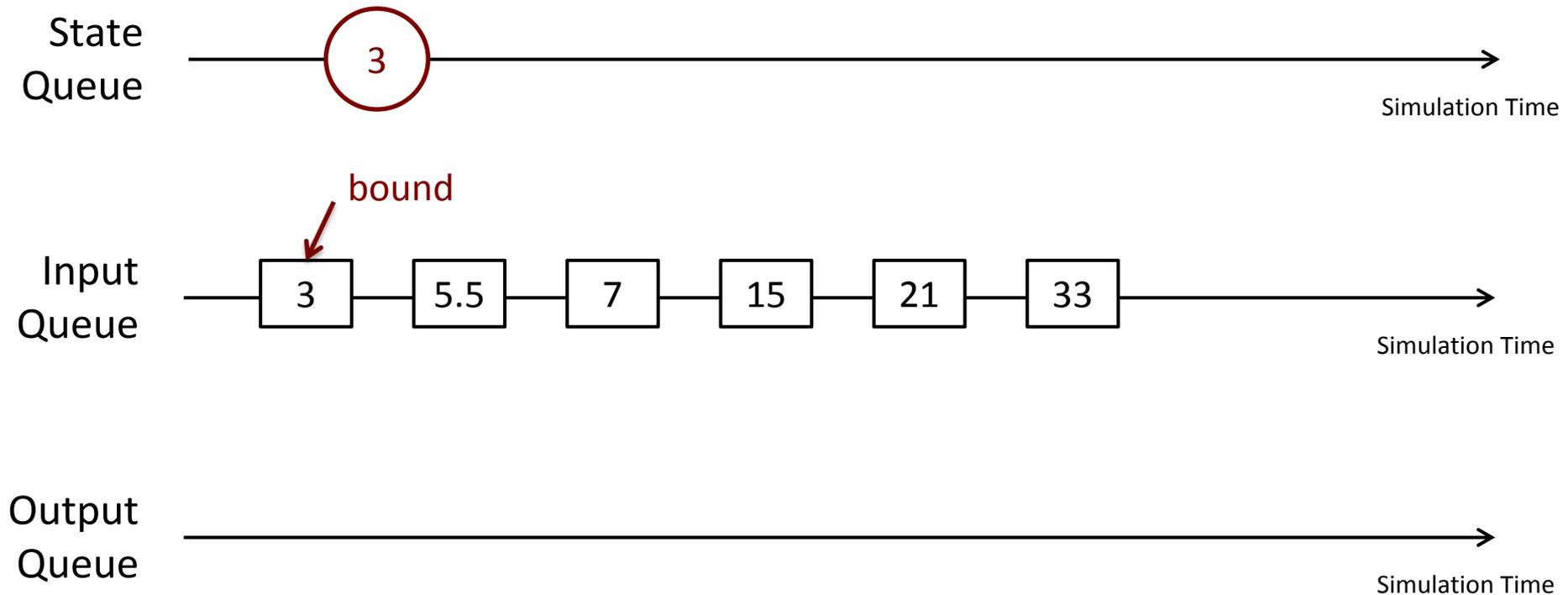# State Saving and Restore

State Saving and Restore

State Queue

3   5.5   7   →   Simulation Time

bound

Input Queue

3   5.5   7   15   21   33   →   Simulation Time

Output Queue

3   3   3   3   7   7   →   Simulation Time

State Saving and Restore

State Queue

Input Queue

Output Queue

3

3

3

3.7

3   3   3   7   7

Simulation Time

Simulation Time

Simulation Time

State Saving and Restore

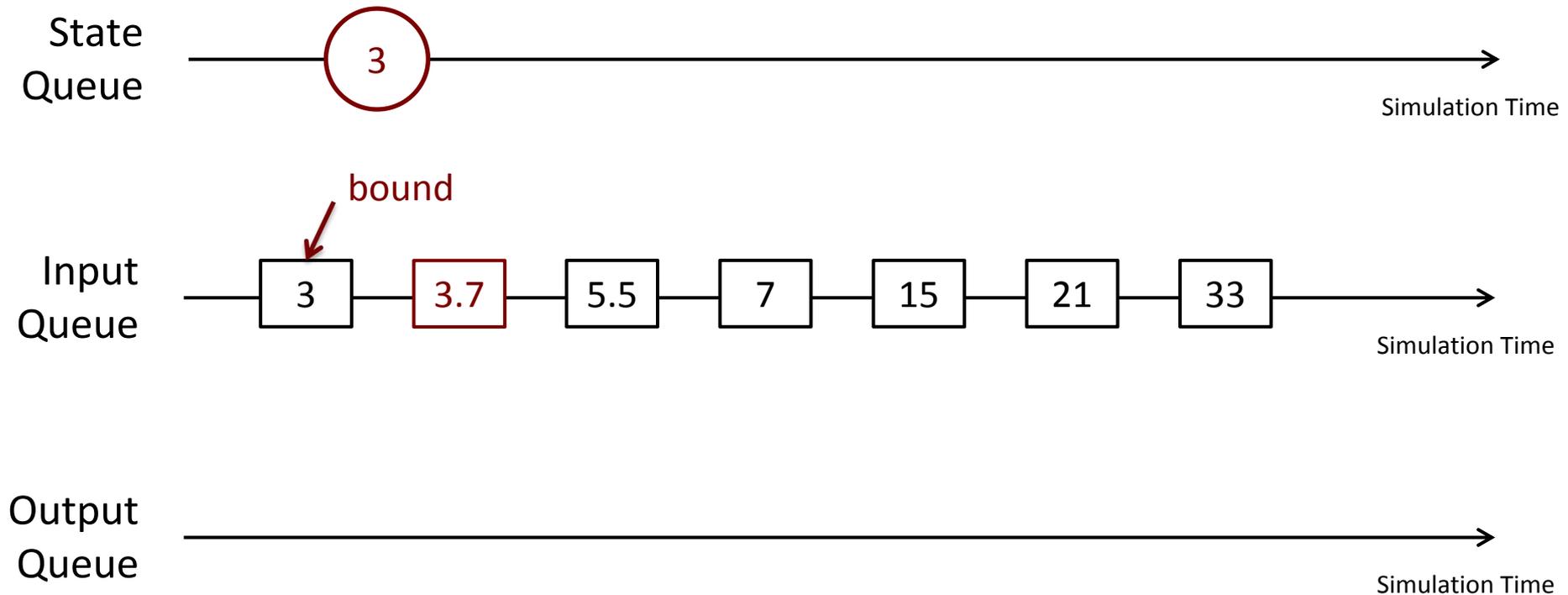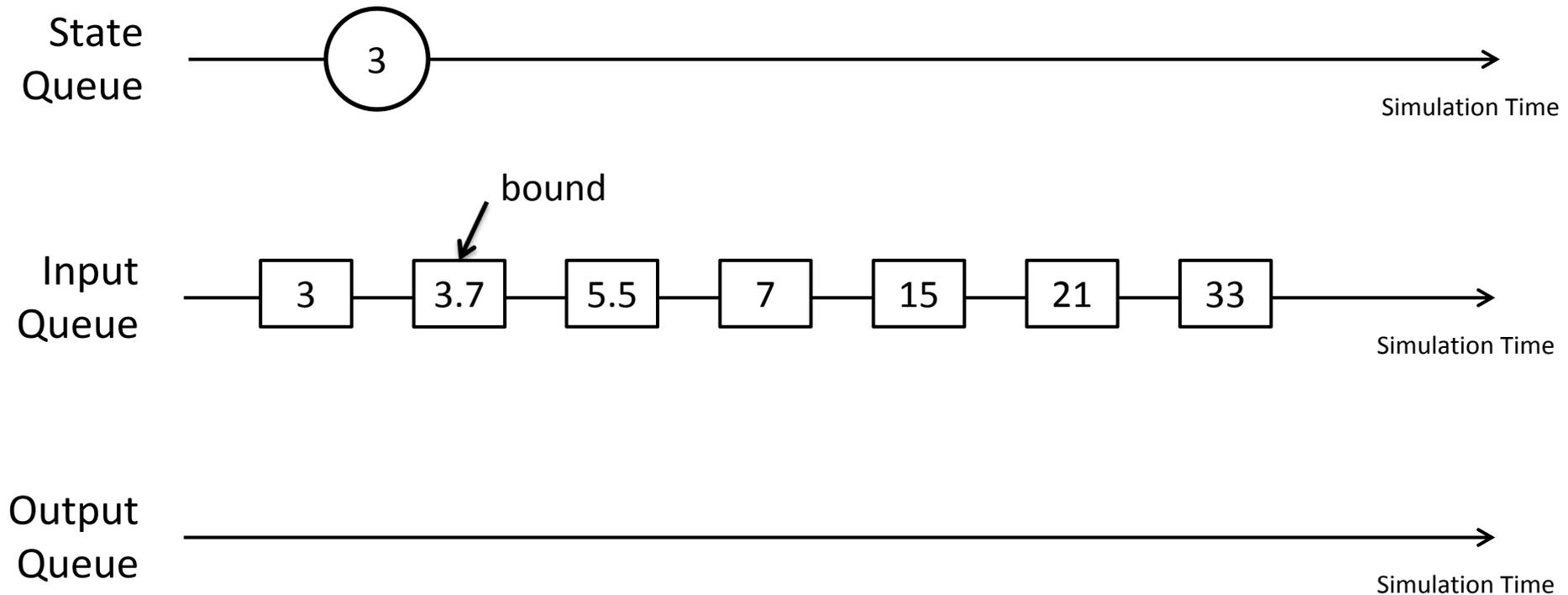# State Saving and Restore

State Saving and Restore

State Queue

3 — Simulation Time

bound

Input Queue

| 3 | 5.5 | 7 | 15 | 21 | 33 | — Simulation Time

Output Queue

| 3 | 3 | 3 | 3 | 7 | 7 | — Simulation Time

State Saving and Restore

State
Queue

3

Simulation Time

bound

Input
Queue

| 3 | 5.5 | 7 | 15 | 21 | 33 |

Simulation Time

Output
Queue

| 3 | 3 | 3 | 3 | 7 | 7 |

Simulation Time

Antimessages

State Saving and Restore

State Saving and Restore

State Saving and Restore

State Queue

3

Simulation Time

Input Queue

bound

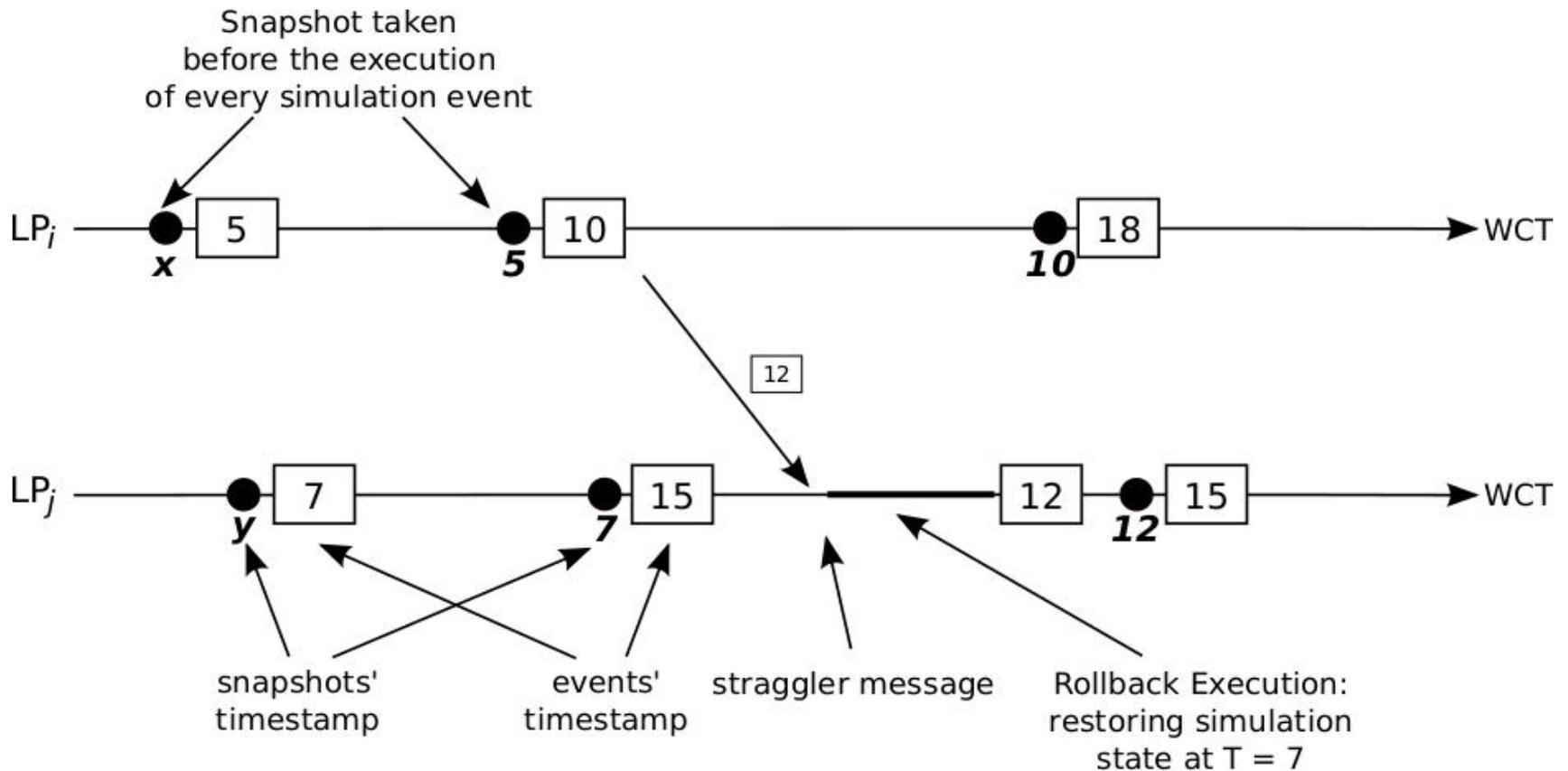| 3 | 3.7 | 5.5 | 7 | 15 | 21 | 33 |

Simulation Time

Output Queue
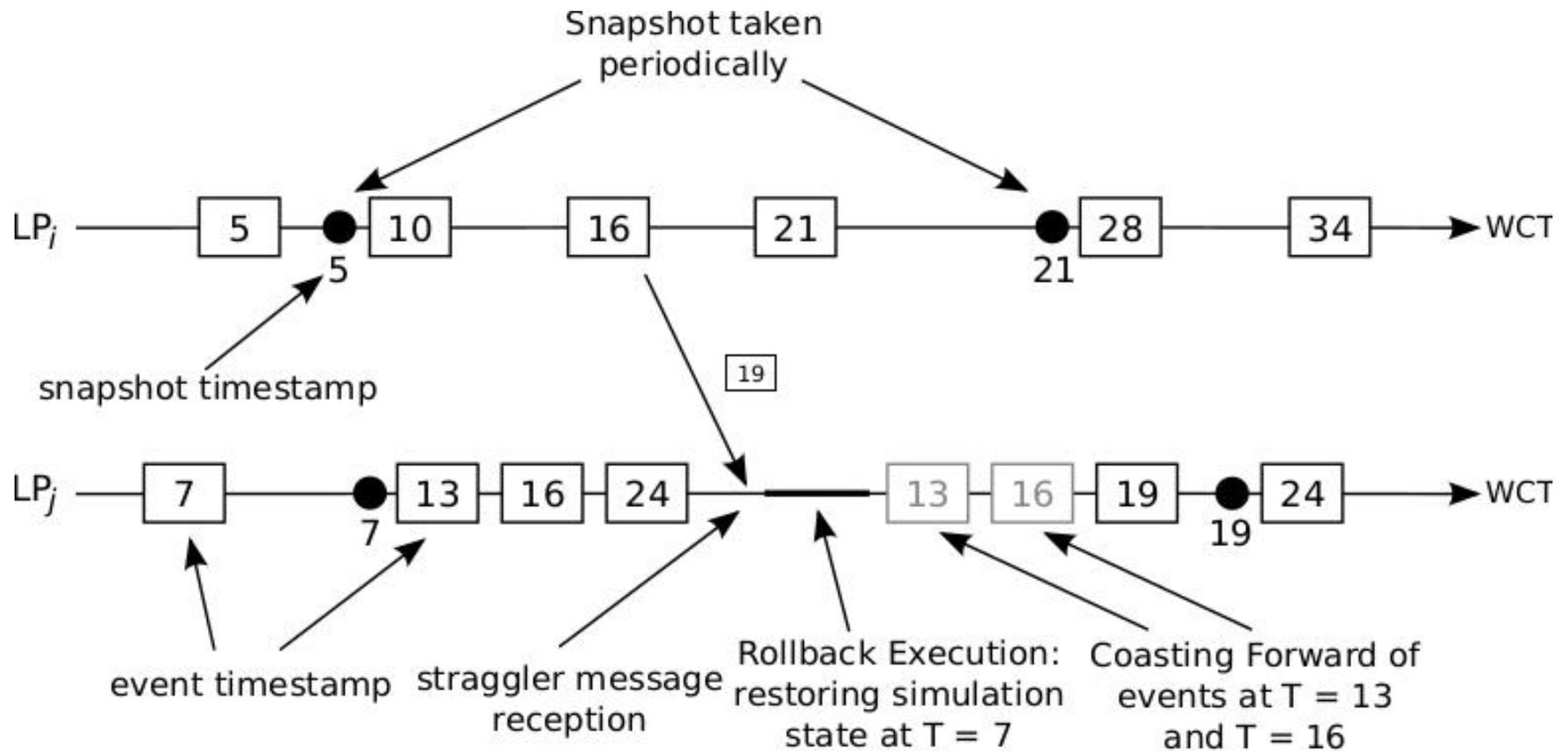
Simulation Time

State Saving Efficiency

- How large is the simulation state?
- How often do we execute a rollback? (*rollback frequency*)
- How many events do we have to undo on average?

- Can we do something better?

Copy State Saving

Sparse State Saving (SSS)

Coasting Forward

- Re-execution of already-processed events
- These events *have been artificially undone!*
- Antimessages have <u>not</u> been sent

- These events must be reprocessed in *silent execution*
  - Otherwise, we duplicate messages in the system!

When to take a checkpoint?

- Classical approach: *periodic state saving*

- Is this efficient?
  - Think in terms of memory footprint and wall-clock time requirements

When to take a checkpoint?

- Classical approach: *periodic state saving*
- Is this efficient?
  - Think in terms of memory footprint and wall-clock time requirements

- Model-based decision making
- This is the basis for *autonomic self-optimizing systems*
- Goal: find the best-suited value for χ

When to take a checkpoint?

$$\chi_{opt} = \left\lceil \sqrt{\frac{2\delta_s}{\delta_c} + \left(\frac{N}{k_r} + \gamma - 1\right)} \right\rceil$$

- δ_s: average time to take a snapshot
- δ_c: the average time to execute coasting forward
- N: total number of committed events
- k_r: number of executed rollbacks
- γ: average rollback length

Incremental State Saving (ISS)

- If the state is large and scarcely updated, ISS might provide a reduced memory footprint and a non-negligible performance increase!

- How to know what state portions have been modified?

Incremental State Saving (ISS)

- If the state is large and scarcely updated, ISS might provide a reduced memory footprint and a non-negligible performance increase!

- How to know what state portions have been modified?
  - Explicit API notification (non-transparent!)
  - Operator Overloading
  - Static Binary Instrumentation
  - Compiler-assisted Binary Generation

Reverse Computation

- It can reduce state saving overhead
- Each event is associated (manually or automatically) with a *reverse event*
- A majority of the operations that modify state variables are *constructive* in nature
  - the undo operation for them requires no history
- *Destructive* operations (assignment, bit-wise operations, …) can only be restored via traditional state saving

Reversible Operations

Type	Description	Application Code			Bit Requirements		
		Original	**Translated**	**Reverse**	**Self**	**Child**	**Total**
T0	simple choice	if() s1	if() {s1; b=1;}	if(b==1){inv(s1);}	1	x1,	1+
		else s2	else {s2; b=0;}	else{inv(s2);}		x2	max(x1,x2)
T1	compound choice	if () s1;	if() {s1; b=1;}	if(b==1) {inv(s1);}	lg(n)	x1,	lg(n) +
	(n-way)	elseif() s2;	elseif() {s2; b=2;}	elseif(b==2) {inv(s2);}		x2,	max(x1….xn)
		elseif() s3;	elseif() {s3; b=3;}	elseif(b==3) {inv(s3);}		….,	
		else() sn;	else {sn; b=n;}	else {inv(sn);}		xn	
T2	fixed iterations (n)	for(n)s;	for(n) s;	for(n) inv(s);	0	x	n*x
T3	variable iterations	while() s;	b=0;	for(b) inv(s);	lg(n)	x	lg(n) +n*x
	(maximum n)		while() {s; b++;}				
T4	function call	foo();	foo();	inv(foo)();	0	x	x
T5	constructive	v@ = w;	v@ = w;	v = @w;	0	0	0
	assignment						
T6	k-byte destructive	v = w;	{b =v; v = w;}	v = b;	8k	0	8k
	assignment						
T7	sequence	s1;	s1;	inv(sn);	0	x1+	x1+…+xn
		s2;	s2;	inv(s2);		….+	
		sn;	sn;	inv(s1);		xn	
T8	Nesting of T0-T7	Recursively apply the above			Recursively apply the above		

Non-Reversible Operations: if/then/else

```
if(qlen > 0) {
    qlen--;
    sent++;
}
```

```
if(qlen "was" > 0)
{
        sent--;
        qlen++;
}
```

- The reverse event must check an "old" state variables' value, which is not available when processing it!

Non-Reversible Operations: if/then/else

```
if(qlen > 0) {
    b = 1;
    qlen--;
    sent++;
}
```

```
if(b == 1) {
    sent--;
    qlen++;
}
```

- Forward events are modified by inserting "bit variables";
- The are additional state variables telling whether a particular branch was taken or not during the forward execution

Random Number Generators

- Fundamental support for stochastic simulation
- They must be <u>aware of the rollback operation</u>!
  - Failing to rollback a random sequence might lead to incorrect results (trajectory divergence)
  - Think for example to the coasting forward operation

- Computers are precise and deterministic:
  - Where does randomness come from?

Random Number Generators

- Practical computer "random" generators are common in use

- They are usually referred to as *pseudo-random generators*

- What is the correct definition of *randomness* in this context?

Random Number Generators

"The deterministic program that produces a random sequence should be different from, and—in all measurable respects—statistically uncorrelated with, the computer program that uses its output"

- Two different RNGs must produce statistically the same results when coupled to an application
- The above definition might seem circular: comparing one generator to another!
- There is a certain list of statistical tests

Uniform Deviates

- They are random numbers lying in a specified range (usually [0,1])

- Other random distributions are drawn from a uniform deviate
  - An essential building block for other distributions

- Usually, there are system-supplied RNGs:

```c
1  #include <stdlib.h>
2  #define RAND_MAX ...
3
4  void srand(unsigned seed);
5  int rand(void);
```

Problems with System-Supplied RNGs

- If you want a random float in [0.0, 1.0):

```
x = rand() / (RAND_MAX + 1.0);
```

- Be very (very!) suspicious of a system-supplied rand() that resembles the above-described one

- They belong to the category of *linear congruential generators*

$$I_{j+1} = a\, I_j + c \;(mod\; m)$$

- The recurrence will eventually repeat itself, with a period no greater than *m*

Problems with System-Supplied RNGs

- If m, a, and c are properly chosen, the period will be of maximal length (m)
  - all possible integers between 0 anbd m - 1 will occur at some point
- In general, it may look a good idea
- Many ANSI-C implementations are flawed

An example RNG (from libc)

```
1  unsigned long next = 1;
2
3  int rand(void) {
4      next = next * 1103515245 + 12345;
5      return (unsigned int)(next / 65536) % 32768;
6  }
7
8  void srand(unsigned int seed) {
9      next = seed;
10 }
```

An example RNG (from libc)

```
1  unsigned long next = 1;
2
3  int rand(void) {
4      next = next * 1103515245 + 12345;
5      return (unsigned int)(next / 65536) % 32768;
6  }
7
8  void srand(unsigned int seed) {
9      next = seed;
10 }
```

This is where we can support the rollback operation: consider the *seed* as part of the simulation state!

Problems with System-Supplied RNGs

Distributions of deviates over 10 million draws

Problems with System-Supplied RNGs
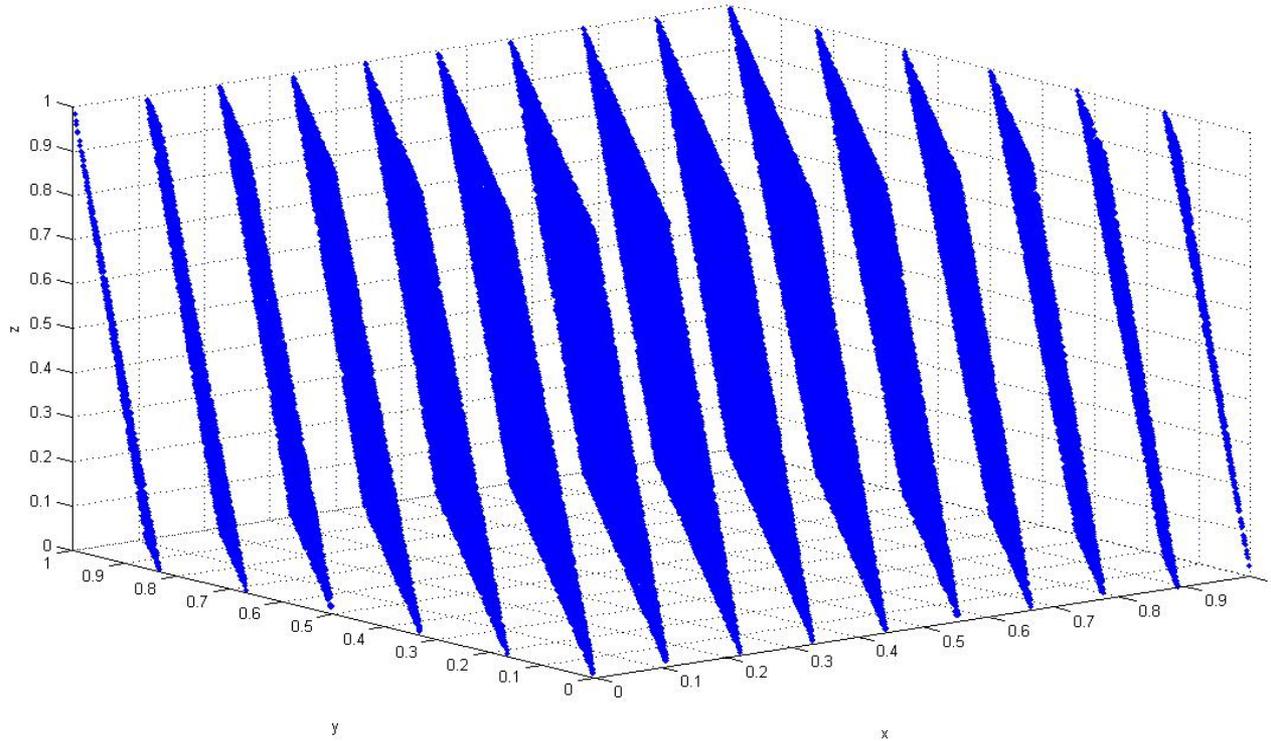
In an *n*-dimensional space, the points lie on
at most $m^{1/n}$ hyperplanes!

Functions of Uniform Deviates

- The probability *p(x)dx* of generating a number between *x* and *x+dx* is:

$$p(x)dx = \begin{cases} dx & 0 < x < 1 \\ 0 & \text{otherwise} \end{cases}$$

- *p(x)* is normalized:

$$\int_{-\infty}^{\infty} p(d)dx = 1$$

- If we take some function of *x* like *y(x)*:

$$|p(y)dy| = |p(x)dx| \quad \Rightarrow p(y) = p(x)\left|\frac{dx}{dy}\right|$$

Exponential Deviates

- Suppose that *y(x) ≡ -ln(x)*, and that *p(x)* is uniform:

$$p(y)dy = \left| \frac{dx}{dy} \right| dy = e^{-y} dy$$

- This is distributed exponentially

- Exponential distribution is fundamental in simulation
  - Poisson-random events, for example the radioactive decay of nuclei, or the more general interarrival time

Exponential Deviates

```c
1  #include <math.h>
2
3  float expent(long *idum) {
4      float dum;
5
6      do {
7          dum = rand(idum);
8      } while (dum == 0.0);
9
10     return -log(dum);
11 }
```

Deviate Transformation
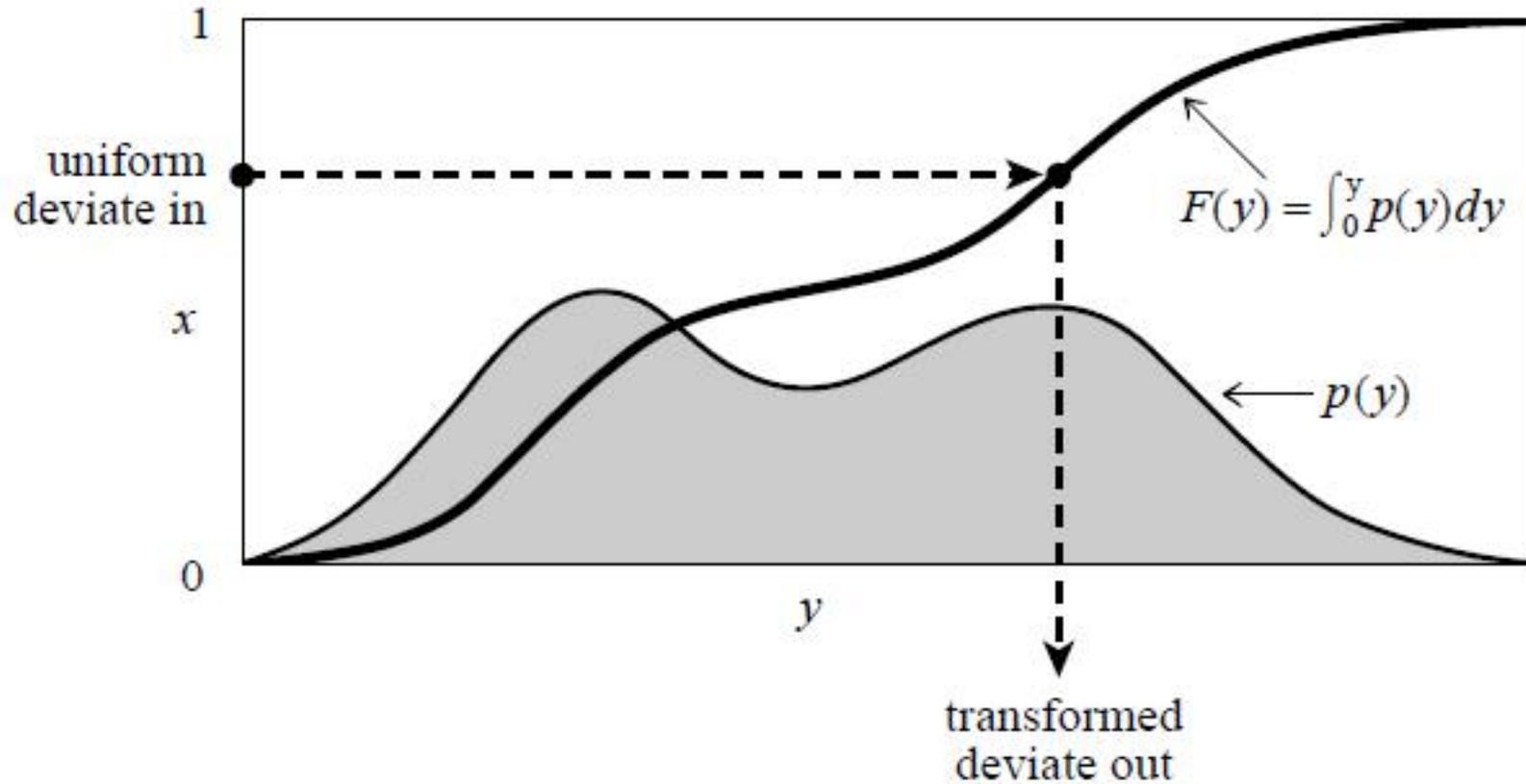
Scheduling Events
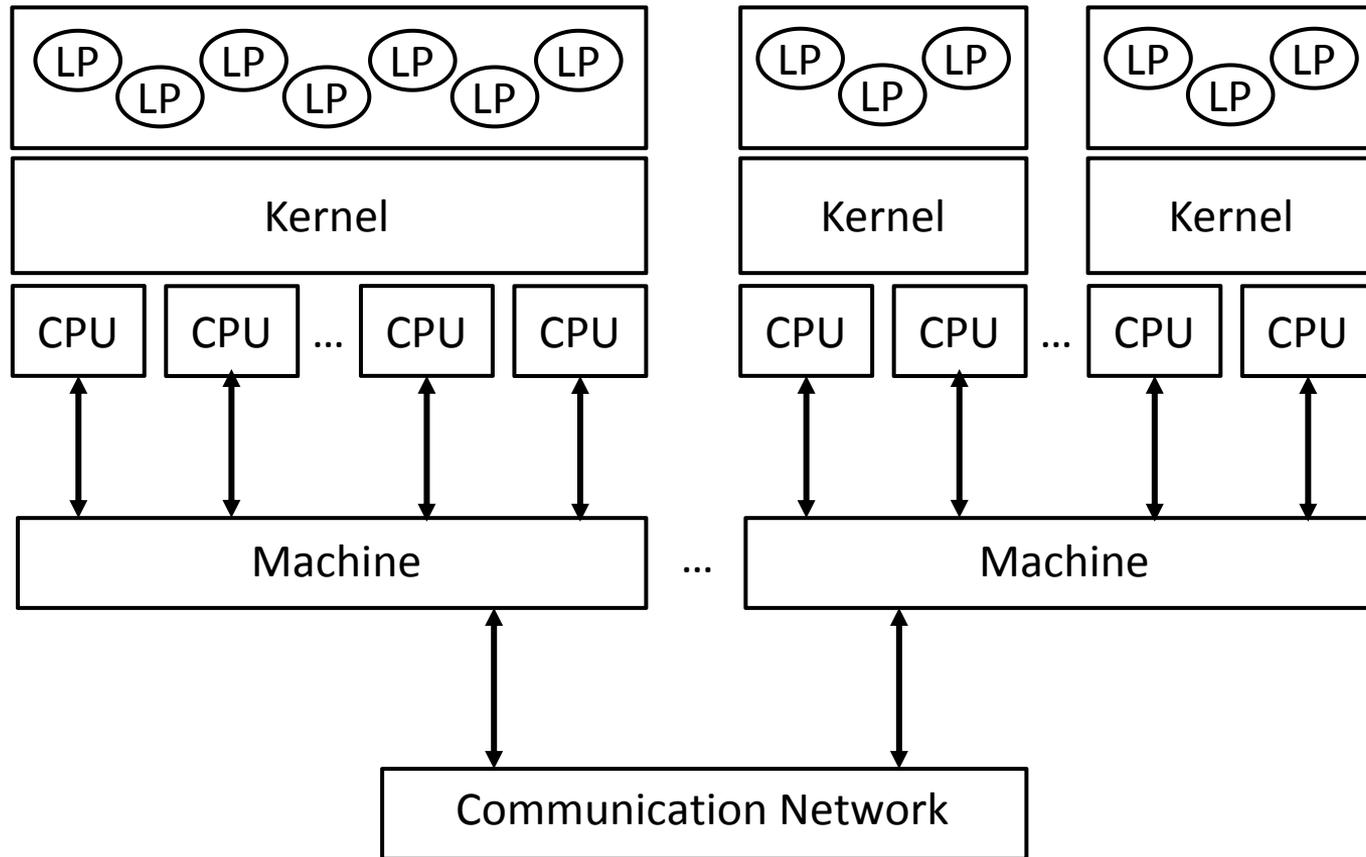
Scheduling Events

- A single thread takes care of a certain number of LPs at any time

- We have to avoid *inter-LPs rollbacks*

- **Lowest-Timestamp First**:
  - Scan the input queue of all LPs
  - Check the bound of each LP
  - Pick the LP whose next event is *closest in simulation time*

Global Virtual Time

- In a PDES system, memory usage is always increasing
  - We do not discard events
  - We take a lot of snapshots!

- We must find a way to implement a garbage collector
  - During the execution of an event at time T, we can schedule events at time $t \geq T$

Global Virtual Time

At a specific wall-clock time t*, the GVT is defined as the minimum between:*

- *All virtual times in all virtual clocks at time* t*;*

- *The timestamps of all sent but not yet processed events at time* t

Global Virtual Time
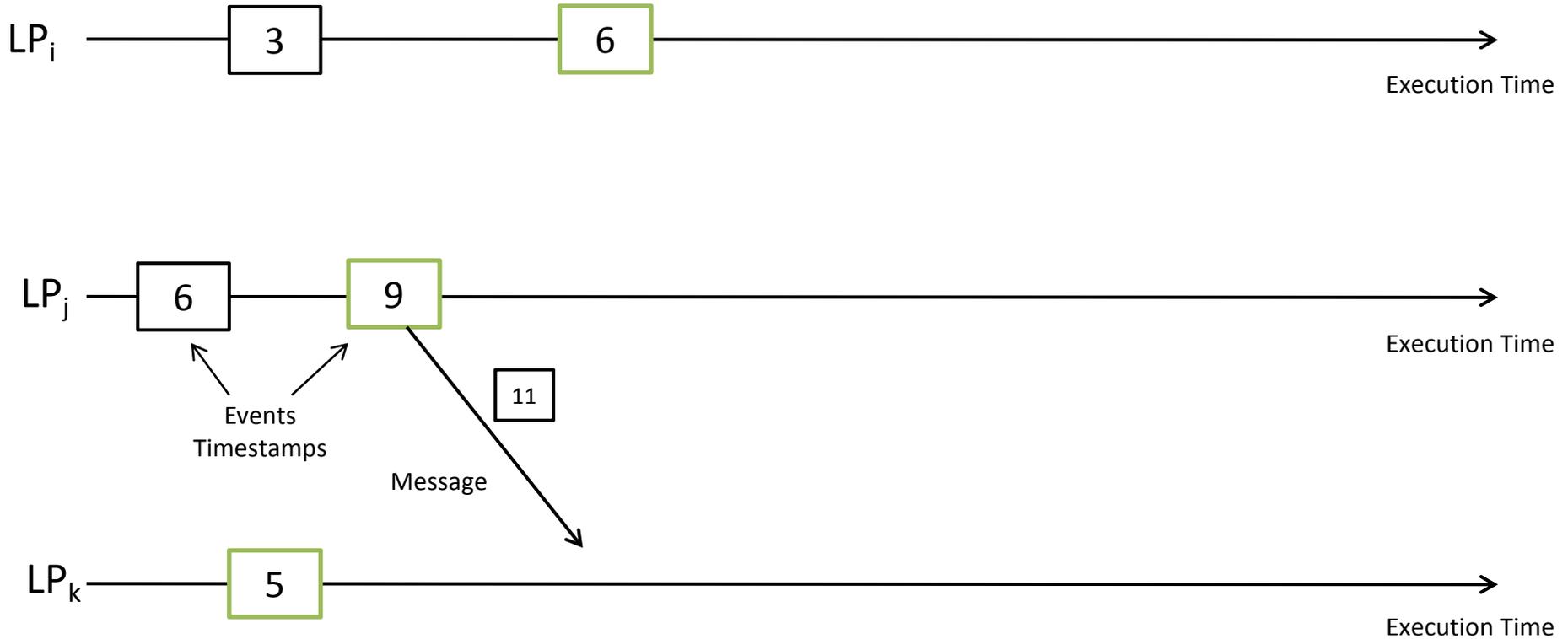
Global Virtual Time
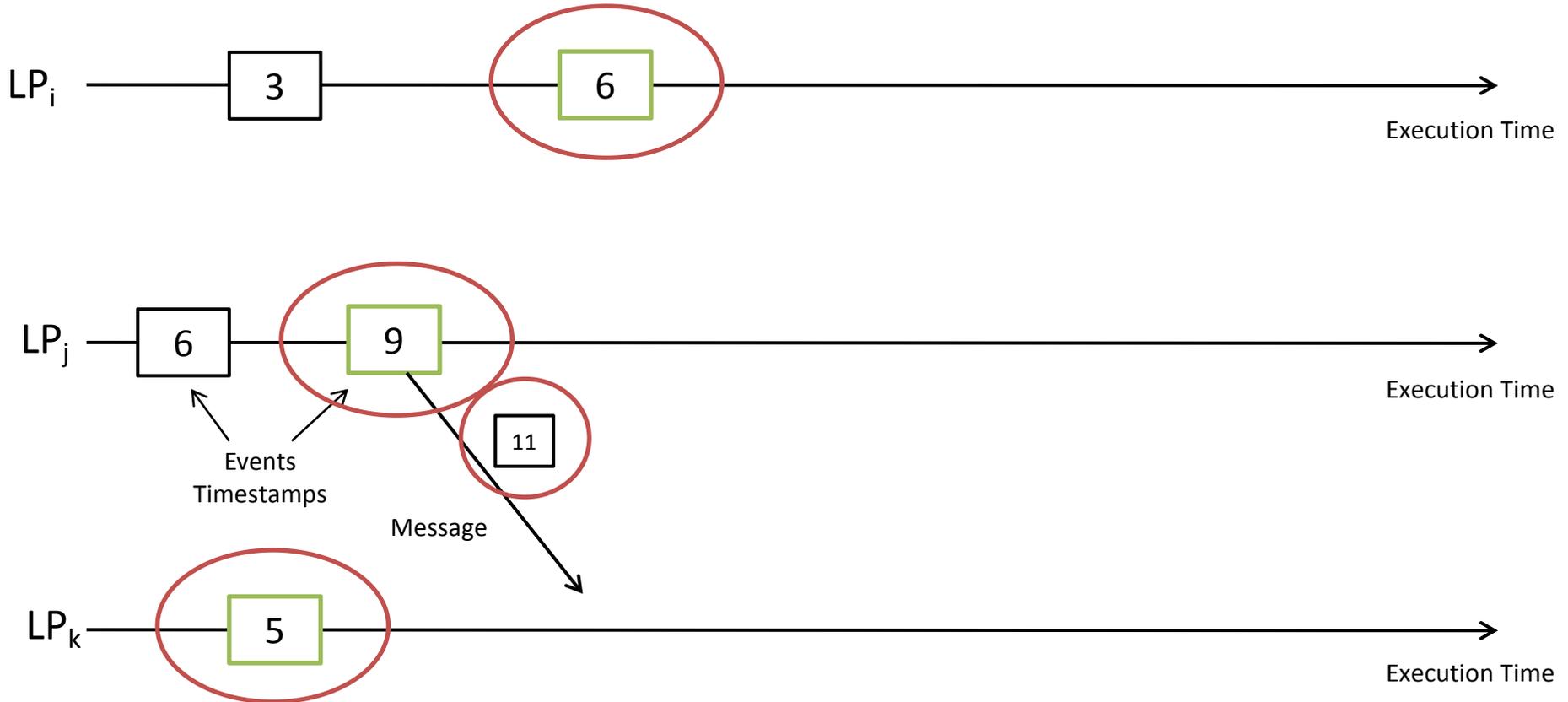
Global Virtual Time
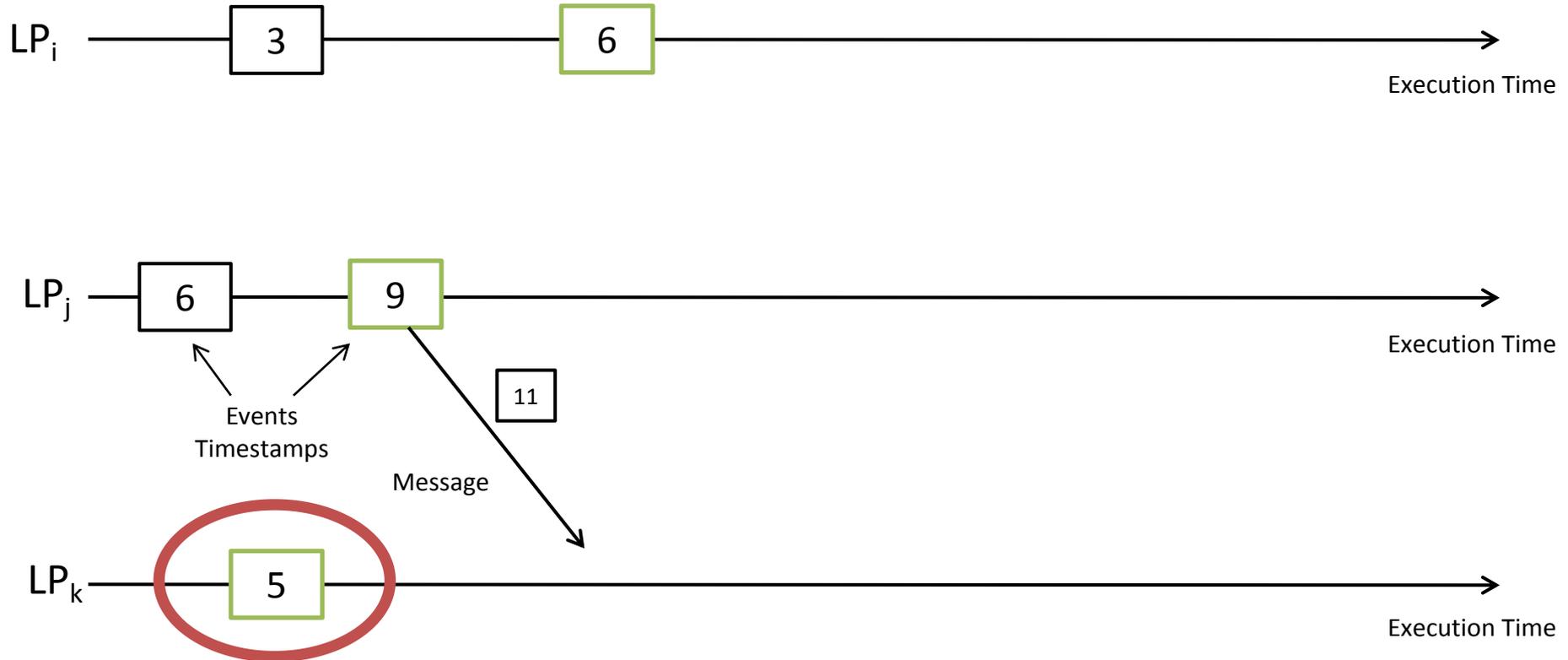
Global Virtual Time
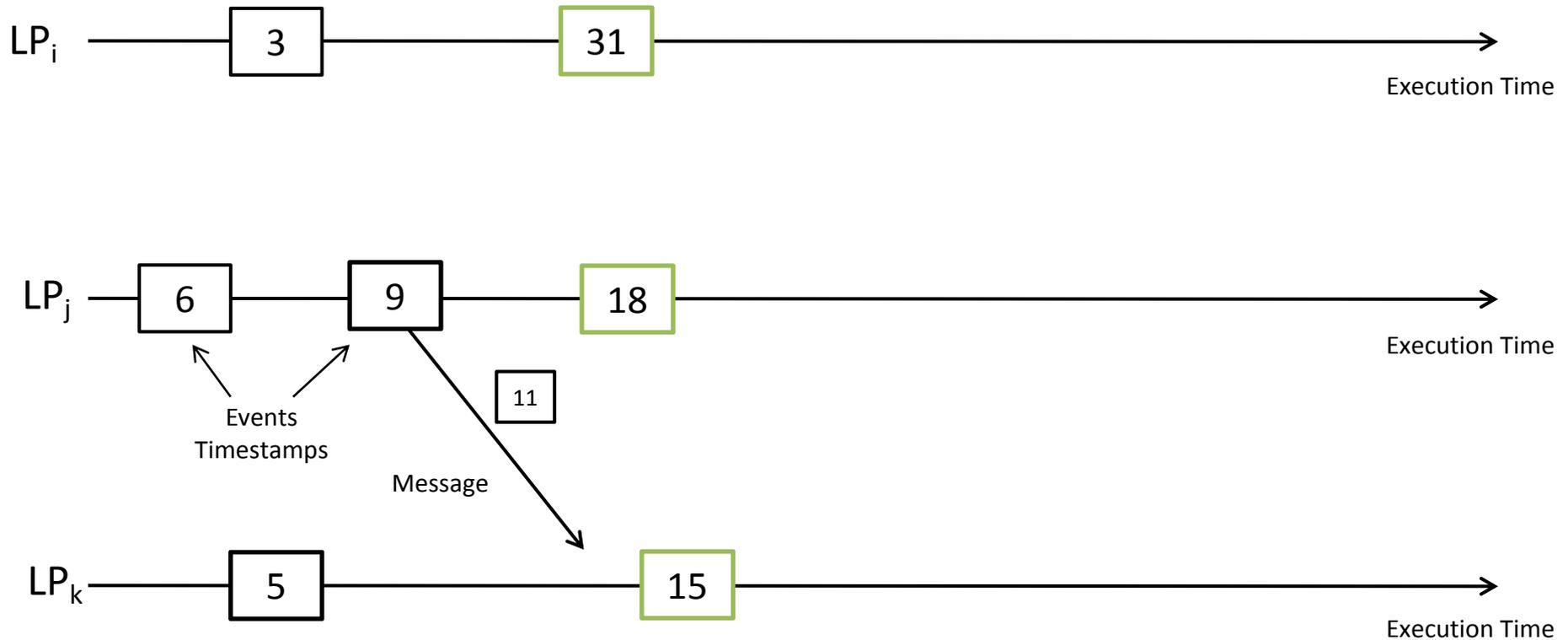
Global Virtual Time
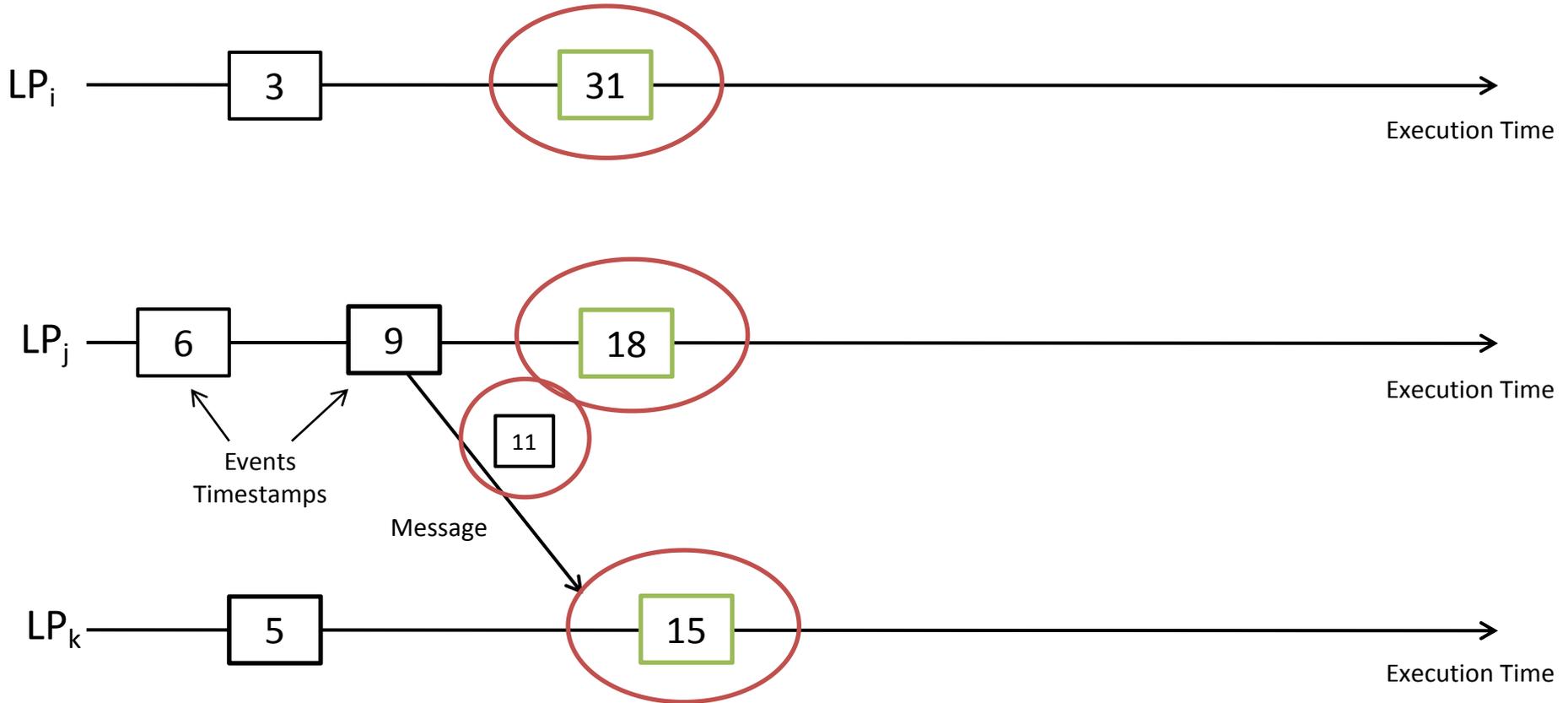
Global Virtual Time
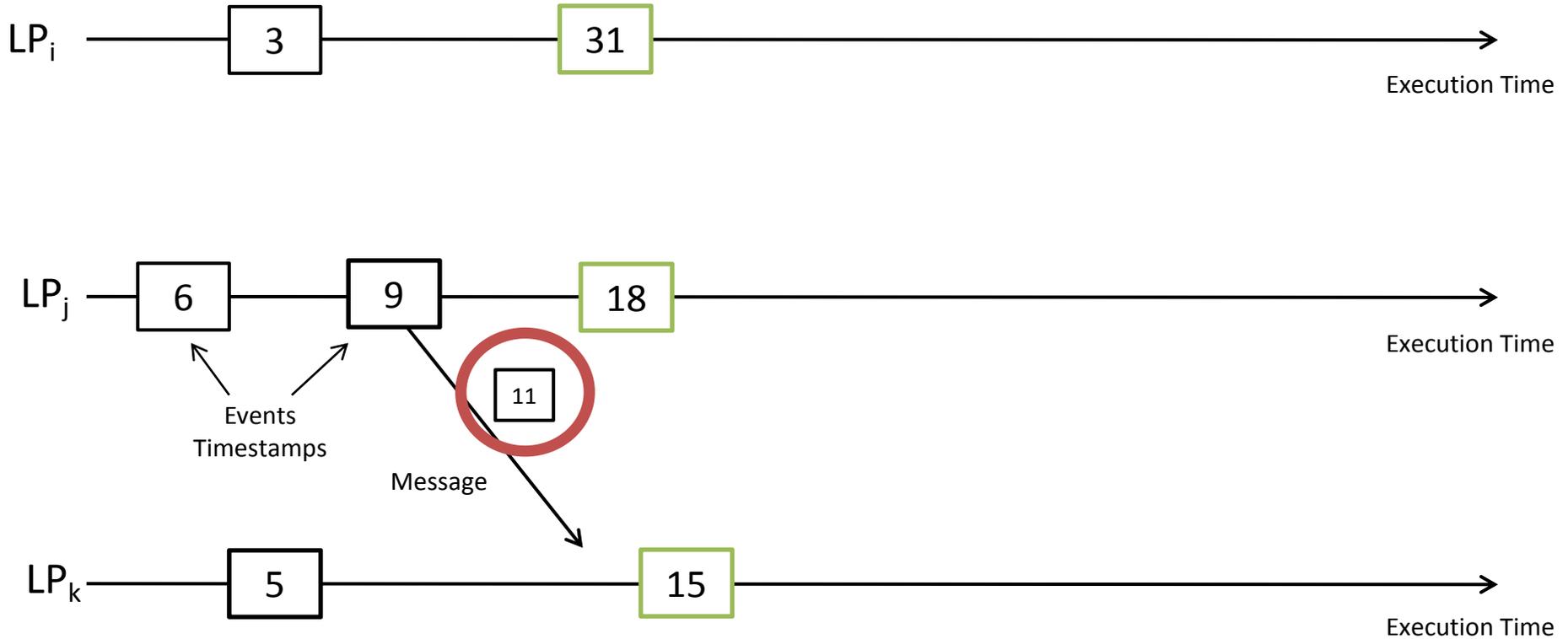
GVT Operations

- Once a correct GVT value is determined we can perform two actions:
  - **Fossil Collection**: the actual garbage collection of old memory buffers
  - **Termination Detection**

- GVT identifies the *commitment horizon* of the speculative execution

How Accurate is Speculative Simulation?

- Sequential Simulation is perfect for fine-grain inspection of predicates
  - It does not scale
  - Models are getting larger and larger everyday
- Parallel/Distributed simulation has great performance
- Fine-grain inspection is not viable
  - Process coordination is required
  - This hampers the achievable speedup
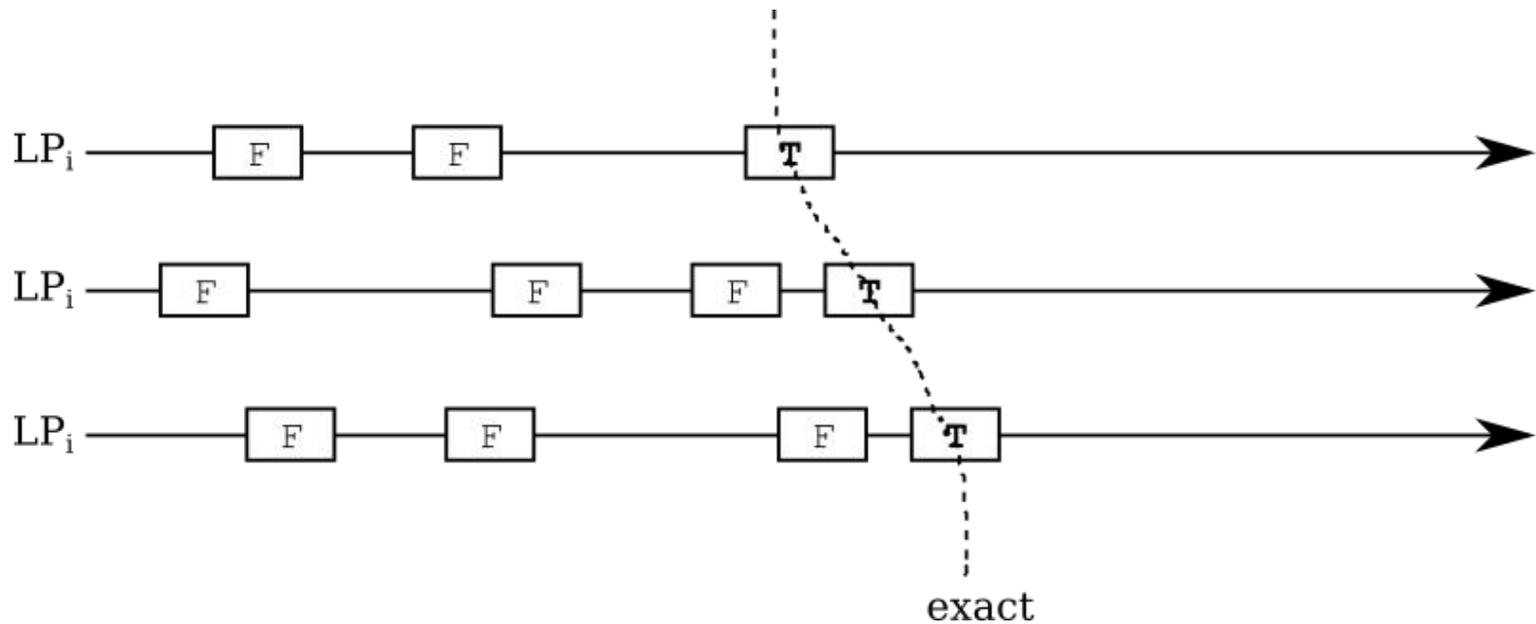
How Accurate is Speculative Simulation?

- Speculative Simulation inserts an additional delay

- The inspection of a global simulation state is delayed until a portion of the simulation trajectory becomes committed

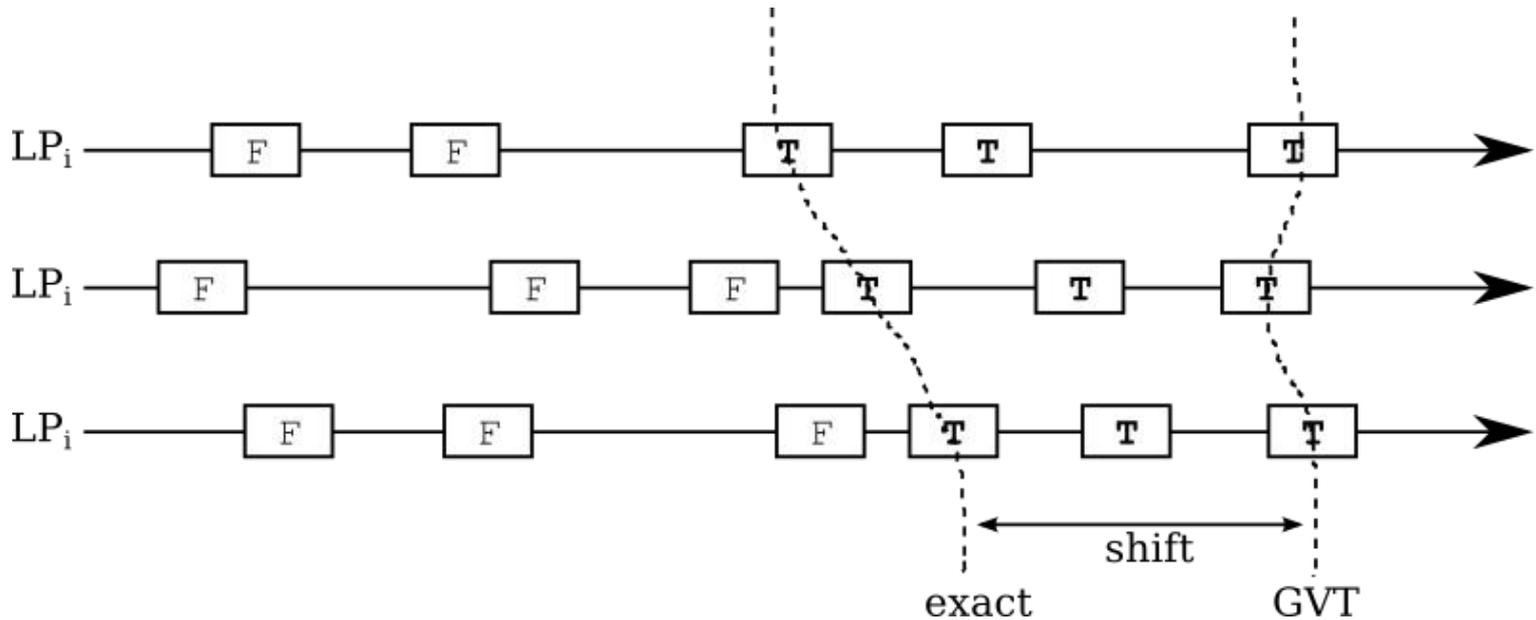- Inspection can be done after a GVT value has been computed
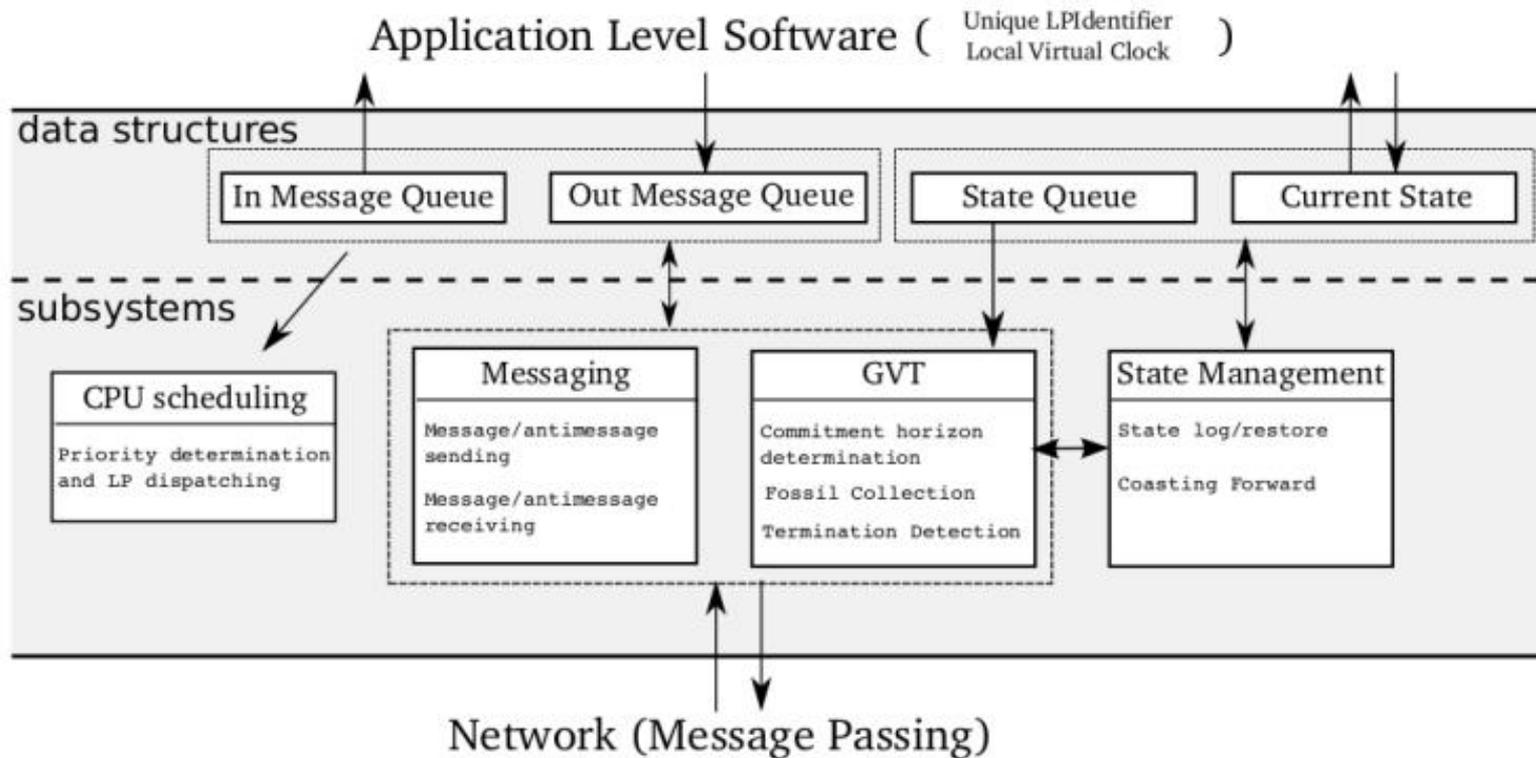
The Completion-Shift Problem
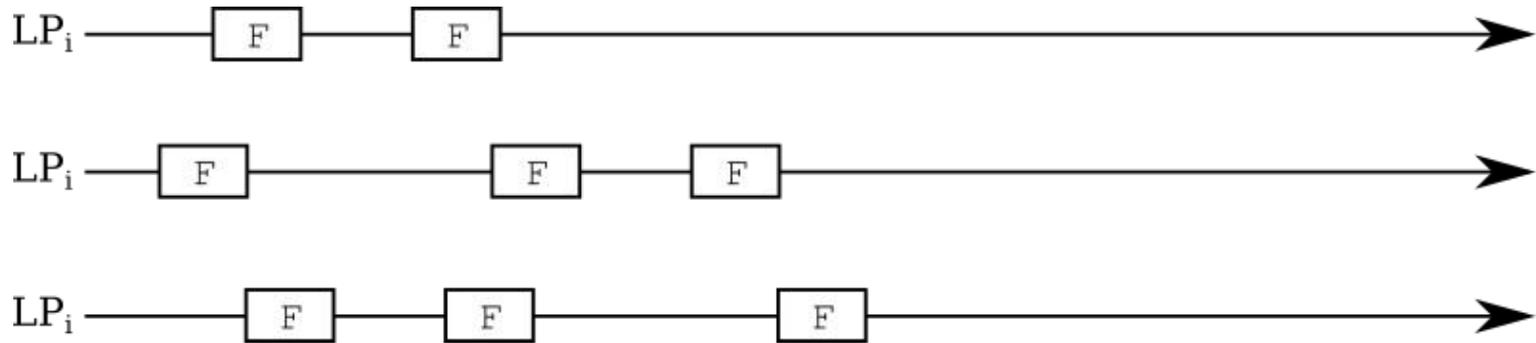
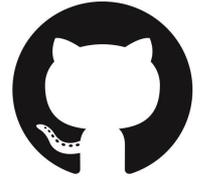# The Completion-Shift Problem

The Completion-Shift Problem

Time Warp Fundamentals

ROOT-Sim

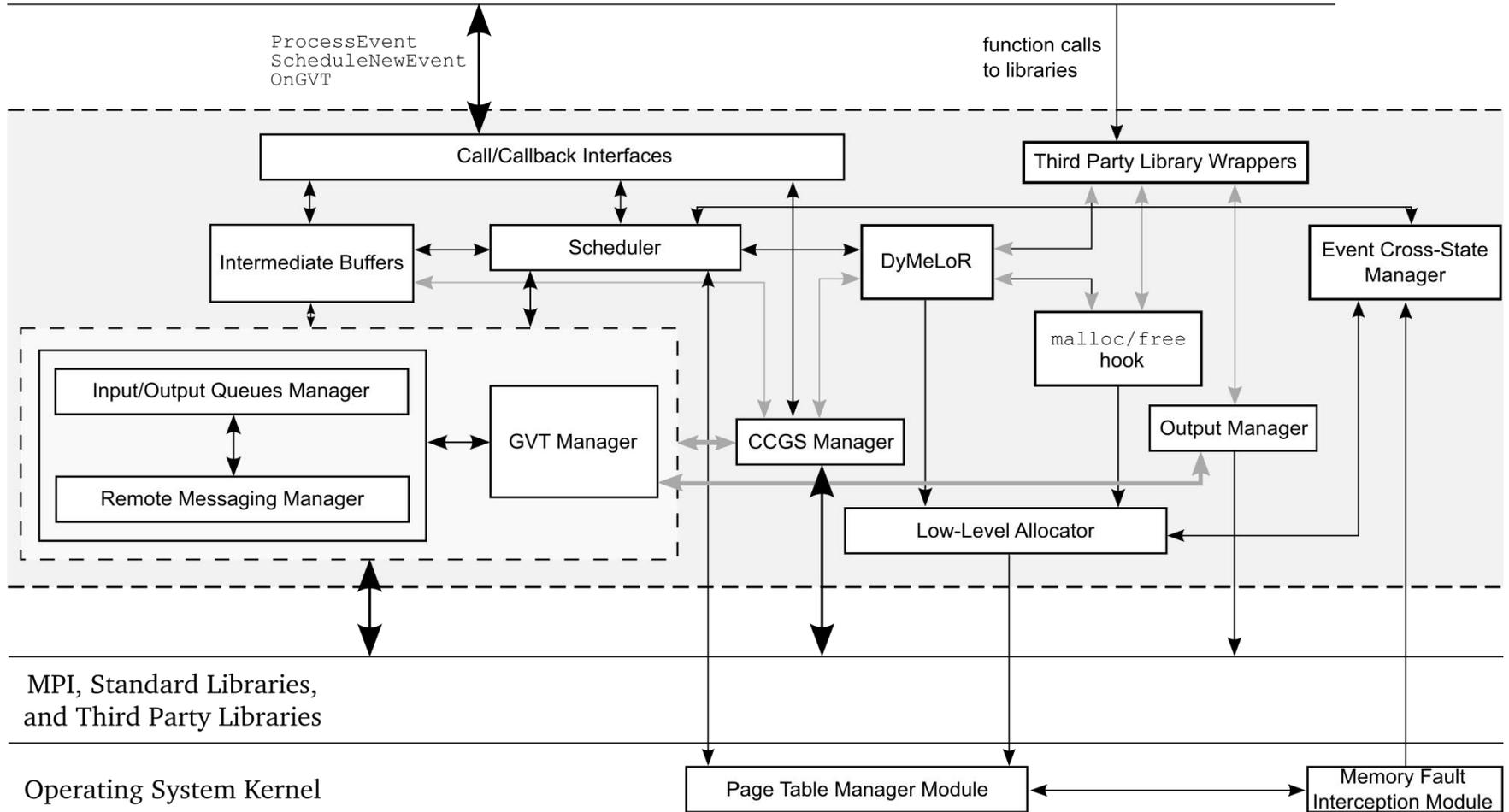- The ROme OpTimisti Simulator
  https://github.com/HPDCS/ROOT-Sim

- A general-purpose speculative simulation kernel based on both state saving and reversibility

- Targets complete transparency towards the model developer

- It can transparently deploy and run legacy models

ROOT-Sim Internals

EXAMPLE SESSION

PCS on ROOT-Sim