Distributed Simulation of Large Computer Systems

Moreno Marzolla

Univ. di Venezia “Ca’ Foscari” Dept. of Computer Science

and

INFN Padova

Email: marzolla@dsi.unive.it
Web: www.dsi.unive.it/~marzolla

Talk Outline

- Motivation
- Introduction to discrete-event simulation
- Conservative Distributed Simulation: Chandy-Misra protocol
- Distributed Simulation in C++
- Some experimental results
- Conclusions

Motivation

Large and *Complex* computer systems are increasingly common

- Large computing centers
- Clusters with hundreds (thousands?) or processors
- Computational Grids
- The global Internet

Performances of these systems should be understood (that is, evaluated *quantitatively*) to identify bottlenecks and other performance problems

Evaluating Computer System Performances

Many ways to do so. A mathematical model of the system is built and analyzed:

- *Exact* solutions can be obtained only for simple cases
- *Approximate* numerical solutions can be obtained often
- *Simulation* is the only viable approach when the system is too complex and/or can't be approximated

Sequential simulation of complex systems takes *time*

- The idea of Parallel and Distributed Simulation: *run the simulation on a parallel machine*

Discrete-Event System Simulation

Simulation models describe a (subset of a) *physical system* in terms of *states* and *events*.

Simulation is performed by repeating event's occurrences and updating the state accordingly.

In a *continuous* simulation it is assumed that the state evolves in a continuous way during time (eg: heat diffusion is a medium).

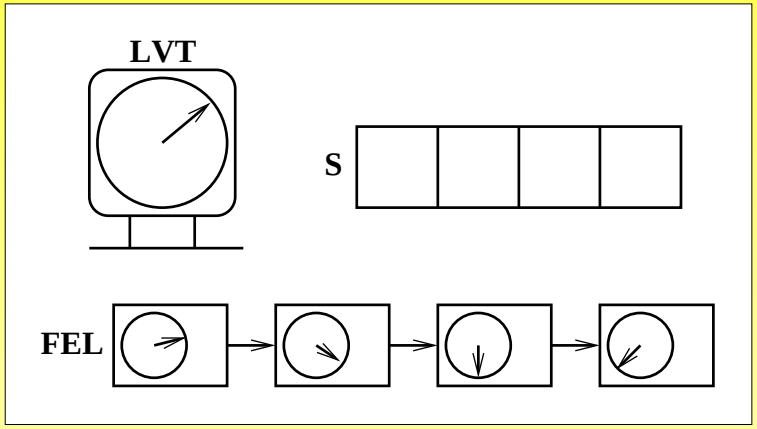
In a *discrete* simulation it is assumed that events happen at discrete time instants. (eg: the arrival of data packets in a communication network).

Discrete-Event System Simulators

A generic Discrete-Event Simulator (DES) is made of:

- A variable LVT (*local virtual time*)
- A *Future Event List* FEL, sorted by events' occurrence time.
- A *state vector* $S = (s_1, s_2, \ldots s_n)$, which represents the state of the system.

Discrete-Event Simulator Structure



Pseudocode of a generic Event-Driven Discrete Simulator:

while Finished **do**
  FirstEvent ← **Head**(FEL)
  FEL ← **Tail**(FEL)
  LVT ← **TimeOf**(FirstEvent)
  S ← **DoEvent**(FirstEvent)
end while

Sequential Simulation's Limitations

The structure of the generic discrete-event simulator is intrinsecally *sequential*; this is caused by the way the Future Event List is handled.

This implies that the sequential simulation algorithm cannot be easily parallelized.

Simulations of complex systems which are mathematically untractable require radically different approaches which should be able to run efficiently on parallel machines (cluster of workstations).

This requires that shared components of the sequential simulator (the *Clock* and *Future Event List*) should be eliminated.

Some problems...

Parallel and Discrete-Event Simulation (PDES) has been investigated since the late 70s[a]. However, it still is a difficult topic:

- Distributed simulators are hard to implement
- Performances are often application-dependent
- Synchronization problems

Anyway, it is believed to have a great potential. And as simulations are becoming increasingly complex and time-consuming, may even be the only viable approach.

---

[a] Peacock, J. K., Wong, J. W., and Manning, E. G. Distributed simulation using a network of processors. *Computer Networks*, 3, 1 (Feb. 1979), 44-56

Simulation using Logical Processes

Assumptions:

- The system composed of *Physical Processes* (*PP*) exchanging messages.
- The behaviour of a PP at time t cannot be influenced by messages it will receive at time $> t$.

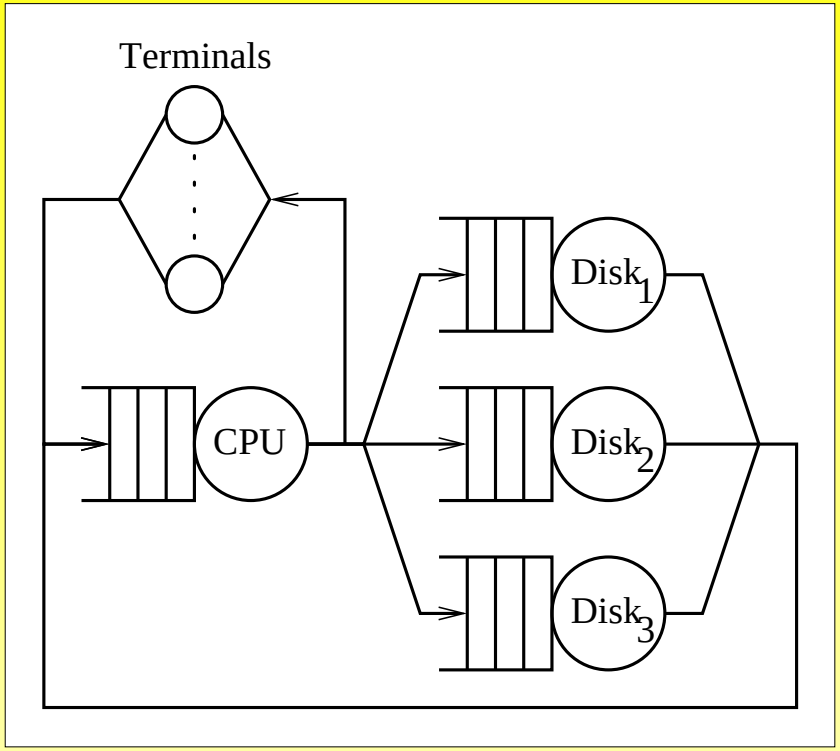
This is a *distributed system*. Each PP can be simulated by a corresponding *Logical Process* (*LP*) as follows:

- If PP_i sends a message m at time t to PP_j, then LP_i will eventually send a message $\langle t, m \rangle$ to LP_j.

We will consider *asynchronous* simulations, where logical processes do not share any global simulation clock.

Example: Central Server

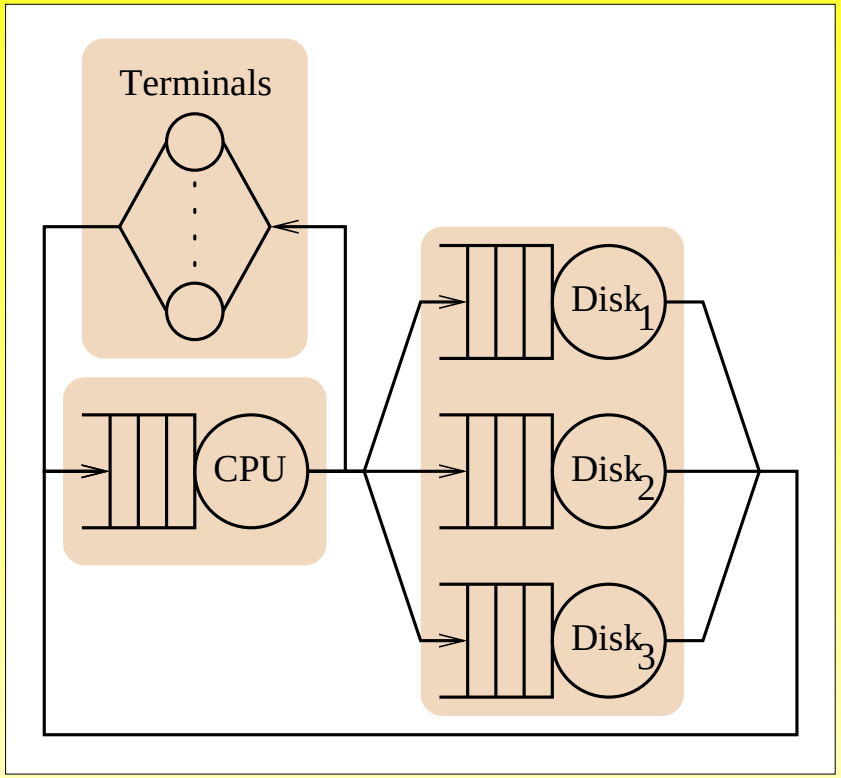
First introduced in 1973[a]. The CPU in the model is the “central server” which dispatches visits to other devices.

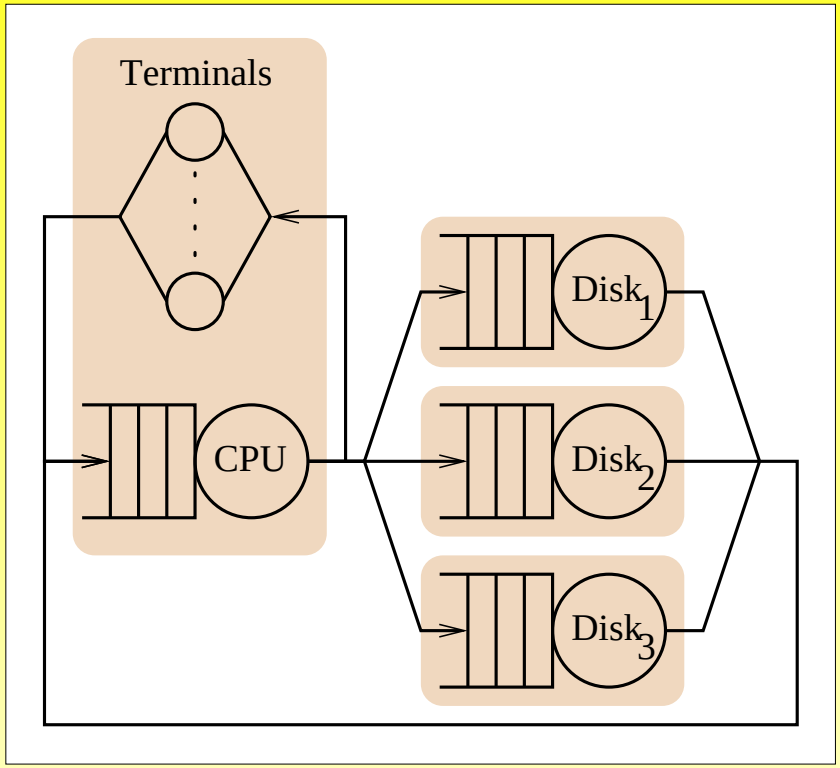


[a]Buzen, J. P., Computational Algorithms for Closed Queueing Networks with Exponential Servers, *Communications of the ACM*, **16**(9), 527–531

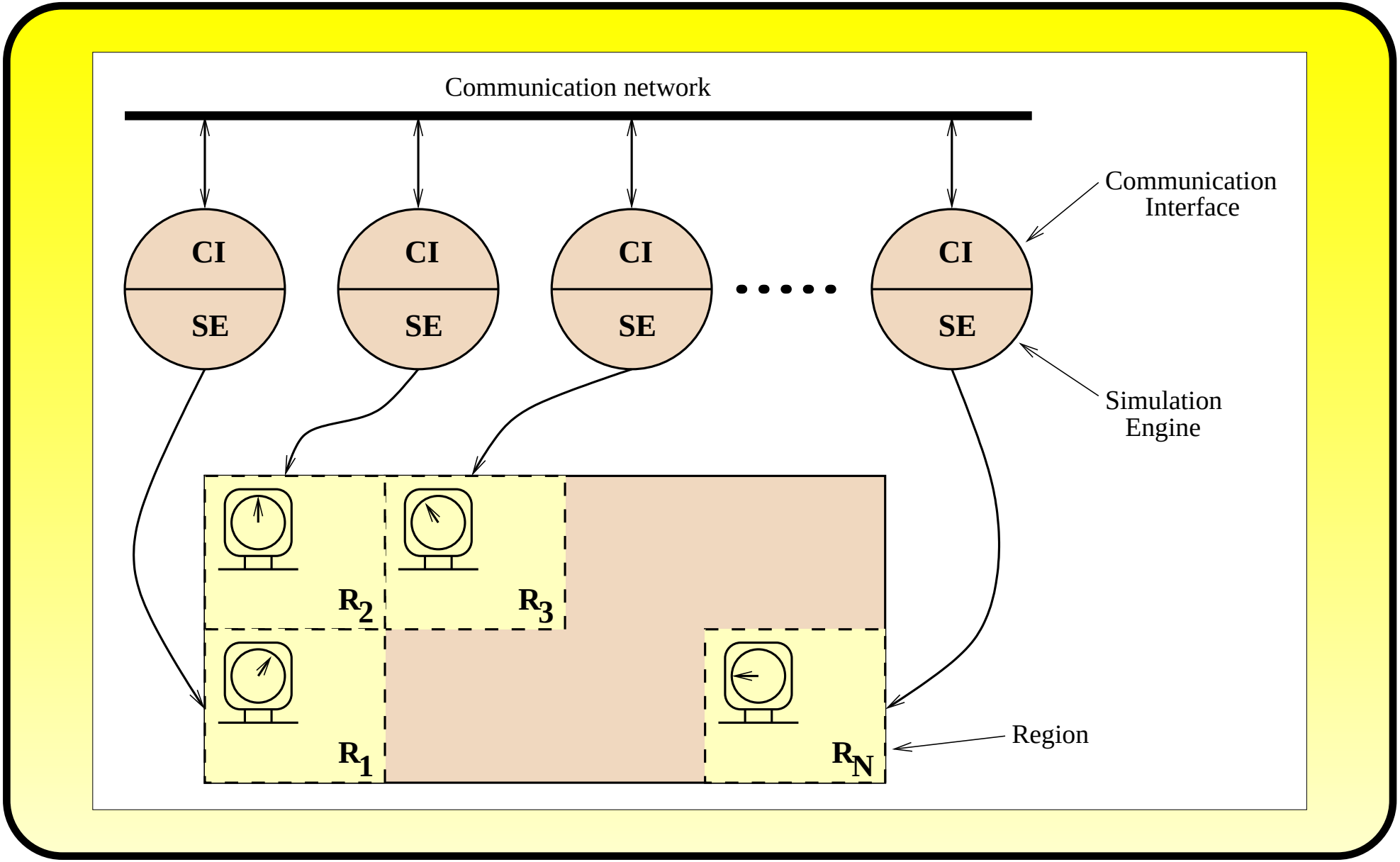
The “distributed” Central Server Simulator

Each node in the model is an LP; shaded regions are possible assignements of LPs to processors:





Structure of a Distributed Simulator



General Idea

Distributed Simulation in four easy steps:

- Each LP simulates its part of the system;
- Each LP has its own Local Virtual (Simulation) Time;
- Each LP can interact with other LPs only by exchanging messages;
- The distributed simulation should preserve *causal consistency*: if LP_i reaches simulated time LVT_i, it cannot process messages with timestamp $t > LVT_i$.

Ensuring Causal Consistency

There are two ways of ensuring causal consistency:

Conservative When an LP processes an event at time t, no other event with timestamp greater than t *will be received*. Causality violations cannot happen.

Optimistic (*Time-Warp*) Each LP processes events *as soon as they arrive*. Causality violations are recognized and resolved by *rolling back* the state of the simulator which processed the out-of-order event.

Conservative Simulation

Historycally it was the first paradigm introduced[a].

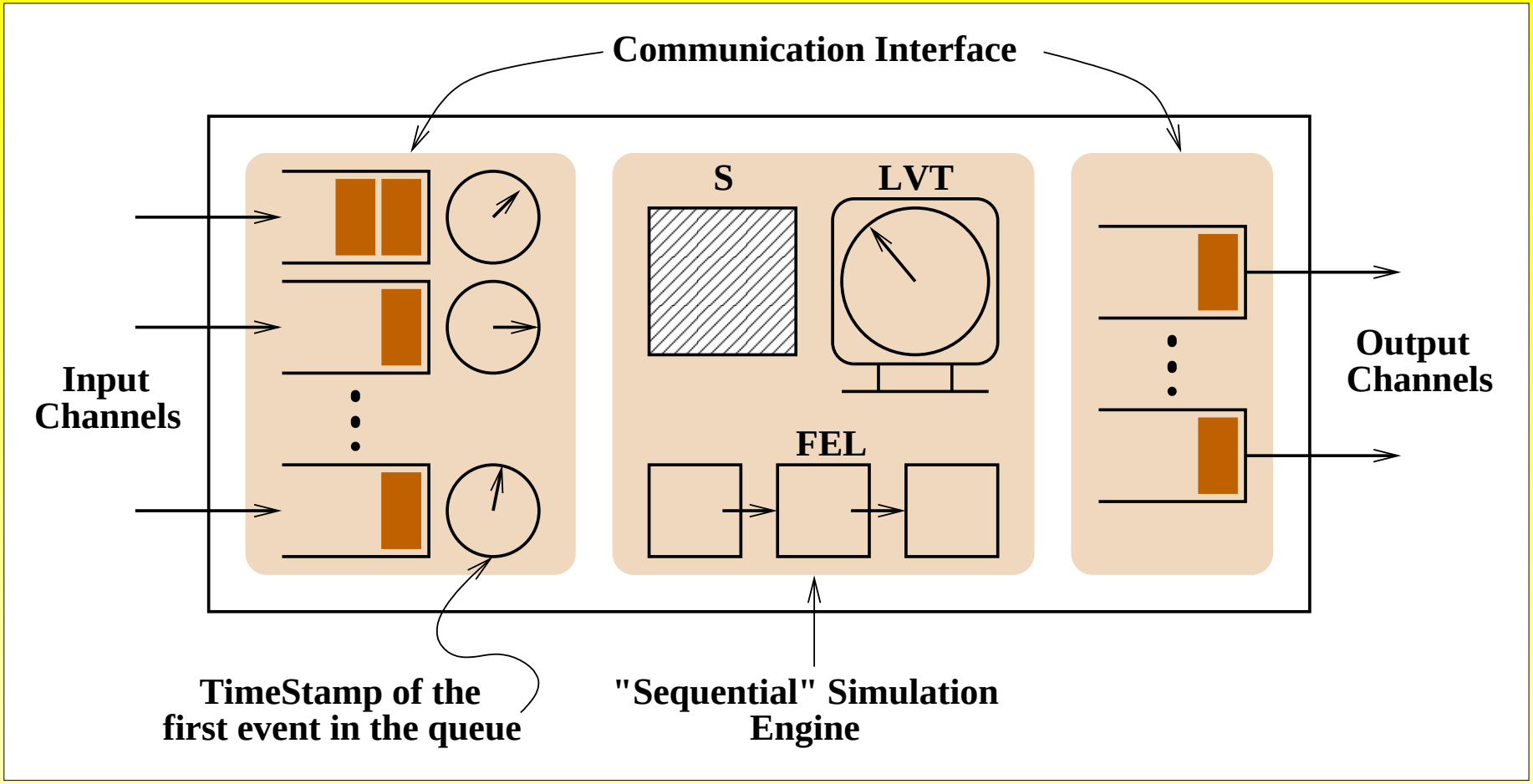
In this paradigm, causality violations are avoided by enforcing that no events should be processed out of order.

The Logical Process LP can process the event $e = \langle m, t \rangle$ iff there is no unreceived event $e' = \langle m', t' \rangle$ such that $t' < t$.

---

[a] Chandy, K. M. and Misra, J., *Asynchronous Distributed Simulation via a Sequence of Parallel Computations*, Communications of the ACM 24, 11(1981)

Structure of a Conservative Simulator



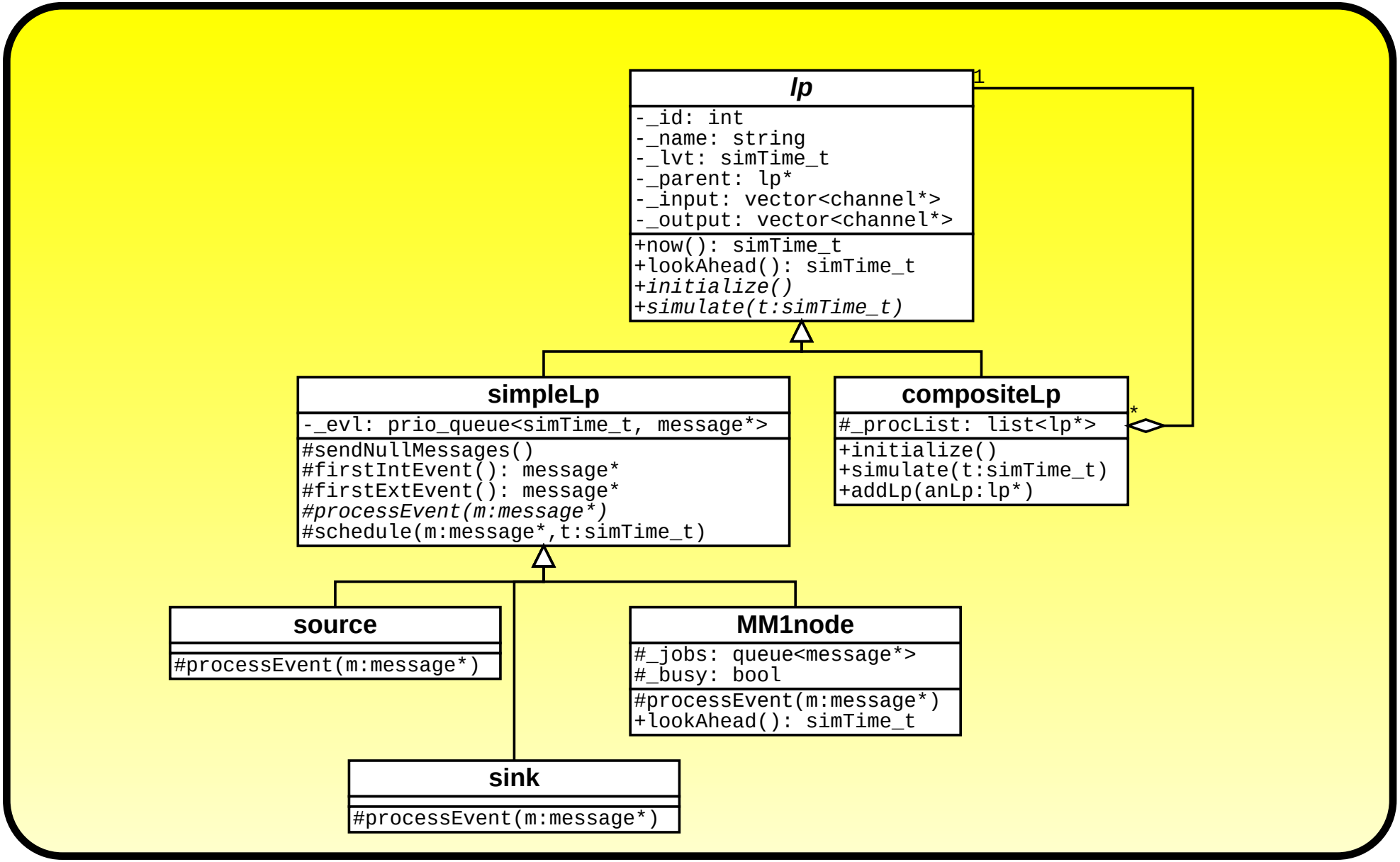
Distributed Simulation in C++

A *minimal* prototype of a distributed discrete-event simulator has been implemented.

Main features:

- Written in C++
- Uses a *conservative* synchronization protocol
- Implements *deadlock avoidance* using null messages and lookahead
- Based on the *MPI* library (Mpich version 1.2.1) for portable, message-based parallel programming.

UML diagram


lp
-_id: int
-_name: string
-_lvt: simTime_t
-_parent: lp*
-_input: vector<channel*>
-_output: vector<channel*>
+now(): simTime_t
+lookAhead(): simTime_t
+initialize()
+simulate(t:simTime_t)
1
*
simpleLp
-_evl: prio_queue<simTime_t, message*>
#sendNullMessages()
#firstIntEvent(): message*
#firstExtEvent(): message*
#processEvent(m:message*)
#schedule(m:message*,t:simTime_t)
compositeLp
#_procList: list<lp*>
+initialize()
+simulate(t:simTime_t)
+addLp(anLp:lp*)
source
#processEvent(m:message*)
MM1node
#_jobs: queue<message*>
#_busy: bool
#processEvent(m:message*)
+lookAhead(): simTime_t
sink
#processEvent(m:message*)

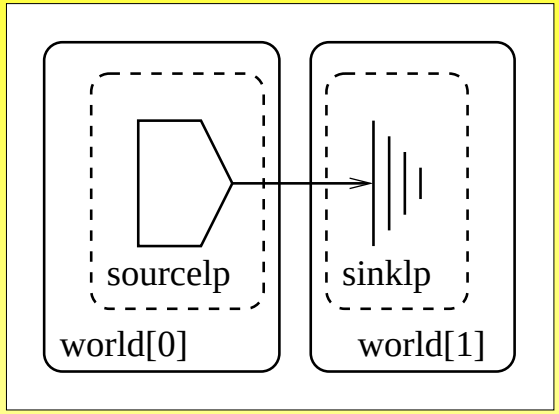
Code Example

```
#include <vector>
#include "compositeLp.hh"
#include "source.hh"
#include "sink.hh"
#include "channelFactory.hh"
#include "mpi++.h"

int main( int argc, char *argv[] )
{
  int myid;
  channelFactory* theFactory = new channelFactory( );
  vector<compositeLp*> world(2);
  world[0] = new compositeLp( "World0", 0, 0 );
  world[1] = new compositeLp( "World1", 0, 0 );
  lp* sourcelp  = new source ( "Source", new constantGen(1) );
  lp* sinklp    = new sink   ( "Sink" );
  world[0]->addLp( sourcelp );
  world[1]->addLp( sinklp   );
  theFactory->connect(sourcelp, 0, sinklp, 0);
  MPI::Init(argc,argv);
  myid=MPI::COMM_WORLD.Get_rank( );

  world[myid]->initialize( );
  while ( world[myid]->now( ) < 400 )
    world[myid]->simulate( 400 );

  MPI::Finalize( );
}
```



First experiment: Central Server model

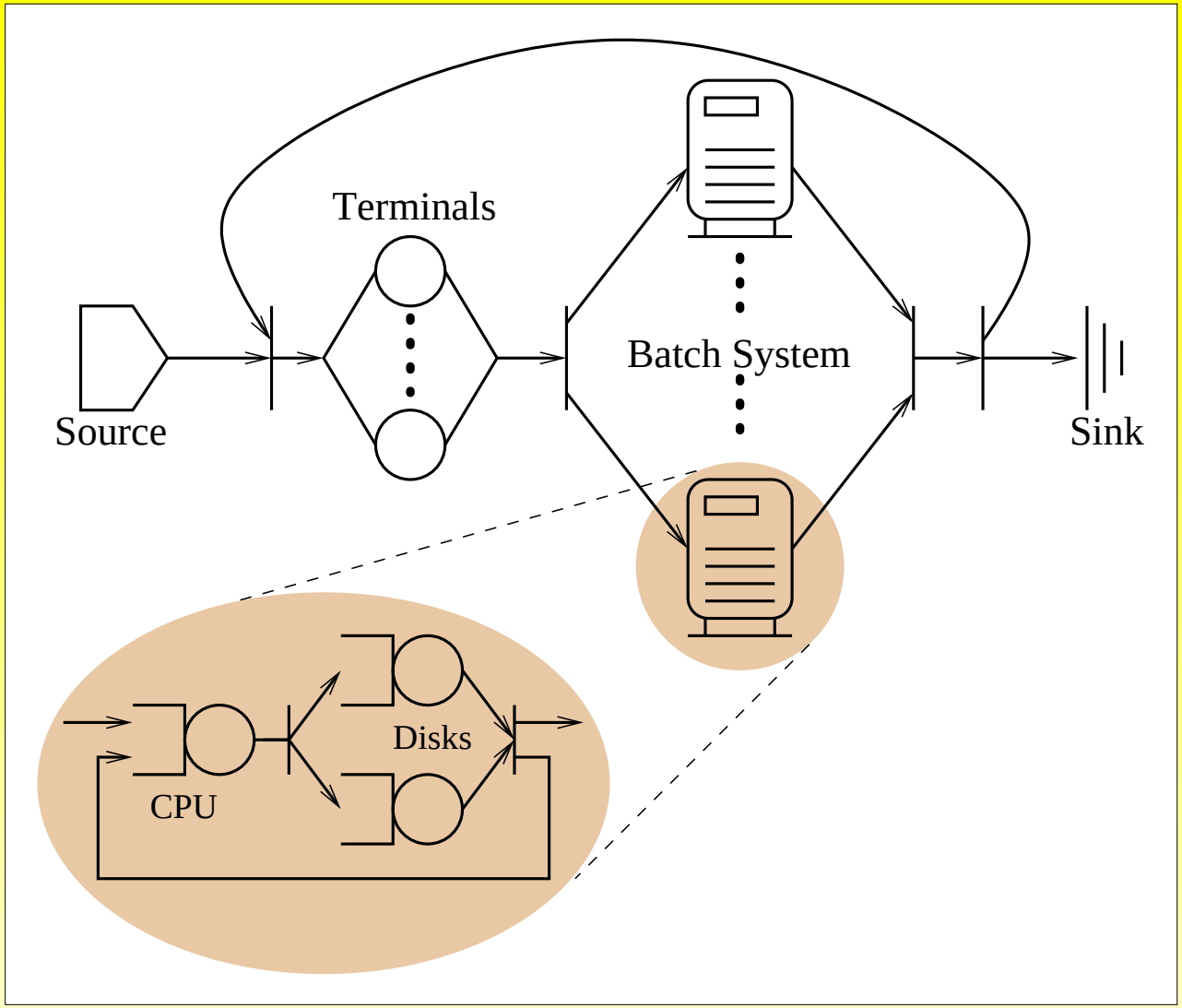
Consider a system in which users submit batch jobs to a batch system.

- The batch system is composed of N workstations.
- User creates a new job at a fixed rate.
- Jobs are sent to a workstation chosen at random.
- After completion, the job may leave the system, or may return to the user for further editing.

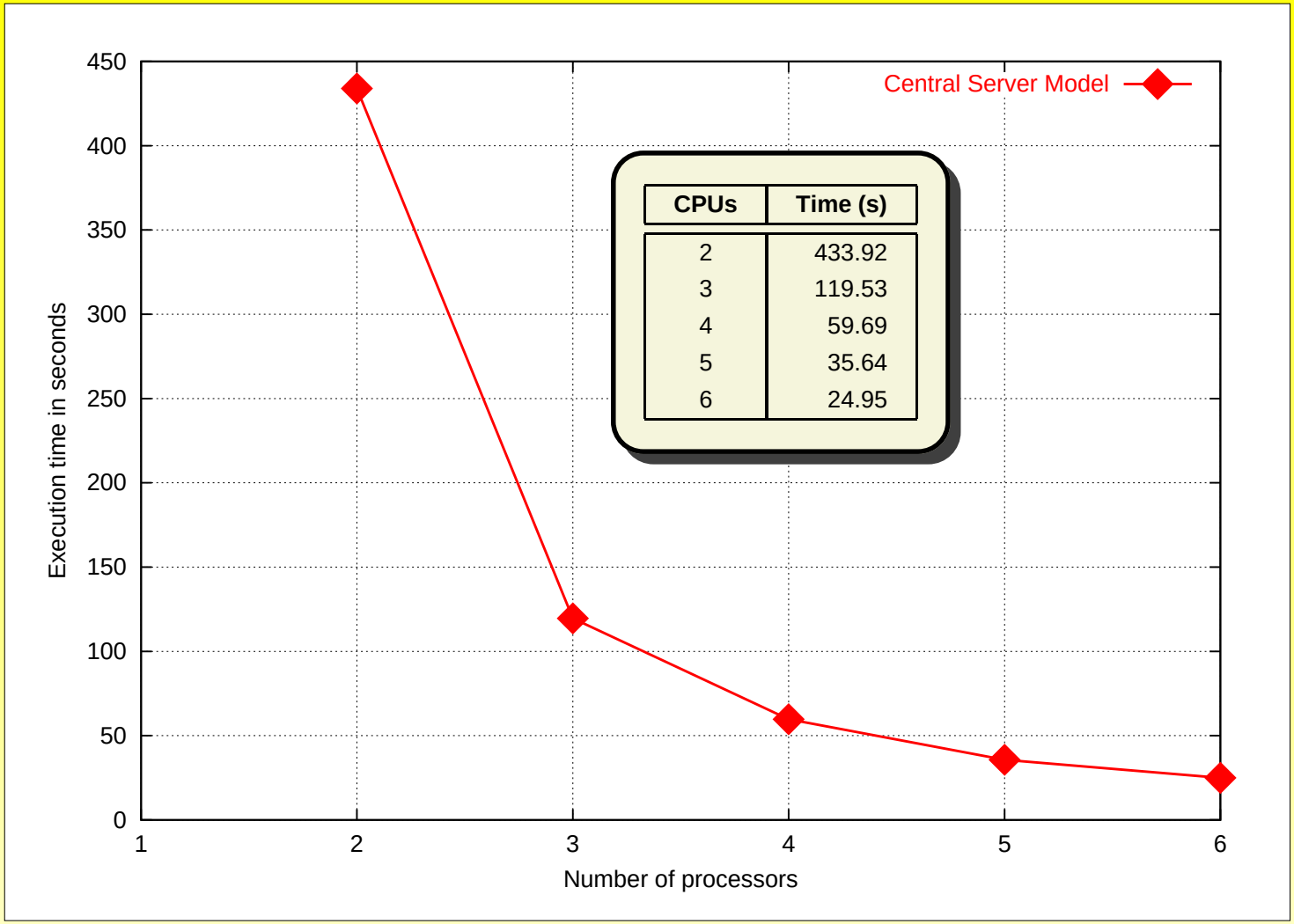
The system has been simulated on k processors by subdividing the N workstations to $k-1$ processors. The remaining processor simulates the terminal, source and sink nodes.

We simulated a system with $N = 60$ workstations, up to time $T = 2000$.

First experiment: Central Server model



Central Server results

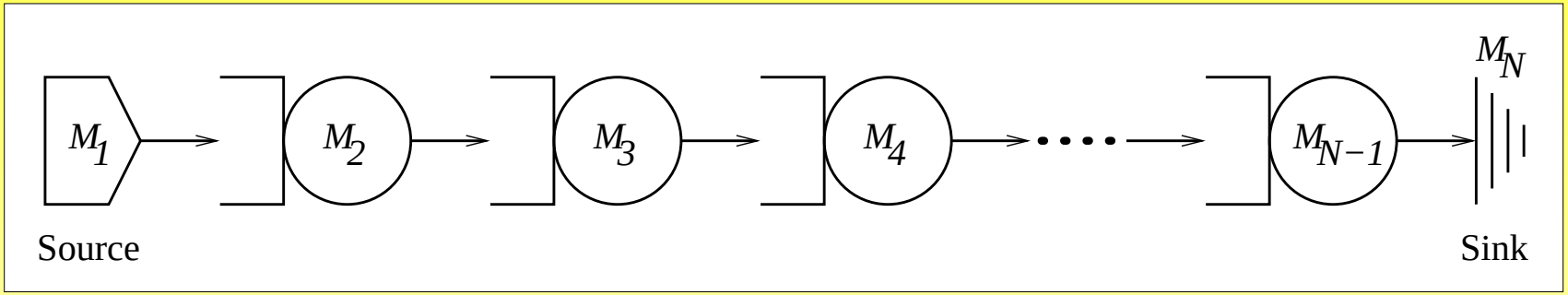


CPUs	Time (s)
2	433.92
3	119.53
4	59.69
5	35.64
6	24.95

Second experiment: Pipeline

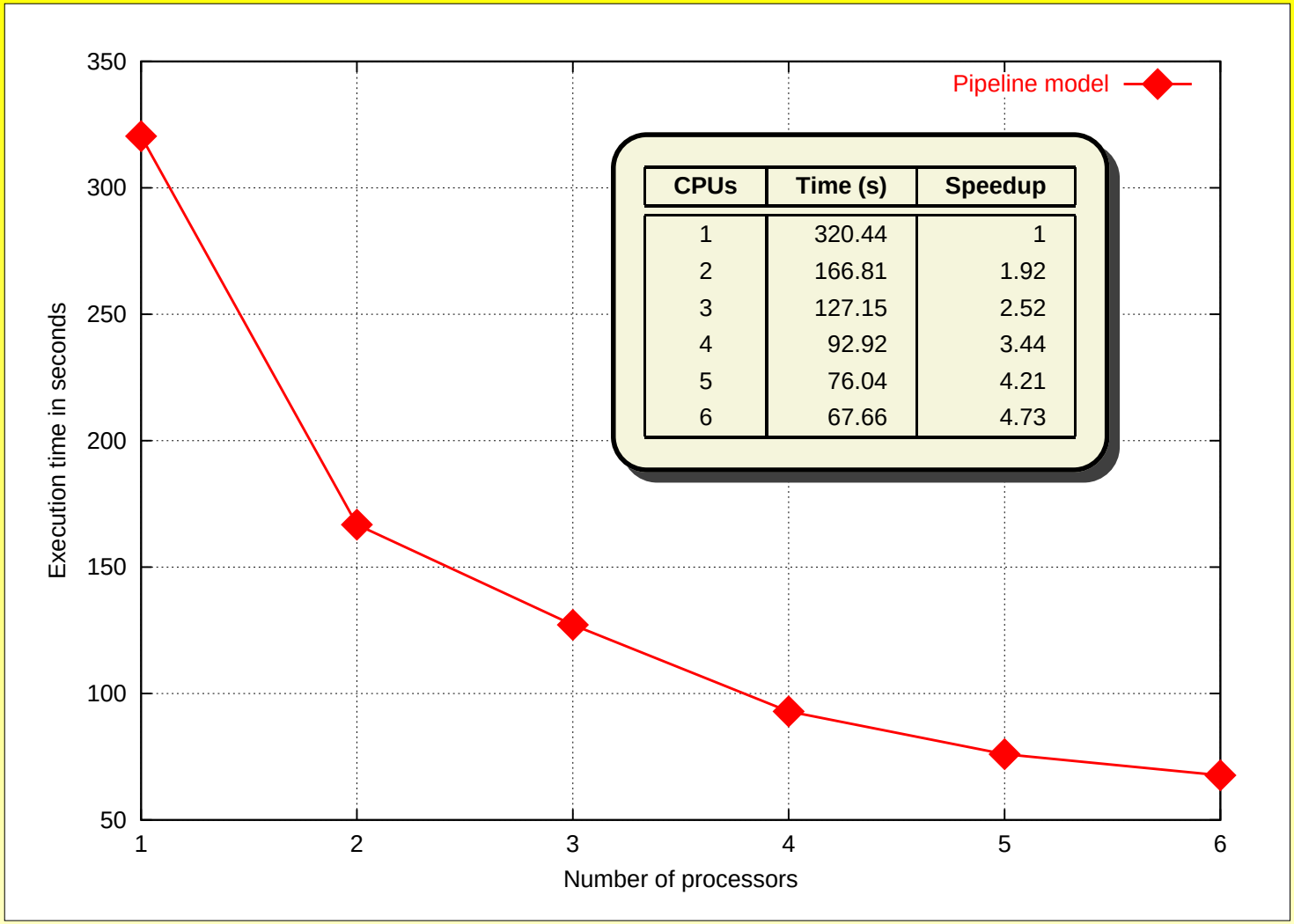
We consider $N > 2$ nodes $M_1, M_2, \ldots, M_N$. M_1 is a *source* node, which emits a packet every time unit. M_N is a *sink*, and M_j, $1 < j < N$ are servers with infinite buffers which require 1 time unit to process each packet.

In the experiment we had $N = 480$, and the system was simulated up to time $T = 10^4$.



The system was partitioned so that, with k processors, N/k adjacent nodes are assigned to each processor.

Pipeline results



CPUs	Time (s)	Speedup
1	320.44	1
2	166.81	1.92
3	127.15	2.52
4	92.92	3.44
5	76.04	4.21
6	67.66	4.73

Conclusions

From the first experiments, distributed simulation seems a viable approach to speed up complex simulations.

An experimental framework made of a few simple C++ classes has been developed for evaluation purposes. Hopefully, such framework will evolve into a more robust one.

This framework will be used to evaluate the performances of a computing farm with 120 dual-processor nodes which will be built in Padova