

JiST – Java in Simulation Time



Virtual Machine-based Simulation

Rimon Barr

<barr@cs.cornell.edu>

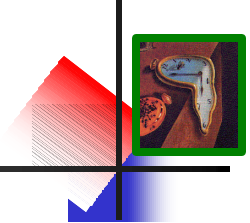
**Wireless Network Lab – Prof. Zygmunt J. Haas
Computer Science, Cornell University**

25 March 2003

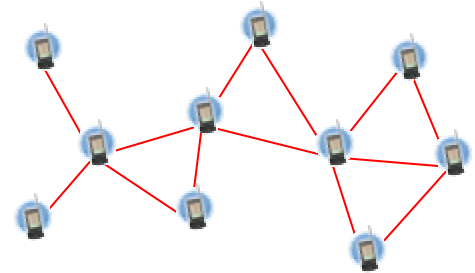
York University, Toronto

<http://www.cs.cornell.edu/barr/repository/jist/>

motivation: wireless networks

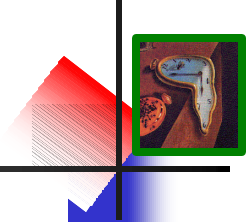


- **discrete event simulators** are useful and needed
 - in physics, biology, finance, meteorology, etc.
- e.g., **wireless networks**
 - published ad hoc network simulations
 - lack network **size** - **~500** nodes; or
 - compromise **detail** - packet level; or
 - curtail **duration** - few minutes; or
 - are of sparse **density** - $<10/\text{km}^2$; or
 - reduce network **traffic** - few packets per node
 - i.e. limited simulation scalability
- wish to simulate
 - a university campus: **30,000** students
 - the U.S. military: **100-150,000** troops
 - sensor networks, smart dust, Ubicomp with **Hundreds of thousands** of cheap wireless devices distributed across the environment



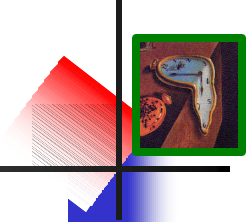
Simulation scalability is important

what is a simulation?



- **unstructured simulation: computers compute**
- **time structured: event-oriented vs. process-oriented**
- **discrete event simulator is a program that:**
 - encodes the simulation **model**
 - stores the **state** of the simulated world
 - performs **events** at discrete simulation times
 - **loops** through a temporally ordered **event queue**
 - works through **simulation time** as quickly as possible
- **desirable properties of a simulator:**
 - **correctness** - valid simulation results
 - **efficiency** - performance in terms of throughput and memory
 - **transparency** - separate correctness from efficiency:
 - write "simple" program in a **standard** language
 - provide implicit optimization, concurrency, distribution, portability, etc.

how do we build simulators?



systems

- **simulation kernels**
 - control scheduling, IPC, clock
 - processes run in virtual time
 - e.g. TimeWarp OS, Warped
 - 👍 transparency 👎 efficiency
- **simulation libraries**
 - move functionality to user-space for performance; monolithic prog.
 - usually event-oriented
 - e.g. Yansl, Compose, ns2
 - 👎 transparency 👍 efficiency

languages

- **generic simulation languages**
 - introduce entities, messages and simulation time semantics
 - event and state constraints allow optimization
 - both event and process oriented
 - e.g. Simula, Parsec/GloMoSim
- **application-specific languages**
 - e.g. Apostle, TeD
 - 👎 transparency 👎 efficiency
 - 👍 👎 new language

virtual machines



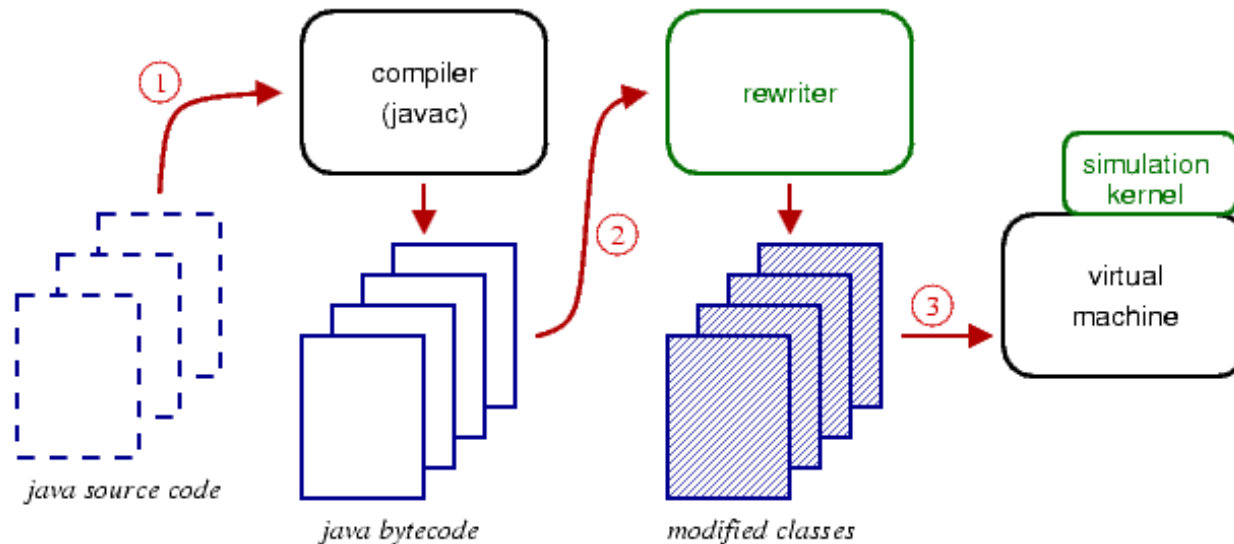
- **JiST – Java in Simulation Time**
 - converts a virtual machine into a simulation platform
 - no new language, no new library, no new runtime
 - merges **modern language** and **simulation semantics**
 - combines systems-based and languages-based approaches
 - result: **virtual machine-based simulation**

	kernel	library	language	JiST
transparent	++		++	++
efficient		+	+	++
standard	++	++		++

system architecture



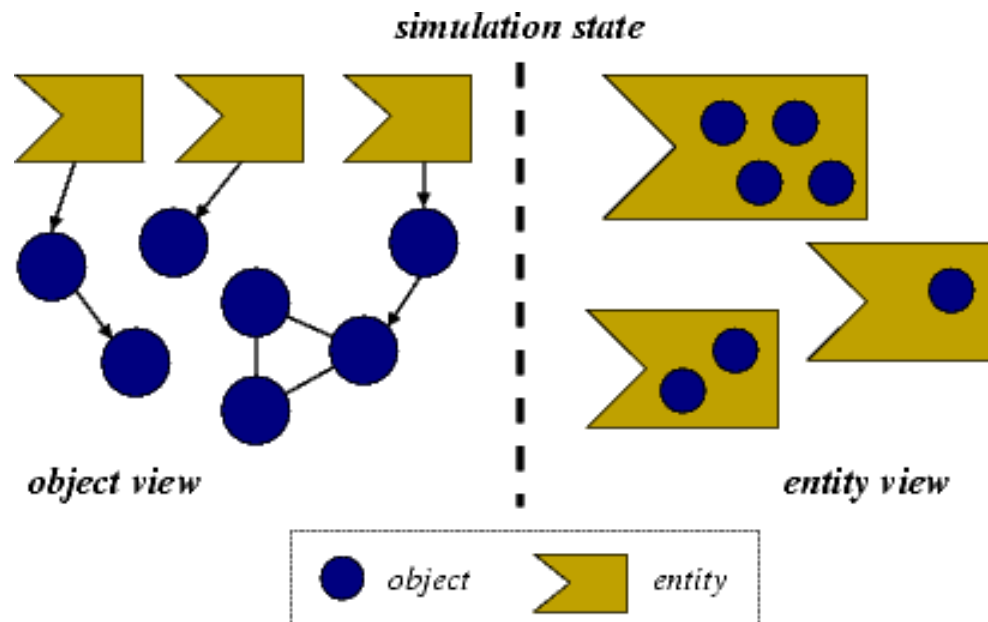
1. Compile simulation with standard Java compiler
2. Run simulation within JiST (within Java); simulation classes are dynamically rewritten to introduce **simulation time** semantics:
 - extend the Java object model and execution model
 - instructions take zero (simulation) time
 - time explicitly advanced by the program: `sleep(time)`
 - **progress of time is dependent on program progress**
3. Rewritten program interacts with simulation kernel



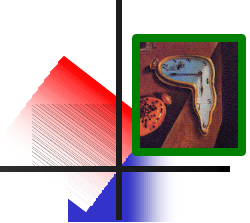
jist object model



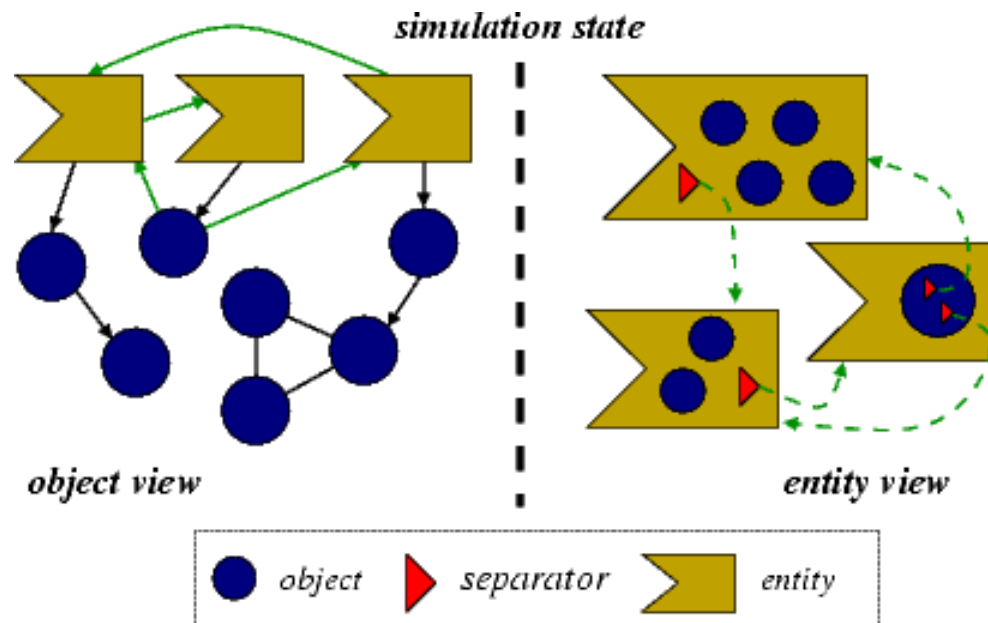
- program state contained in **objects**
- objects contained in **entities**
 - think of an entity as a simulation component
 - an entity is any class tagged with the `Entity` interface
 - each entity runs at its own simulation **time**
 - as with objects, entities do not share state
 - akin to a JKernel process in spirit, but without the threads!



jist execution model



- **entity methods are an event interface**
 - **simulation time invocation**
 - **non-blocking**; invoked at caller entity time; no continuation
 - like co-routines, but scheduled in simulation time
- **entity references replaced with separators**
 - event channels; act as **state-time boundary**
 - demarcate a TimeWarp-like process, but at finer granularity



a basic example



- the “hello world” of event simulations

```
class HelloWorld implements JistAPI.Entity
{
    public void hello()
    {
        JistAPI.sleep(1);
        hello();
        System.out.println("hello world, " +
            "time=" + JistAPI.getTime() );
    }
}
```

- demo!

Java	JiST
Stack overflow @hello	hello world, time=1 hello world, time=2 hello world, time=3 etc.

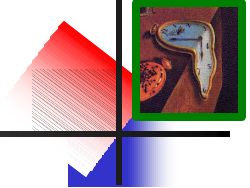


- **JistAPI** class is the JiST kernel **system call** interface
- permits **standard Java** compilation and execution

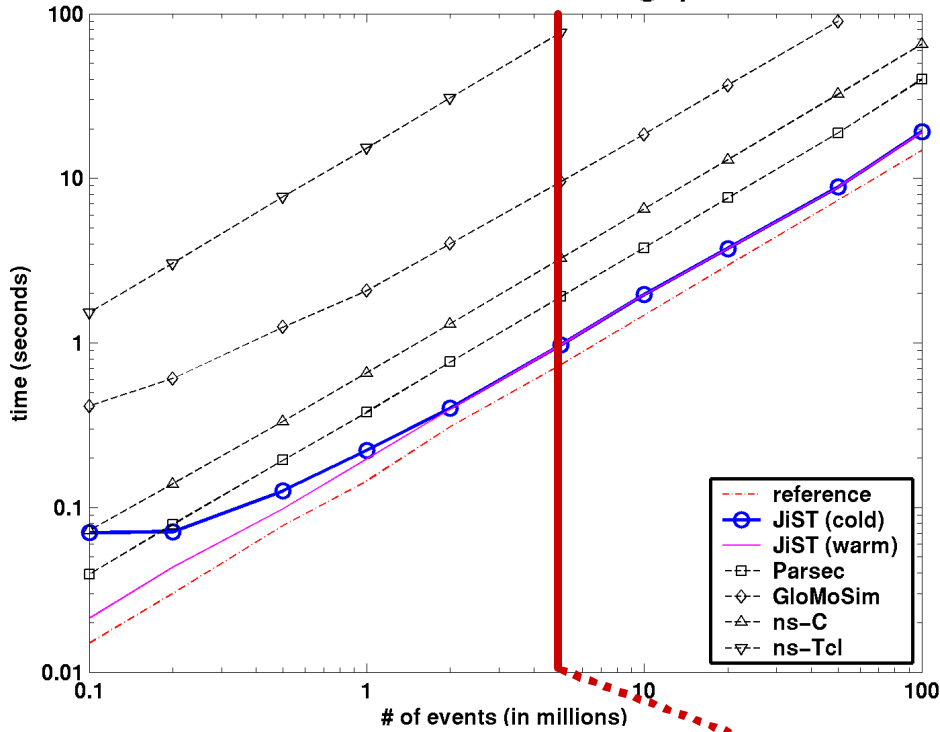
```
// used in hello example
interface Entity                - tag object as entity
long getTime()                 - return simulation time
void sleep(long ticks)         - advance simulation time

// others, to be introduced shortly
interface Timeless              - tag object as timeless
interface Proxiabile            - tag object as proxiabile
Entity proxy(target, intface)   - create proxy entity
class Continuation ext. Error   - tag method as blocking
void run(type,name,args,...)    - run program or script
void runAt(Runnable r)         - schedule procedure
void endAt(long time)          - end simulation
Channel createChannel()         - sim. time CSP Channel
void installRewrite(rewriter)   - install transformation
EntityRef THIS                  - this entity reference
EntityRef ref(Entity e)         - get entity reference
// ... and few more
```

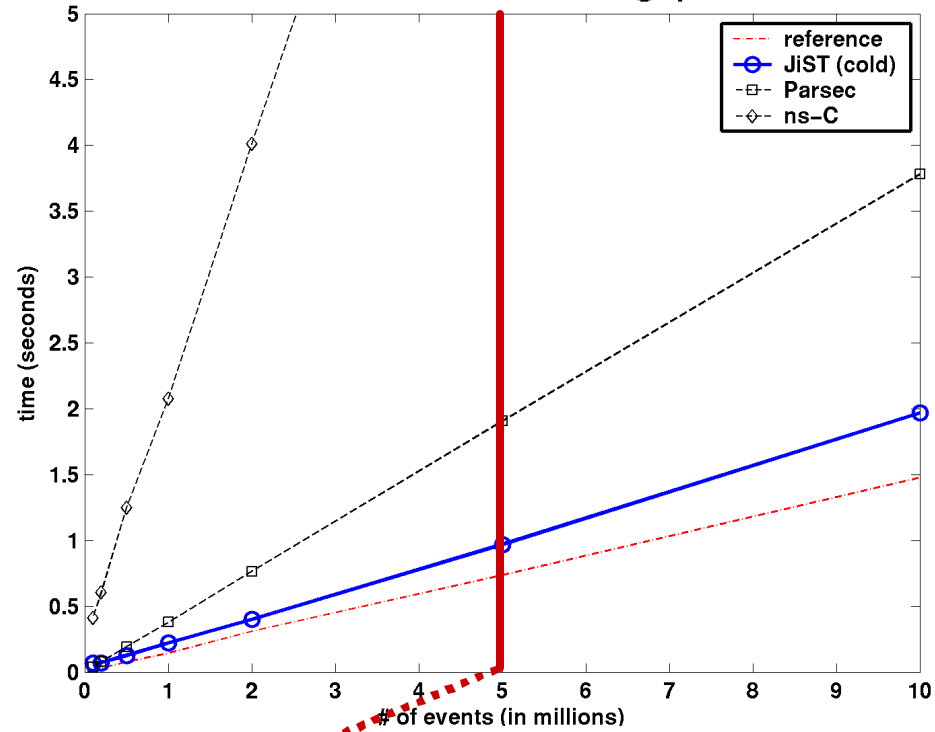
jist micro-benchmark: event throughput



Simulation event throughput

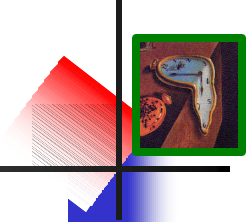


Simulation event throughput

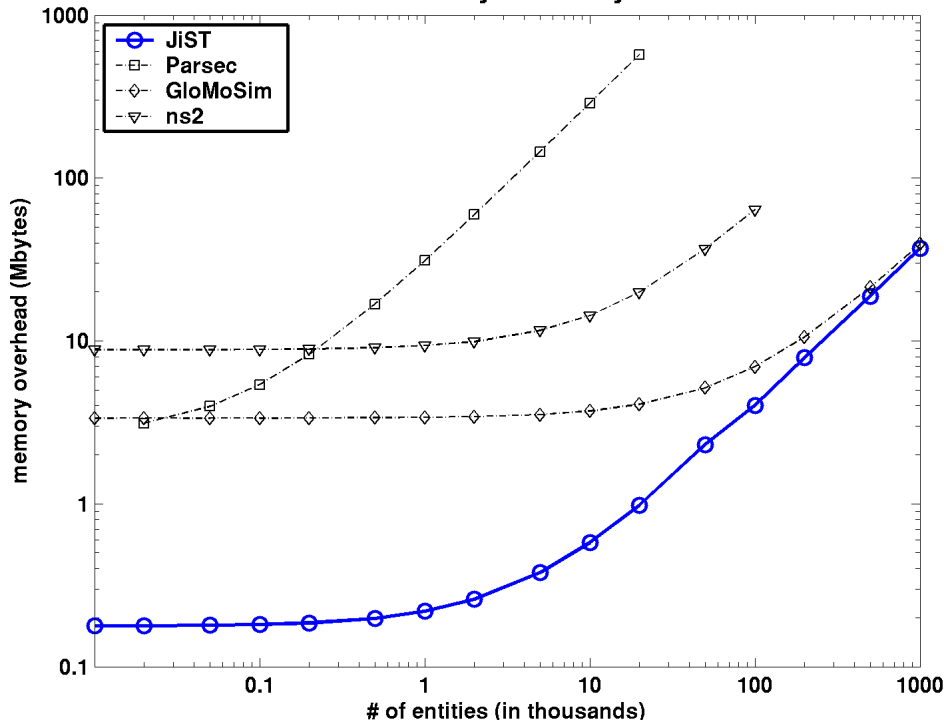


5x10 ⁶ events	time (sec)	vs. reference	vs. JiST
reference	0.74		0.76x
JiST	0.97	1.31x	
Parsec	1.91	2.59x	1.97x
ns2-C	3.26	4.42x	3.36x
GloMoSim	9.54	12.93x	9.84x
ns2-Tcl	76.56	103.81x	78.97x

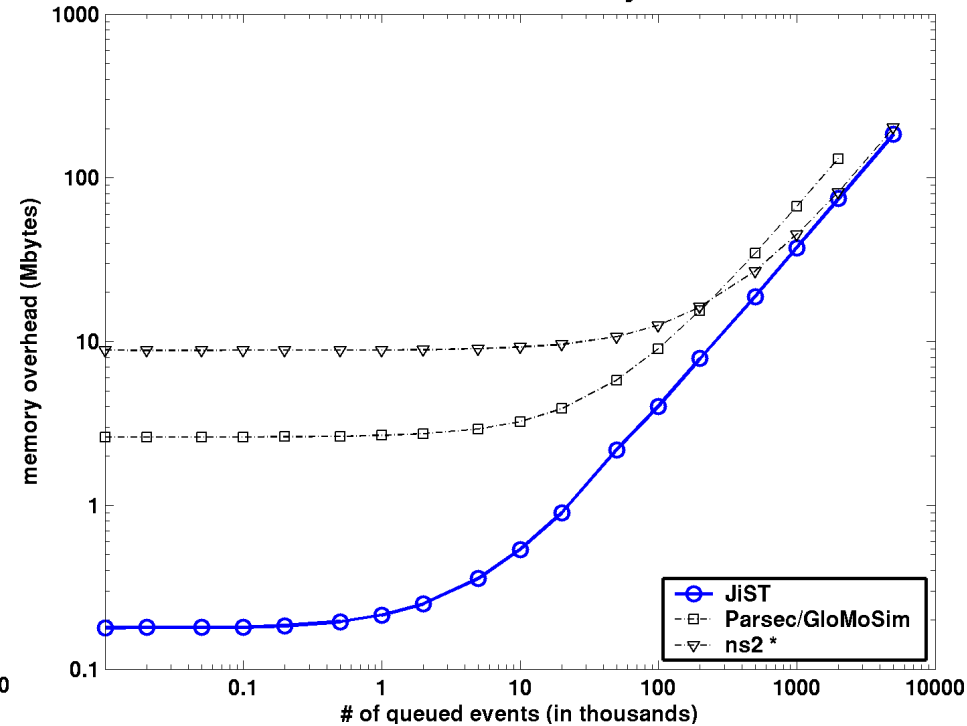
jist micro-benchmark: memory overhead



Simulation entity memory overhead



Simulation event memory overhead



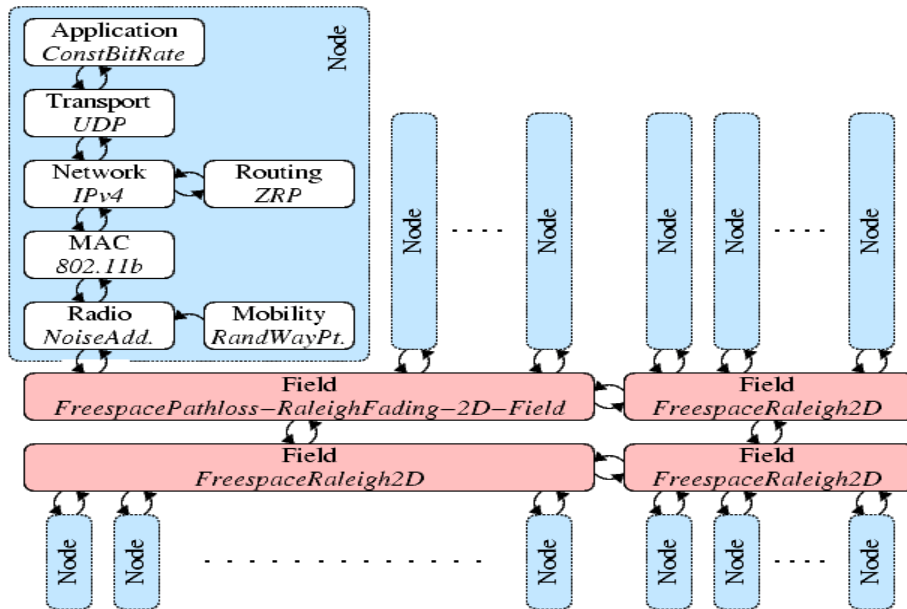
memory	per entity	per event	10K nodes sim.
JiST	36 B	36 B	21 MB
GloMoSim	36 B	64 B	35 MB
ns2 *	544 B	40 B	74 MB
Parsec	28536 B	64 B	2885 MB



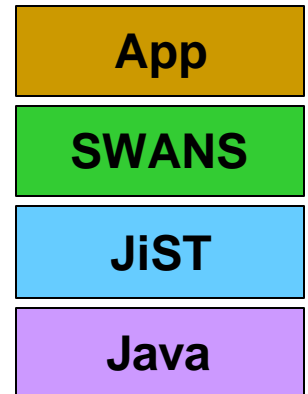


- **Scalable Wireless Ad hoc Network Simulator**

- similar functionality to ns2 and GloMoSim, but...
- runs **standard Java network applications** over simulated networks
- can simulate networks of **1,000,000 nodes** sequentially, on a single commodity uni-processor
- runs on top of **JiST**; SWANS is a JiST application
- uses **hierarchical binning** for efficient propagation
- **component-based** architecture written in Java

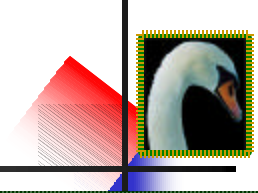


sim. stack



	files	classes	lines	semi
JiST	29	113	14057	3502
SWANS	85	220	28617	6536
Other	32	80	6939	2509
	146	413	49613	12547

existing wireless simulators



sequential

ns2 is the gold standard

- written in C++ with Tcl bindings
- created for TCP simulation, modified for wireless networks
- processor and memory intensive
- sequential; max. **~500 nodes**
- recently “fixed” for **~5000 nodes**

OpNet – popular commercial option

- good modeling capabilities
- poor scalability

GloMoSim

- implemented in Parsec, a custom C-like language
- implements “node aggregation,” to conserve memory
- shown **~10,000 nodes** on NUMA machine (SPARC 1000, est. \$300k)

custom-made simulators

- fast, specialized computation
- lack sophisticated execution and also credibility

parallel

PDNS – parallel distributed ns2

- event loop uses RTI-KIT
- uses fast inter-connect to distribute memory requirements
- shown **~100,000 nodes**

SWAN

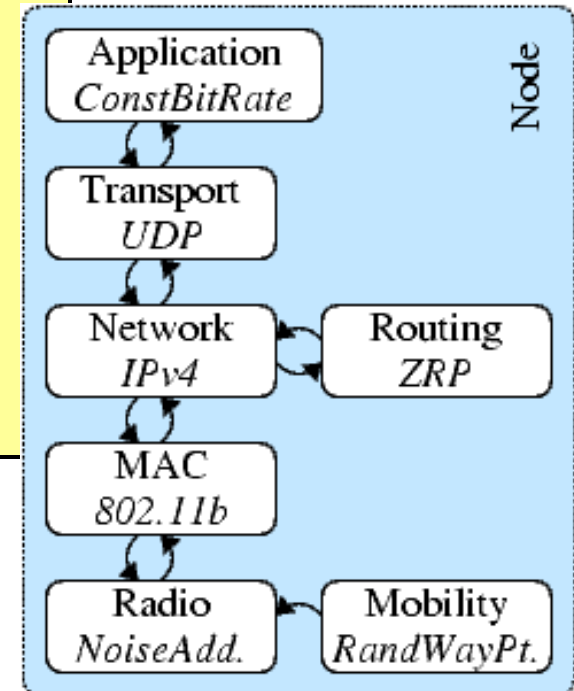
- parallelized and distributed using the DaSSF framework
- similar capabilities to GloMoSim
- shown **~100,000 nodes**

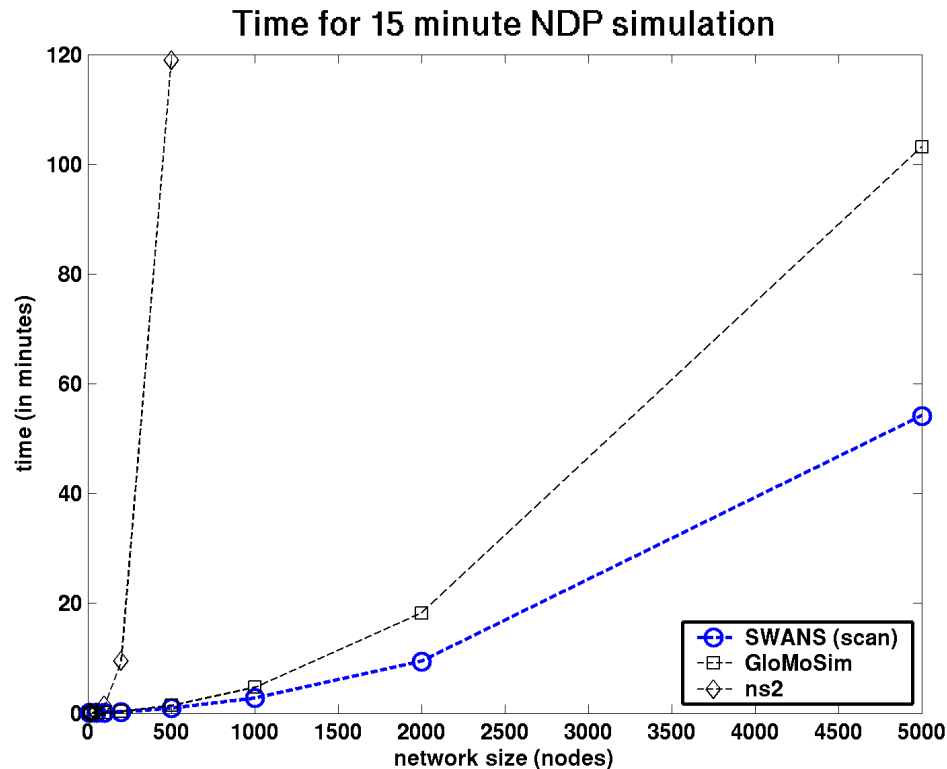
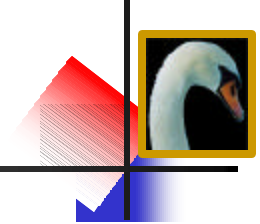
rule of thumb: extra 10x in scale,
using at least 10x hardware and cost

SWANS components



function	implementation
application	- <i>heartbeat</i> ; <i>any Java network application</i>
transport	- <i>UDP</i> ; <i>TCP</i> [Tamtoro]
network	- <i>IPv4</i>
routing	- <i>ZRP</i> ; <i>DSR</i> [Viglietta]; <i>AODV</i> [Lin]
link	- <i>802.11b</i> ; <i>naïve</i> ; <i>wired</i>
placement	- <i>random</i> ; <i>input file</i>
mobility	- <i>static</i> ; <i>random waypoint</i> ; <i>input file</i>
interference	- <i>independent</i> , ns2; <i>additive</i> , GloMoSim
fading	- <i>zero</i> ; <i>Raleigh</i> ; <i>Rician</i>
pathloss	- <i>free-space</i> ; <i>two-ray</i>
propagation	- <i>linear scan</i> , ns2;
algorithm	<i>flat binning</i> , GloMoSim; <i>hierarchical binning</i>

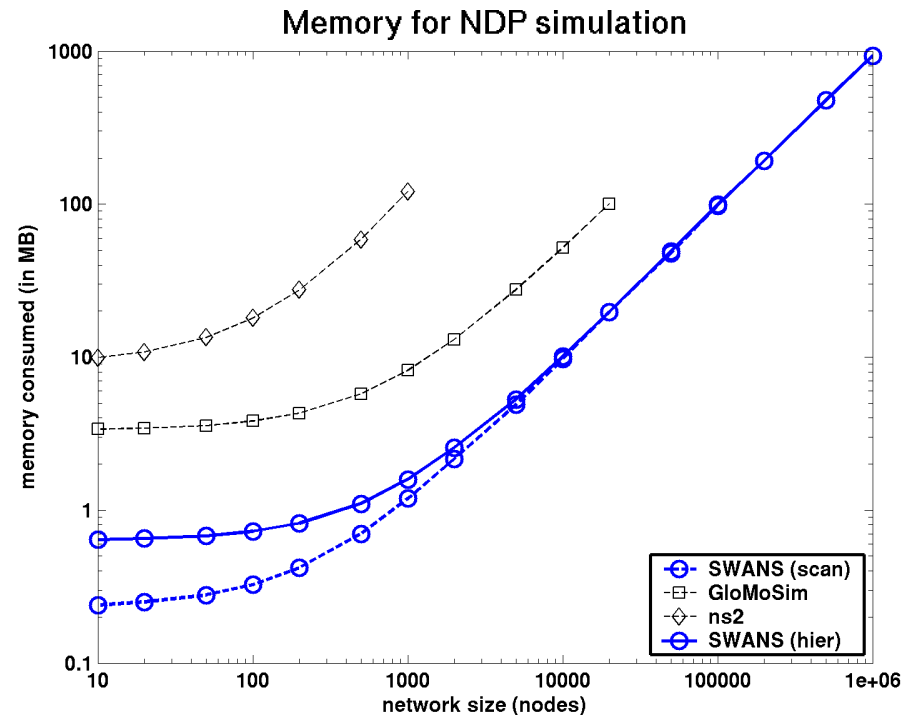
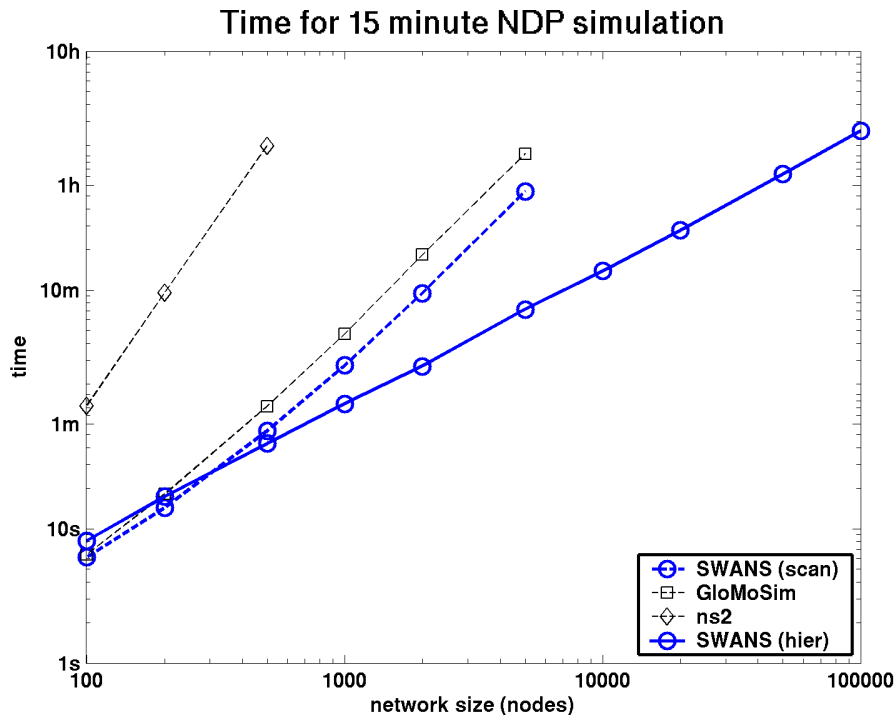




- **simulation configuration**

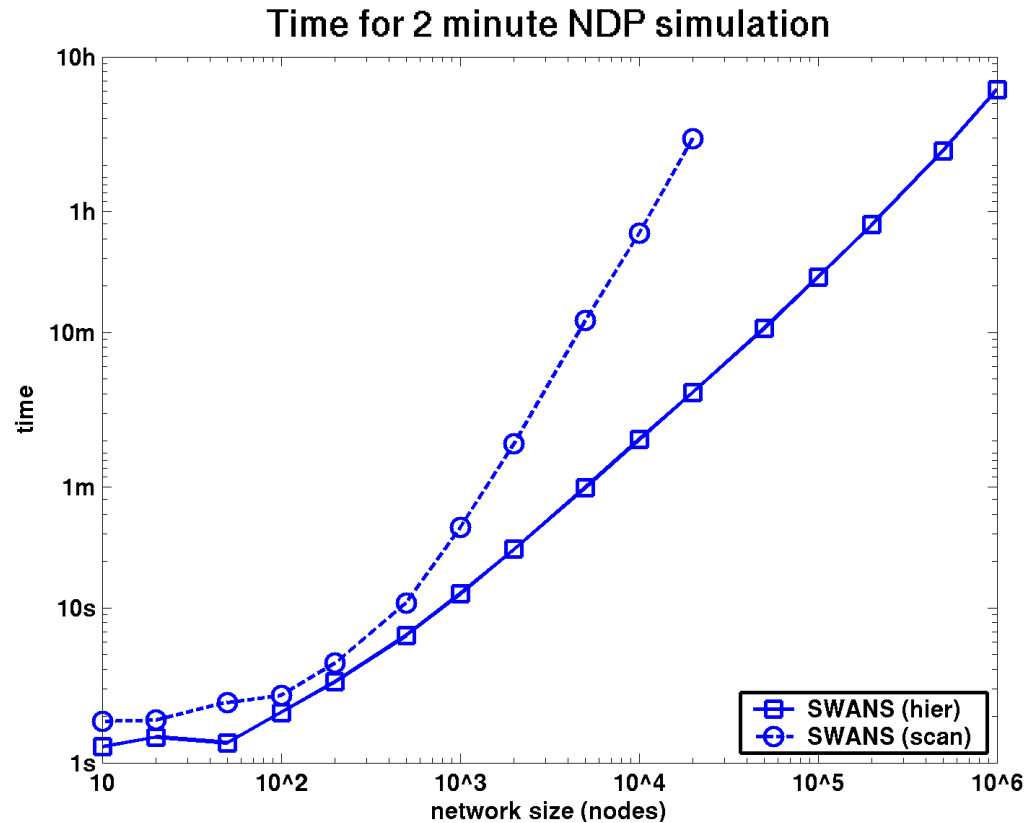
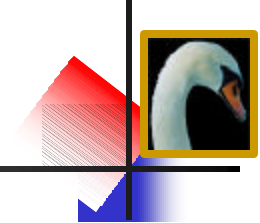
- **application** - heartbeat neighbor discovery
- **field** - 5x5km²; free-space path loss; zero fading
- **mobility** - random waypoint: v=2-5m, p=10s
- **radio** - additive noise; standard power, gain, etc.
- **stack** - 802.11b, IPv4, UDP

SWANS performance



$t=15m$ nodes	ns2		GloMoSim		SWANS		SWANS-hier	
	time	memory	time	memory	time	memory	time	memory
500	7136.3 s	58761 KB	81.6 s	5759 KB	53.5 s	700 KB	43.1 s	1101 KB
5000			6191.4 s	27570 KB	3249.6 s	4887 KB	433.0 s	5284 KB
50000					47717 KB		4377.0 s	49262 KB

SWANS performance

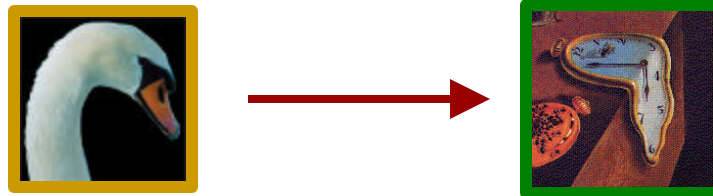


$t=2m$	SWANS-hier			
nodes	10,000	100,000	1 million	per node
initial memory	13 MB	100 MB	1000 MB	1.0 KB
avg. memory	45 MB	160 MB	1200 MB	1.2 KB
time	2 m	25 m	5.5 h	20 ms

performance summary

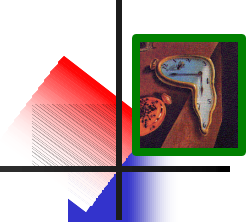


- **SWANS scalability**
 - can simulate **million node wireless networks**
 - **hierarchical binning** allows linear scaling with network size
- **SWANS is a JiST application**
 - a simulation program written using the “JiST approach”

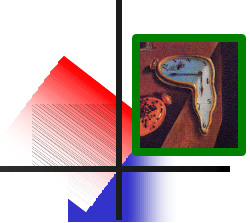


- **scalability depends on:**
 - **time** – efficient simulation event processing
 - **space** – efficient simulation state encoding

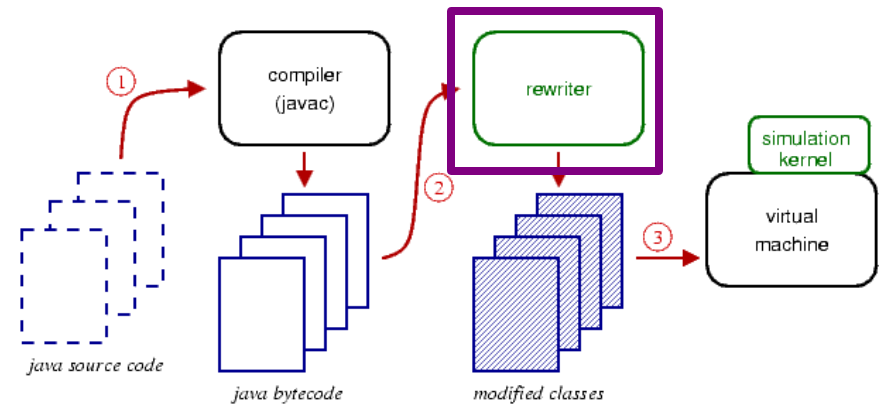
benefits of the jist approach



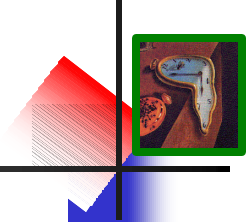
- **more than just performance...**
- **application-oriented benefits**
 - **type safety** source and target statically checked
 - **event types** not required (implicit)
 - **event structures** not required (implicit)
 - **debugging** dispatch source location and state available
- **language-oriented benefits**
 - **Java** standard language, compiler, runtime
 - **garbage collection** cleaner code, memory savings
 - **reflection** script-based simulation configuration
 - **safety** fine grained isolation
 - **robustness** no memory leaks, no crashes
- **system-oriented benefits**
 - **IPC** no context switch, no serialization, zero-copy
 - **Java kernel** cross-layer optimization
 - **rewriting** no source-code access required
 - **distribution** provides a single system image abstraction
 - **concurrency** model supports parallel and speculative execution
- **hardware-oriented benefits**
 - **cost** COTS hardware and clusters
 - **portability** runs on everything



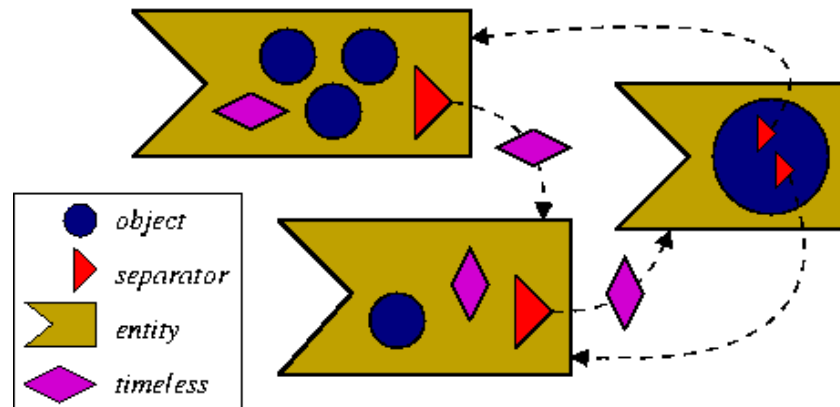
- **rewriter properties**
 - dynamic class loader
 - **no source code access required**
 - operates on application packages, not system classes
 - uses Apache Byte Code Engineering Library (BCEL)
 - **allows orthogonal additions, transformations and optimizations**
- **rewriting phases**
 - application-specific rewrites
 - verification
 - add entity self reference
 - intercept entity state access
 - add method stub fields
 - intercept entity invocations
 - modify entity creation
 - modify entity references
 - modify typed instructions
 - continuable analysis
 - continuation transformation
 - translate JiST API calls



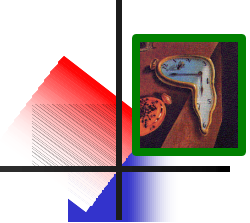
zero-copy semantics



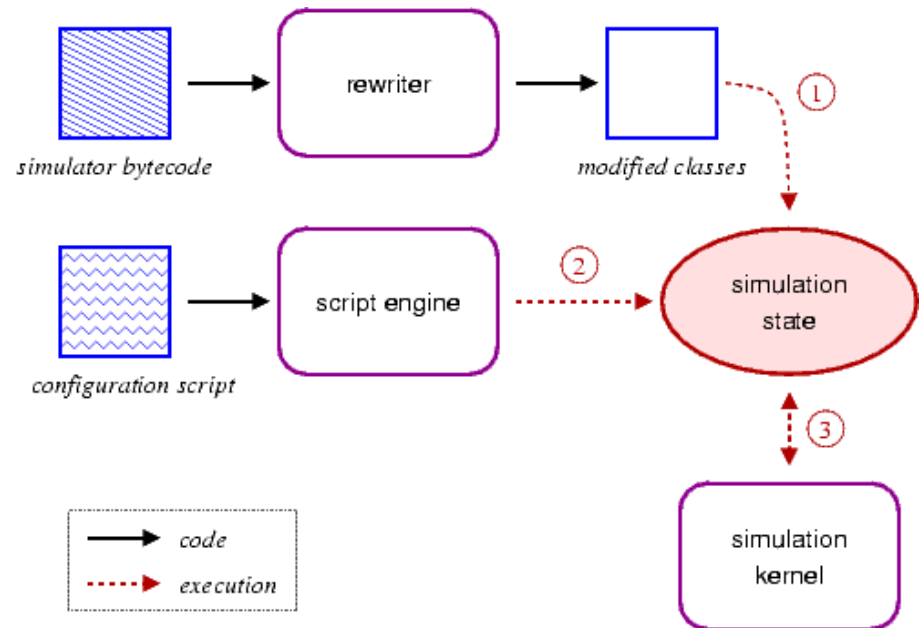
- **timeless object: a **temporally stable** object**
 - **inferred statically** as open-world immutable
 - or **tagged explicitly** with the **Timeless** interface
- **benefits**
 - **pass-by-reference **saves memory copy****
 - zero-copy semantics for inter-entity communication
 - **saves memory** for common shared objects
 - e.g. broadcast network packets
 - rewrite **new** of common types to **hashcons**



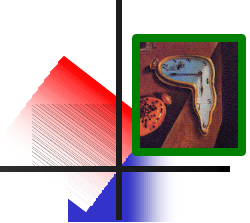
configurability



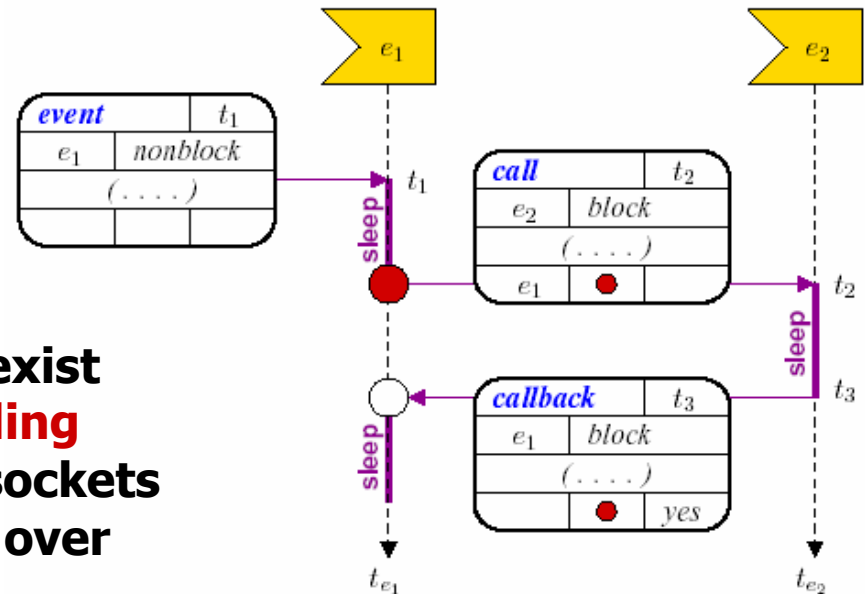
- **configurability** is essential for simulators
 1. source level reuse; **recompilation**
 2. **configuration files** read by driver program
 3. driver program is a **scripting language engine**
- support for multiple scripting languages by **reflection**
 - **no additional code**
 - **no memory overhead**
 - **no performance hit**
 - **Bsh** - scripted Java
 - **Jython** - Python
 - **Smalltalk, Tcl, Ruby, Scheme and JavaScript**



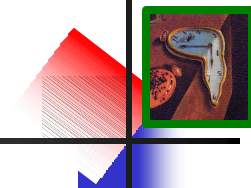
simulations using real applications



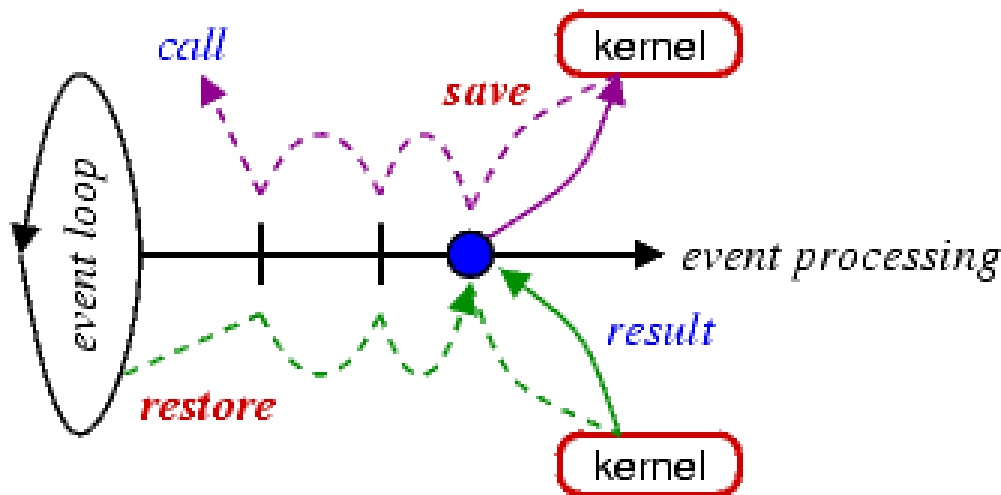
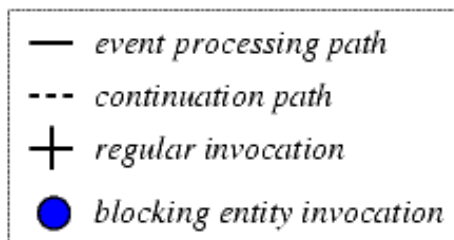
- using entity method invocations...
 - one can easily write **event-driven** entities.
 - what about **process-oriented** simulation?
- **blocking events**
 - any entity method that “throws” a **Continuation** exception
 - event processing frozen at invocation
 - continues after call event completes, at some later simulation time
- **benefits**
 - no explicit process
 - blocking and non-blocking coexist
 - akin to **simulation time threading**
 - can build simulated network sockets
 - can **run standard applications** over these simulated sockets



capturing continuations



- mark entity method as blocking: throws **Continuation**
- saving and restoring the stack is non-trivial in Java!



Before CPS transform:

```

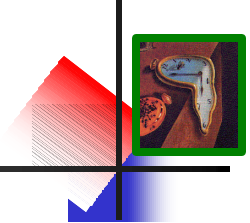
1 METHOD continuable:
2   instructions
3   invocation BLOCKING
4   more instructions
  
```

After CPS transform:

```

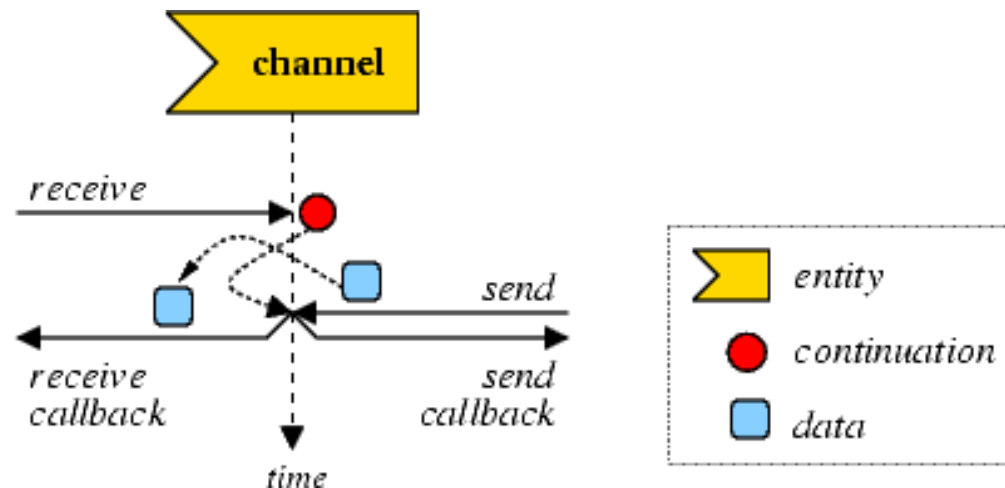
1 METHOD continuable:
2   if jist.isRestoreMode:
3     jist.popFrame f
4     switch f.pc:
5       case PC1:
6         restoreLocals f.locals
7         restoreStack f.stack
8         goto PC1
9     ...
10  instructions
11
12  setPC f.pc, PC1
13  saveLocals f.locals
14  saveStack f.stack
15
16 PC1:
17  invocation BLOCKING
18  if jist.isSaveMode:
19    jist.pushFrame f
20    return
21
22  more instructions
  
```

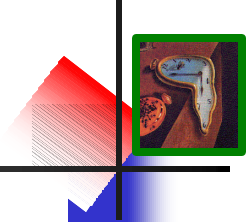
simulation time concurrency



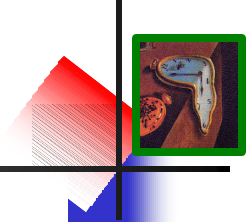
using continuations...

- **simulation time Thread**
 - **cooperative** concurrency
 - can also support **pre-emptive**, but not necessary
- **simulation time concurrency primitives:**
 - **CSP Channel:** `JistAPI.createChannel()`
 - **locks, semaphores, barriers, monitors, FIFOs, ...**





- **simulation time transformation**
 - extend Java object model with entities
 - extend Java execution model with events
 - language-based simulation kernel
- **extensions to the model**
 - **timeless objects**: pass-by-reference to avoid copy, saves memory
 - **reflection**: scripting, simulation configuration, tracing
 - **tight event coupling**: cross-layer optimization, debugging
 - **proxy entities**: interface-based entity definition
 - **blocking events**: call and callback, CPS transformation, standard applications
 - **simulation time concurrency**: Threads, Channels and other synch. primitives
 - **distribution**: location independence of entities, single system image abstraction
 - **parallelism**: concurrent and speculative execution
 - **orthogonal additions, transformations and optimizations**
- **platform for simulation research**
 - e.g. reverse computations in optimistic simulation [Carothers '99]
 - e.g. stack-less process oriented simulation [Booth '97]



- **JiST** – **J**ava **i**n **S**imulation **T**ime
 - converts **virtual machine** into simulation platform
 - merges systems and language-based approaches
 - runs **SWANS**: **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator
 - **efficient**: both in terms of **throughput** and **memory**
 - **flexible**: timeless objects, reflection-based scripting, tight event coupling, proxy entities, continuations and blocking methods, simulation time concurrency, distribution, concurrency ... serve as a research platform

JiST – Java in Simulation Time

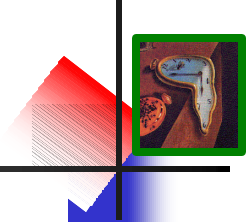


Virtual Machine-based Simulation

THANK YOU.

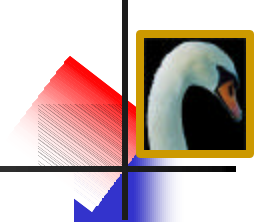
<http://www.cs.cornell.edu/barr/repository/jist/>

simulation time



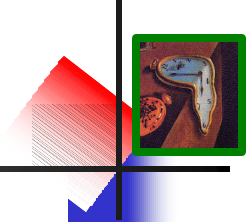
- **actual time**
 - standard Java program execution semantics
 - progress of program independent of time
- **real time**
 - need stronger guarantees on progress
 - progress of program made dependent on time
- **simulation time**
 - **progress of time is dependent on program progress**
 - instructions take zero (simulation) time
 - time explicitly advanced by the program: `sleep(time)`
 - simulation event loop embedded in virtual machine
 - rewriter introduces simulation time semantics by
 - extending the Java object model
 - extending the Java execution model

JiST features in SWANS



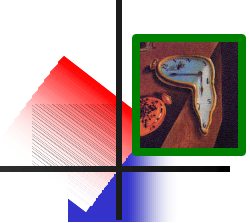
- **SWANS is a JiST application**
 - **entity invocation** tracking time
no context switching; zero-copy; cross-layer optimizations;
type-safety; implicit event structures and types
 - **timeless objects** packets
saves memory; simplifies memory management
 - **proxy entities** network stack
restricts communication pattern; simplifies development
 - **reflection** script-based configuration
no memory or performance hit; no additional code
 - **continuations** socket implementations
run standard Java network applications over simulated network

tight event coupling

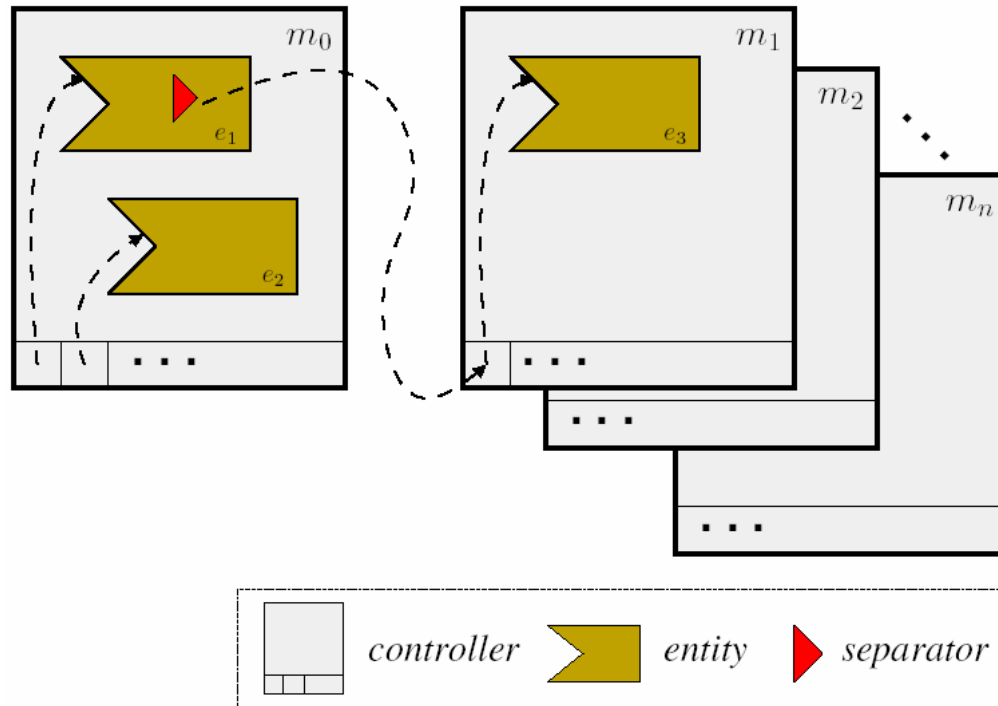


- **tight coupling of event dispatch and delivery provides numerous benefits:**
 - **type safety** source and target of event statically verified by compiler
 - **event typing** not required; events automatically type-cast as they are dequeued
 - **event structures** not required; event parameters automatically marshaled
 - **debugging** event dispatch location and state are available
 - **execution** transparently allows for parallel, optimistic and distributed execution

distribution and concurrency



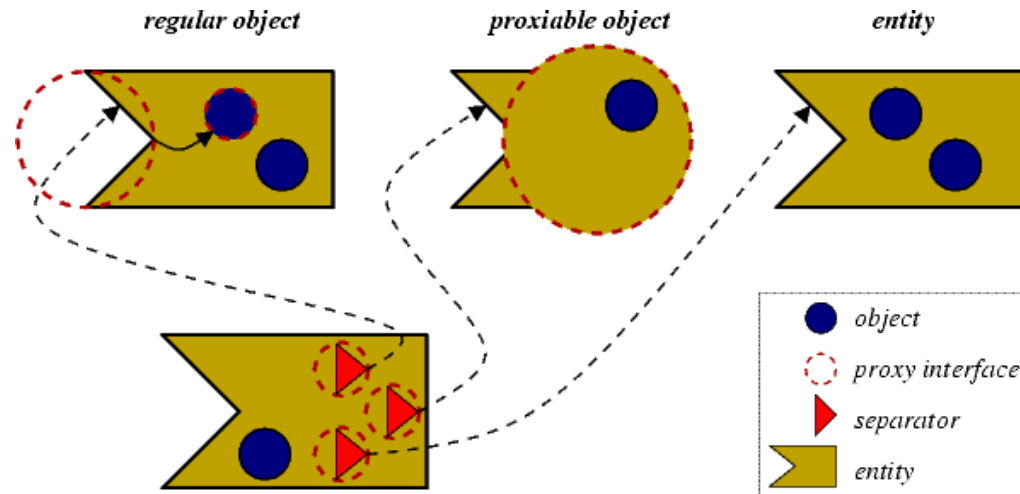
- **parallelism** **multiple controllers**
- **distribution** **separators allow migration and provide location independence**
- **optimism** **check-pointing implicitly supported**

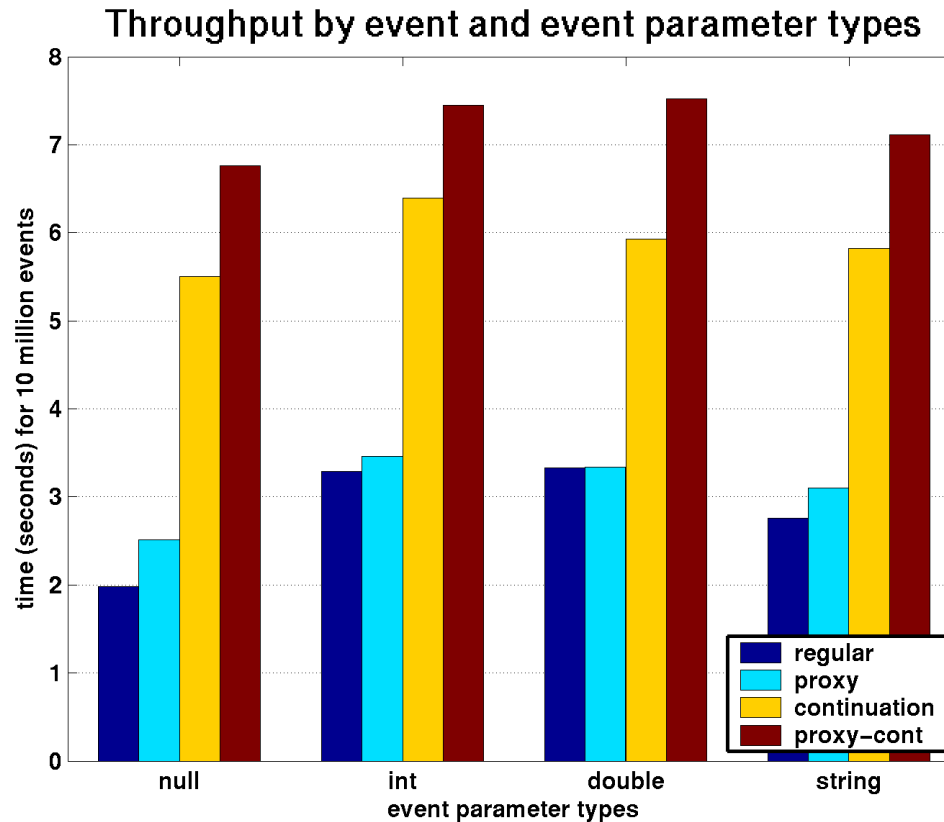


proxy entities



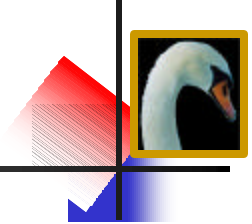
- proxy entities **relay events to a target**
 - possible targets: regular object, **proxiabile** object, entity
 - proxiabile: any object tagged with **Proxiabile** interface
- benefits
 - **equivalent performance**: `JistAPI.proxy(target, intfce)`
 - **interface-based**: does not interfere with object hierarchy
 - mix simulation time invocations with regular invocations
 - provides a **capability-like isolation** for entities



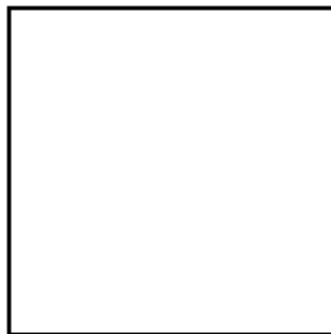


- manually need to box Java primitive types
- tail invocations not properly detected
- need API for type-safe stack access
- exceptions are very expensive

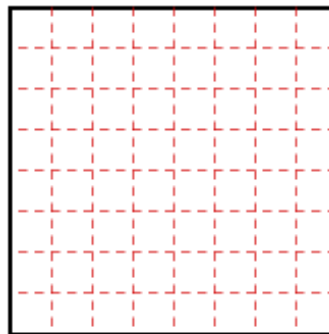
hierarchical binning



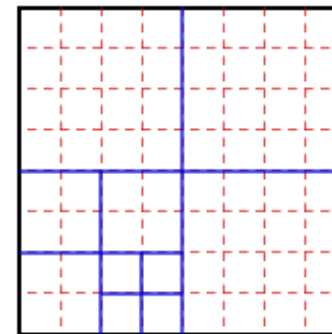
- **simulating signal propagation**
 - critical to performance and scalability
 - **find radios within a given radius**
 - prior approaches
 - linear scan ns2
 - flat binning GloMoSim, ns2' (MSWiM '03)
 - function caching SWiMNet (WN '01), ns2' (WSC '03)
 - **hierarchical binning**
 - location update: amortized expected **constant time**
 - neighborhood search: time linear in receivers, **O(result set)**
 - amortized expected asymptotically optimal time



linear lookup



flat binning



hierarchical binning



JistAPI.java

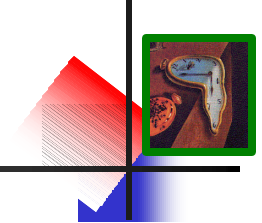
```
1 package jist.runtime;
2
3 public class JistAPI
4 {
5     public static interface Entity { }
6     public static class Continuation extends Error { }
7     public static interface Timeless { }
8
9     public static long getTime() { ... }
10    public static void sleep(long n) { }
11    public static void end() { }
12    public static void endAt(long t) { }
13
14    public static JistAPI.Entity THIS;
15    public static EntityRef ref(Entity e) { ... }
16
17    public static interface Proxiable { }
18    public static Object proxy(Object proxyTarget, Class proxyInterface) { ... }
19    public static Object proxyMany(Object proxyTarget, Class[] proxyInterface) { ... }
20
21    public static final int RUN_CLASS = 0;
22    public static final int RUN_BSH = 1;
23    public static final int RUN_JPY = 2;
24    public static void run(int type, String name, String[] args, Object properties) { }
25
26    public static Channel createChannel() { ... }
27
28    public static void setSimUnits(long ticks, String name) { }
29
30    public static interface CustomRewriter {
31        JavaClass process(JavaClass jcl);
32    }
33    public static void installRewrite(CustomRewriter rewrite) { }
34 }
```

example: hello world



```
1 import jist.runtime.JistAPI;
2
3 class hello implements JistAPI.Entity
4 {
5     public static void main(String[] args)
6     {
7         System.out.println("simulation start");
8         hello h = new hello();
9         h.myEvent();
10    }
11
12    public void myEvent()
13    {
14        JistAPI.sleep(1);
15        myEvent();
16        System.out.println("hello world, t="
17            +JistAPI.getTime());
18    }
19 }
```

example: scripts



hello.bsh

```
1 System.out.println("starting simulation from BeanShell script!");
2 import jist.minisim.hello;
3 hello h = new hello();
4 h.myEvent();
```

BeanShell – scripted Java

hello.jpy

```
1 print 'starting simulation from Jython script!'
2 import jist.minisim.hello as hello
3 h = hello()
4 h.myEvent();
```

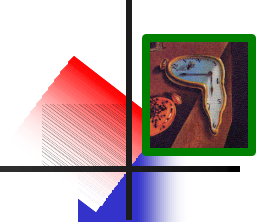
Jython – Python

example: blocking methods



```
1 import jist.runtime.JistAPI;
2
3 public class cont implements JistAPI.Entity
4 {
5     public void blocking() throws JistAPI.Continuation
6     {
7         System.out.println("called at t="+JistAPI.getTime());
8         JistAPI.sleep(1);
9     }
10
11     public static void main(String args[])
12     {
13         cont c = new cont();
14         for(int i=0; i<3; i++)
15         {
16             System.out.println("i="+i+" t="+JistAPI.getTime());
17             c.blocking();
18         }
19     }
20 }
```


rewriter overhead



class	Object	Timeless	Entity	total
base size	265438	121987	11358	398783
total increase	23171 (8.7%)	9751 (8.0%)	9887 (87.0%)	42809 (10.7%)
constant pool code, etc.	15340 (5.8%)	6817 (5.6%)	7223 (63.6%)	29380 (7.4%)
	7831 (3.0%)	2934 (2.4%)	2664 (23.5%)	13429 (3.4%)

Figure 21: Rewriter processing increases class sizes. The figures shown above are the increases in bytes (and as a percentage), from the processing of the complete SWANS code-base. The data is split into the three JiST class categories, showing that the majority of the increase occurs among entity classes and that much of the increase is due only to new constant pool entries.