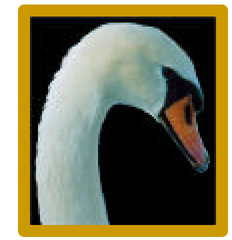
JiST –
Java in Simulation Time

for the

Scalable Simulation of Mobile Ad hoc Networks



Rimon Barr
<barr@cs.cornell.edu>
Wireless Network Laboratory
Advisor: Prof. Z. J. Haas

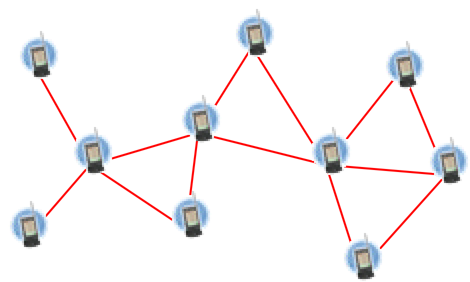
ECE Communications Seminar
Cornell University
12 November 2003

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

motivation

- discrete event simulations are useful and needed
- but, most 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/km^2; or
  - reduce network **traffic** - few packets per node

  i.e. limited simulation scalability



- A university campus
  - **30,000** students, < 4 km^2, 1 device/student
- The United States military
  - **100-150,000** troops, clustered around cities
- Sensor networks, smart dust, Ubicomp
  - **Many thousands** of cheap wireless devices distributed across the environment

Simulation scalability is important

existing simulator options

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

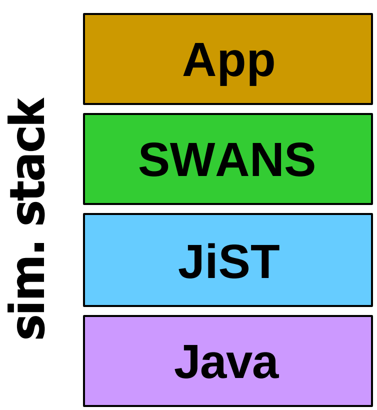
SWAN

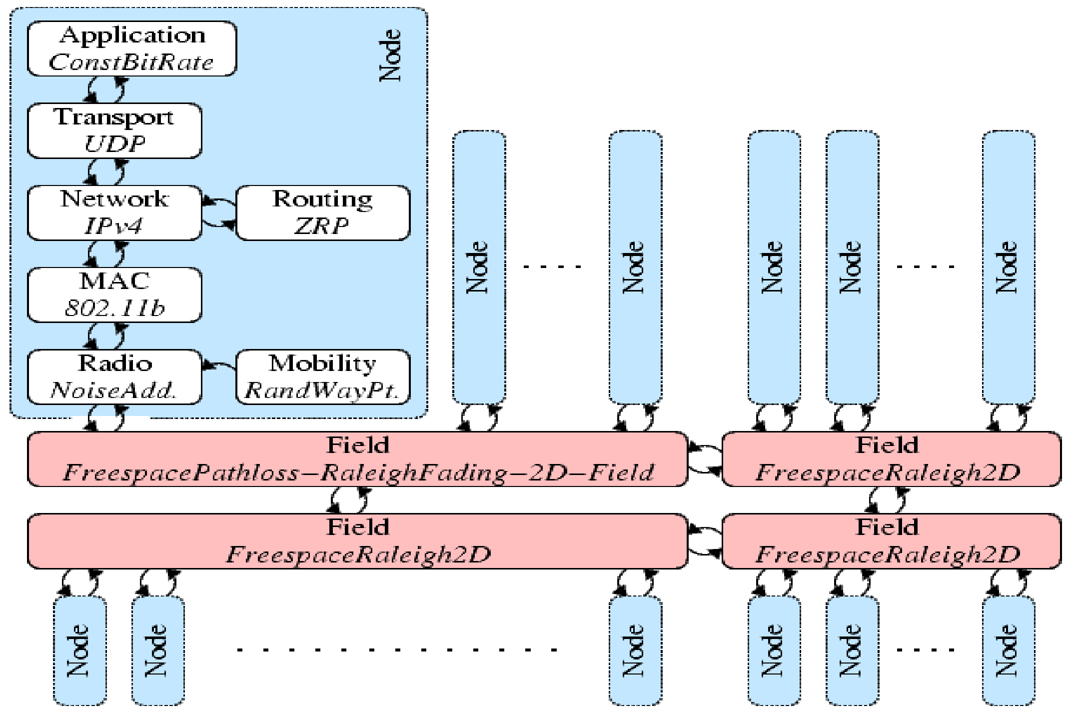
- parallelized and distributed using the DaSSF framework
- similar capabilities to GloMoSim
- shown ~100,000 nodes

rule of thumb: extra order of magnitude in scale, with at least 10x the hardware and cost

SWANS

- **<u>S</u>calable <u>W</u>ireless <u>A</u>d hoc <u>N</u>etwork <u>S</u>imulator**
  - **runs standard Java network applications over simulated networks**
  - **can simulate networks of 1,000,000 nodes sequentially, on a single commodity uni-processor**
  - **uses hierarchical binning for efficient signal propagation**
  - **runs on top of JiST; SWANS is a JiST application**
  - **component-based simulation architecture written in Java**

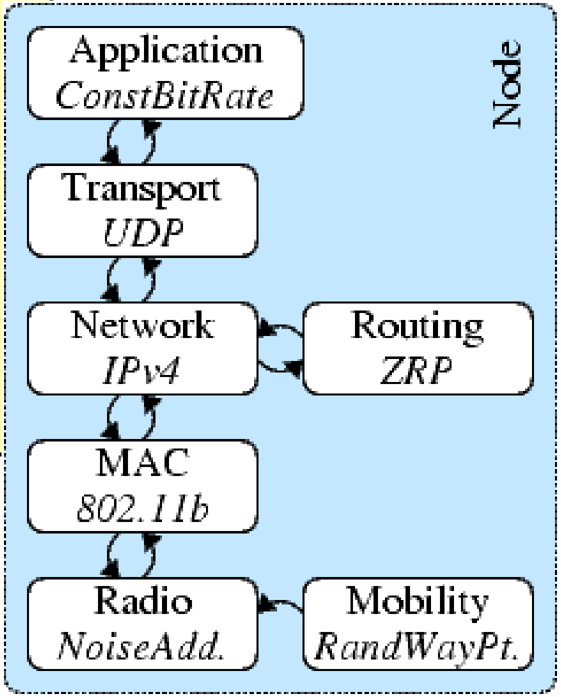




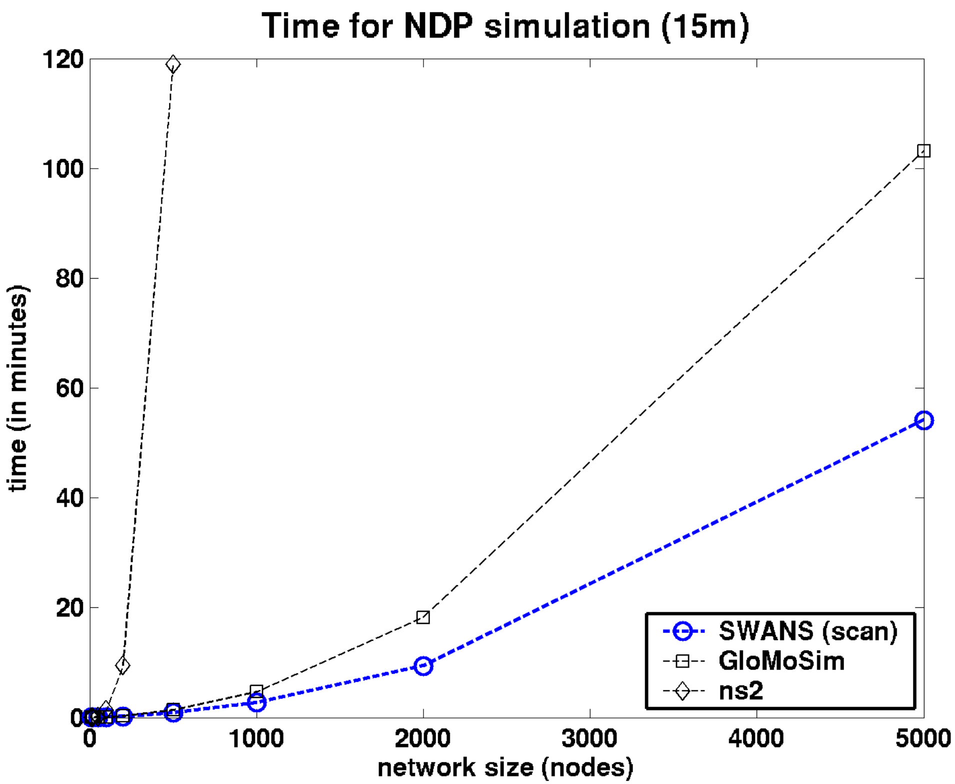
SWANS components

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

	files	classes	lines
JiST	25	92	11019
SWANS	65	143	16594
Other	24	49	3640
	114	**284**	**31253**



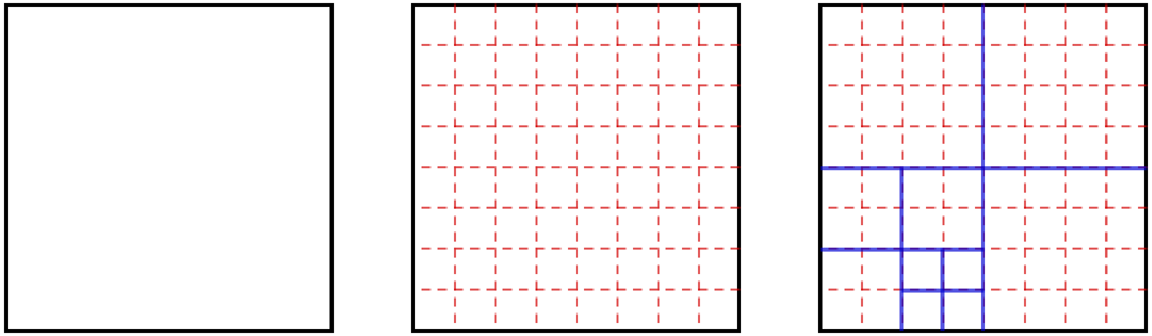
SWANS performance



- **simulation configuration**
  - **application** - **heartbeat neighbor discovery**
  - **field** - **5x5km^2; 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**

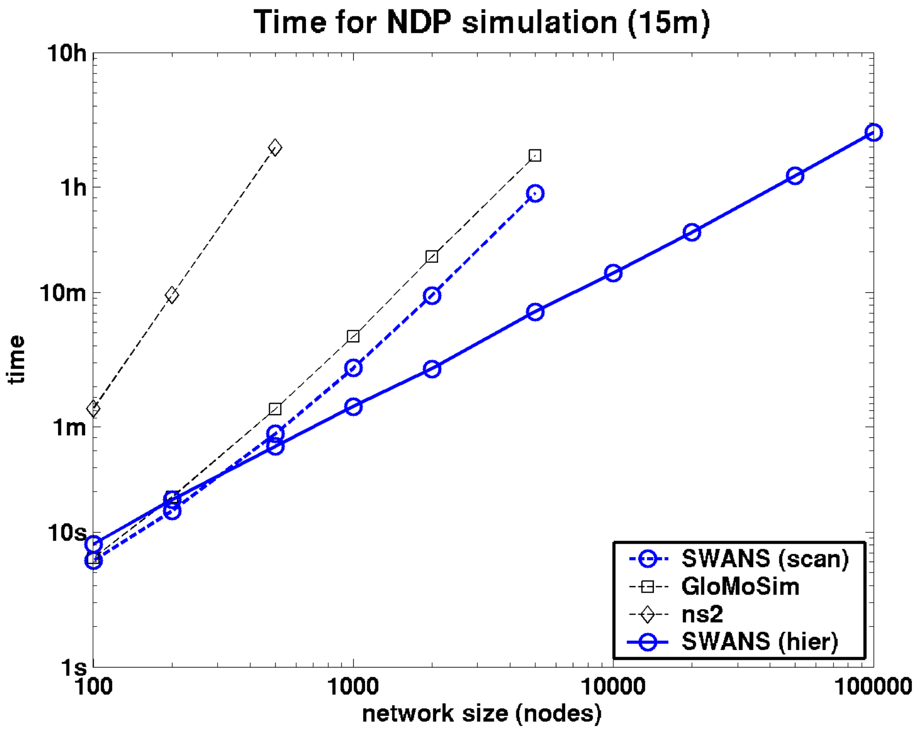
hierarchical binning

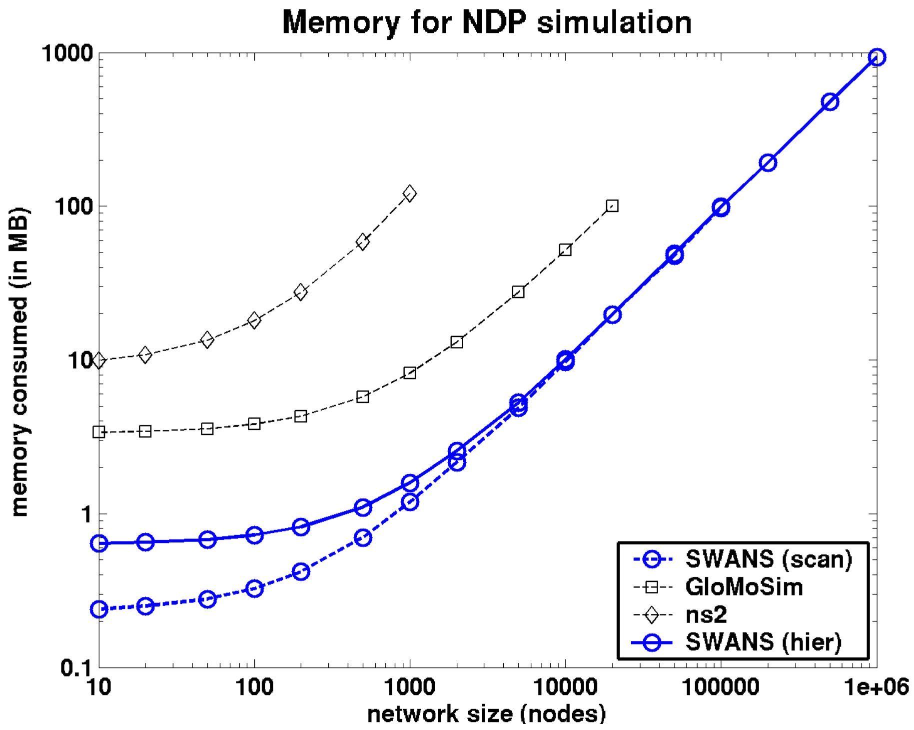
- simulating signal propagation
  - critical to performance and scalability
  - find radios within a given radius
  - prior approaches
    - linear scan — ns2
    - flat binning — GloMoSim, ns2′ (MSWiM '03)
    - function caching — SWiMNet (WN ’01), ns2′ (WSC '03)
  - hierarchical binning
    - location 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*

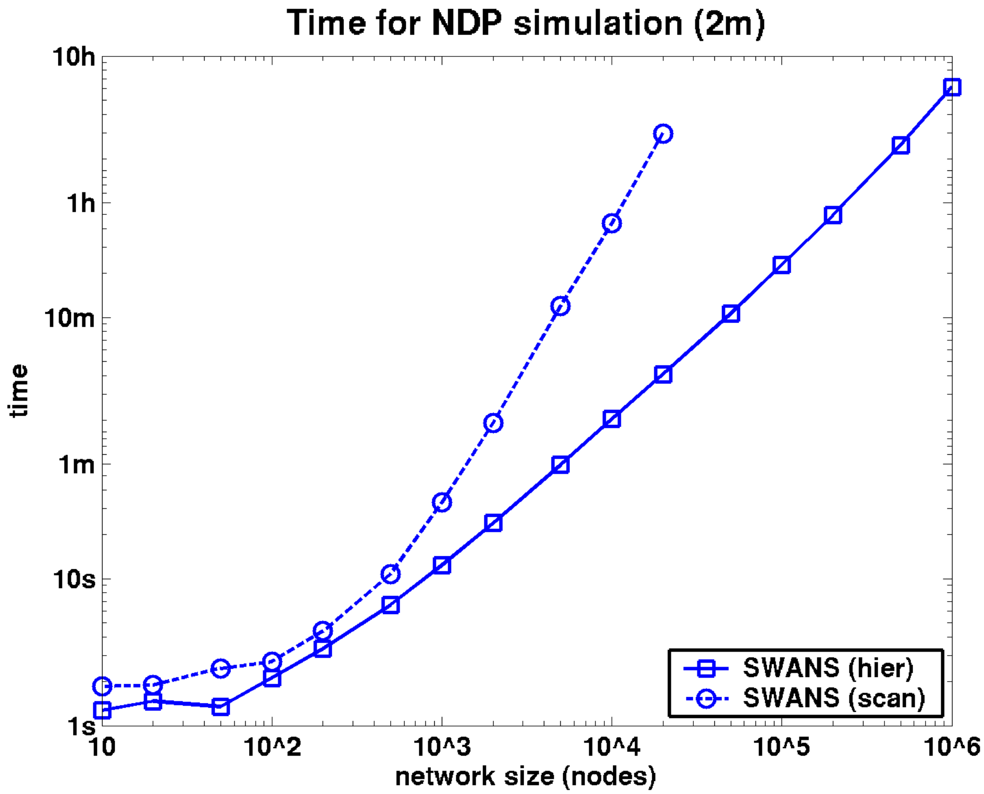
SWANS performance





t=15m	ns2		GloMoSim		SWANS		SWANS-hier	
nodes	**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**

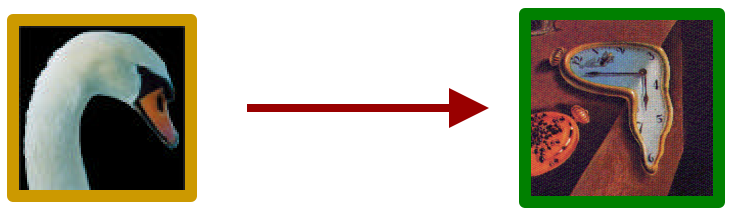
SWANS performance



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**

summary

- **SWANS scalability**
  - **hierarchical binning allows linear scaling with network size**
  - **can simulate million node wireless networks**
- **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**

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: throughput, memory
  - transparency
    - want to write a standard program in a standard language
    - with implicit optimization, concurrency, distribution, portability, fault-tolerance, 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

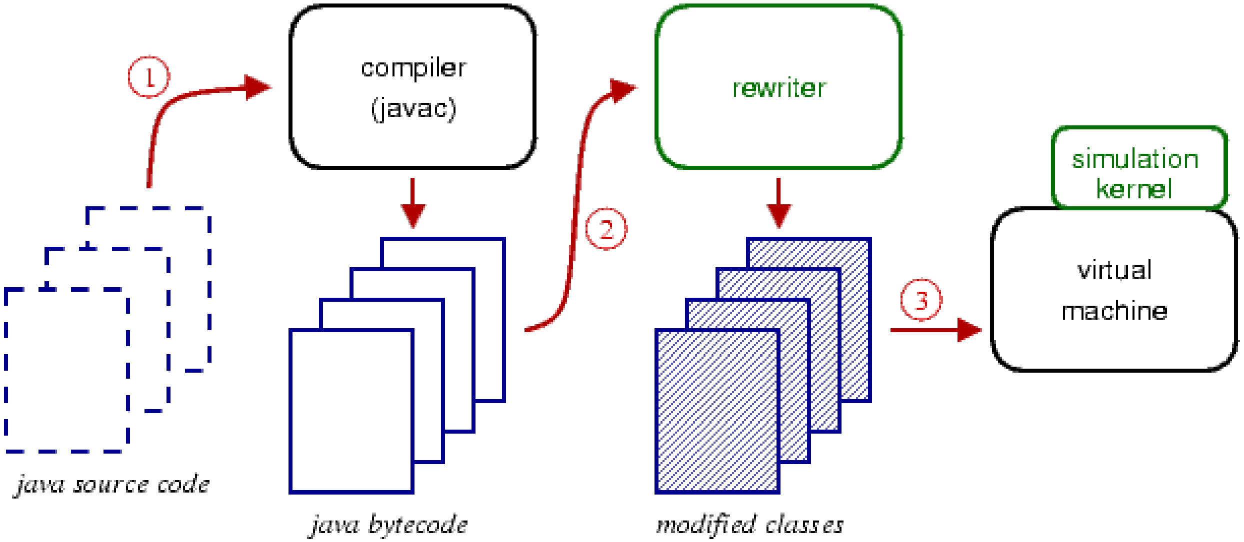
the jist approach

- JiST – Java in Simulation Time
  - convert 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

	kernel	library	language	JiST
transparent	✓		✓	✓
efficient		✓, ×	✓, ×	✓
standard	✓	✓		✓

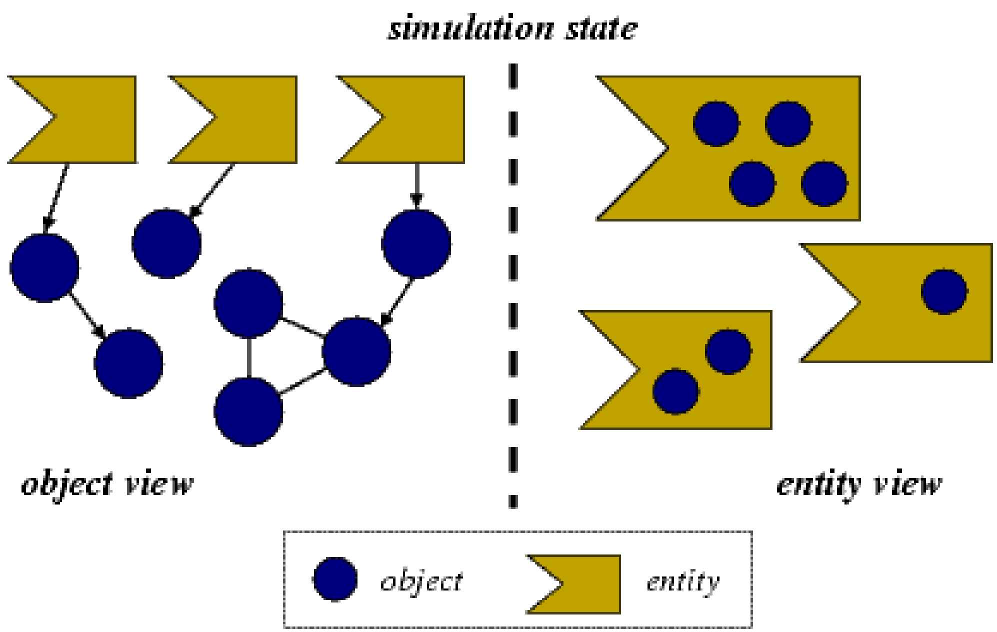
system architecture

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**
   - **progress of time is dependent on program progress**
   - **instructions take zero (simulation) time**
   - **time explicitly advanced by the program: `sleep(time)`**
3. **Rewritten program interacts with simulation kernel**



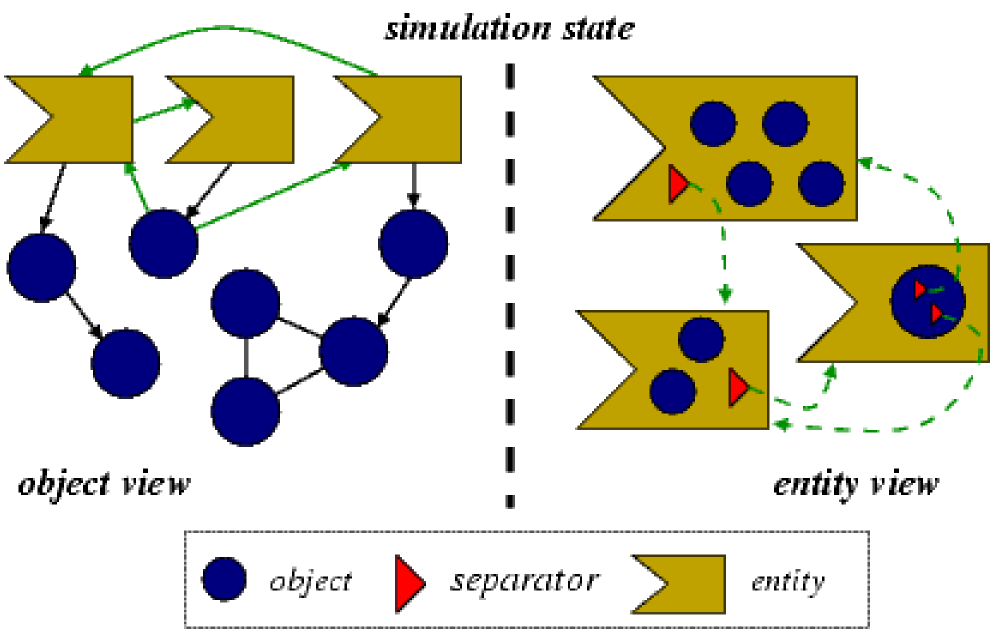
jist object model

- 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



jist api

- **`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 Proxiable               - tag object as proxiable
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()       - simulation time CSP Channel
void installRewrite(rewriter) – install transformation
EntityRef THIS                    - this entity reference
EntityRef ref(Entity e) - reference of an entity
// ... and more
```

*

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() );
  }
}
```

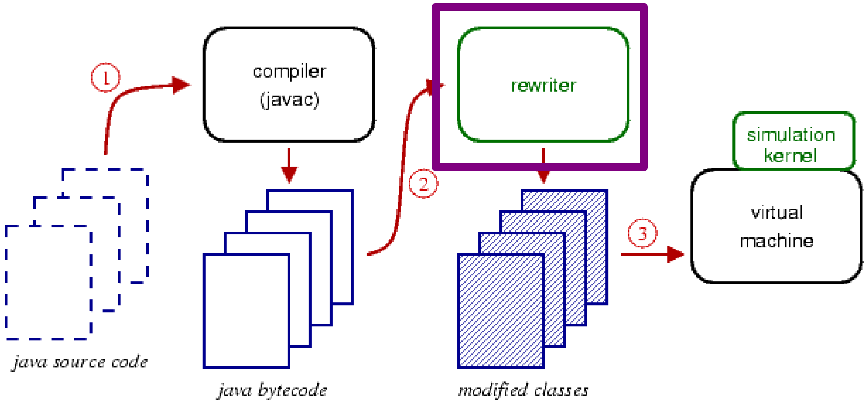
- demo!

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

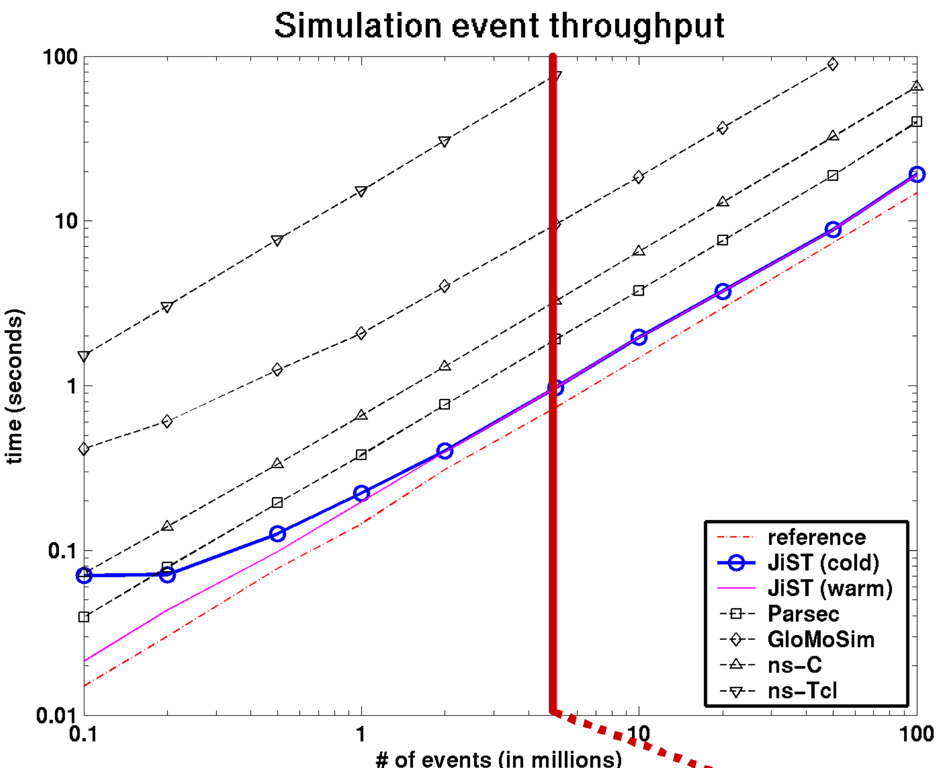
simulation time rewriter

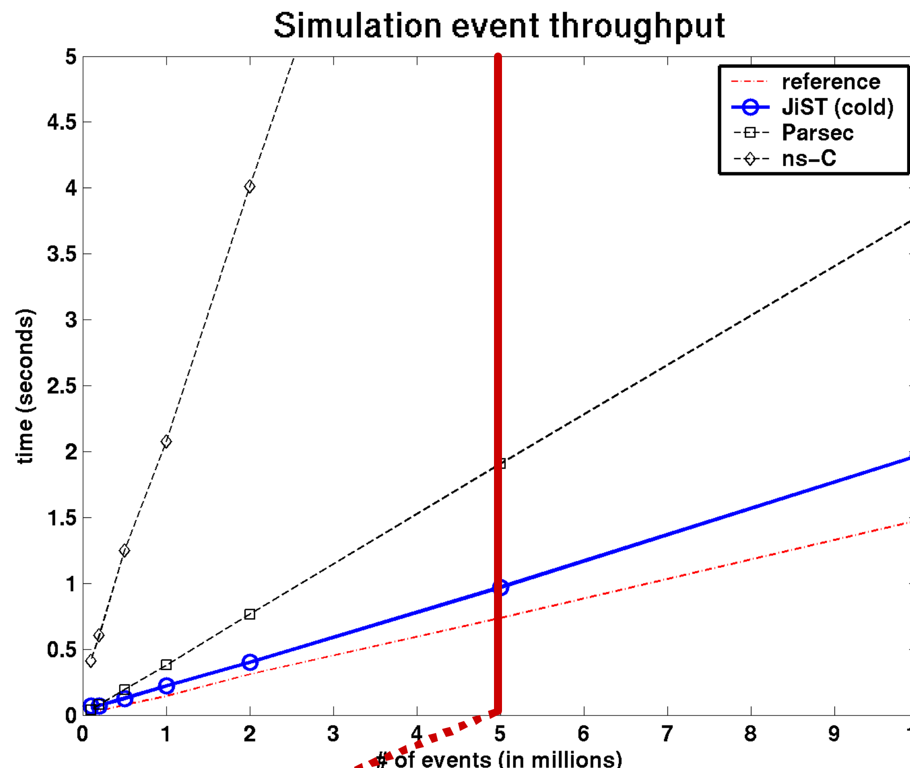
- **rewriter properties**
  - **dynamic class loader**
  - **no source code access required**
  - **operates on application packages, not system classes**
  - **uses Apache Byte Code Engineering Library (BCEL)**

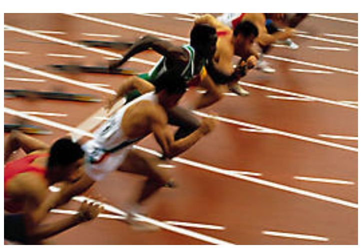
- **rewriting phases**
  - **application-specific rewrites**
  - **verification**
  - **add entity self reference**
  - **intercept entity state access**
  - **add method stub fields**
  - **intercept entity invocations**
  - **modify entity creation**
  - **modify entity references**
  - **modify typed instructions**
  - **continuable analysis**
  - **continuation transformation**
  - **translate JiST API calls**



jist performance: event throughput

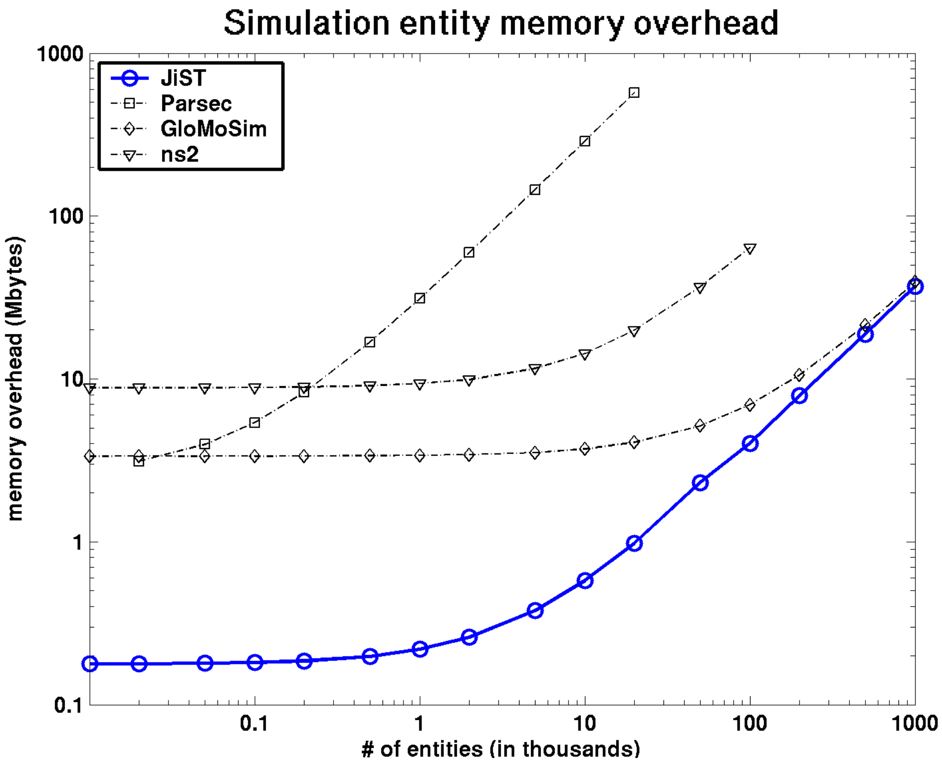


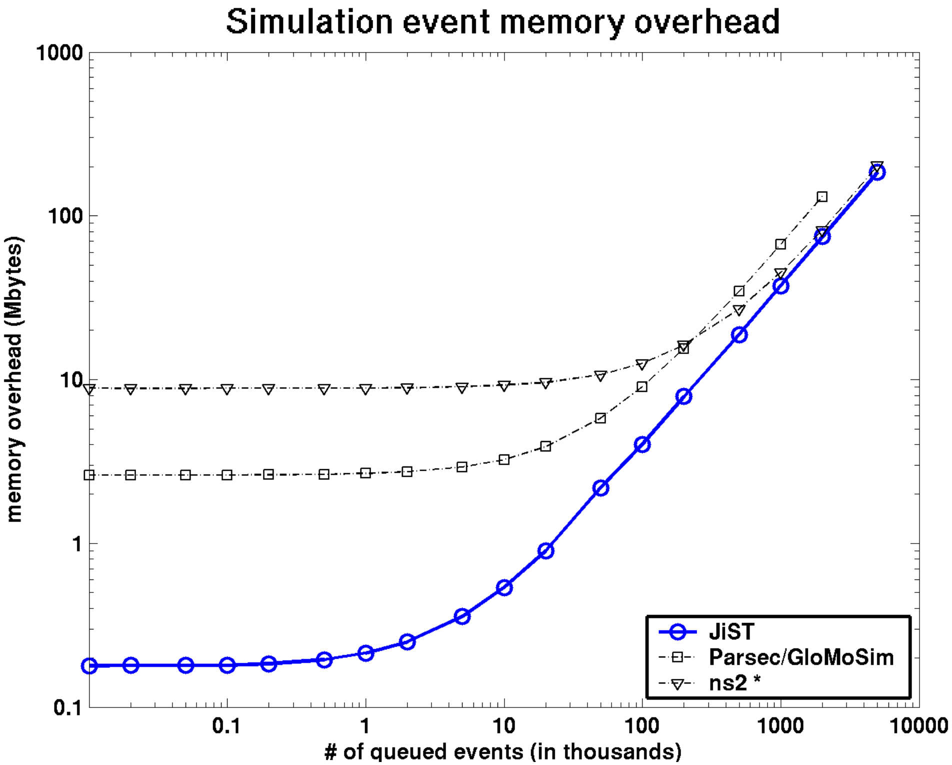




5x10^6 events	time (sec)	vs. reference	vs. JiST
reference	0.74		0.76x
JiST	**0.97**	1.31x	
Parsec	1.91	2.59x	**1.97x**
ns2-C	3.26	4.42x	**3.36x**
GloMoSim	9.54	12.93x	**9.84x**
ns2-Tcl	76.56	103.81x	**78.97x**

jist performance: 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

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

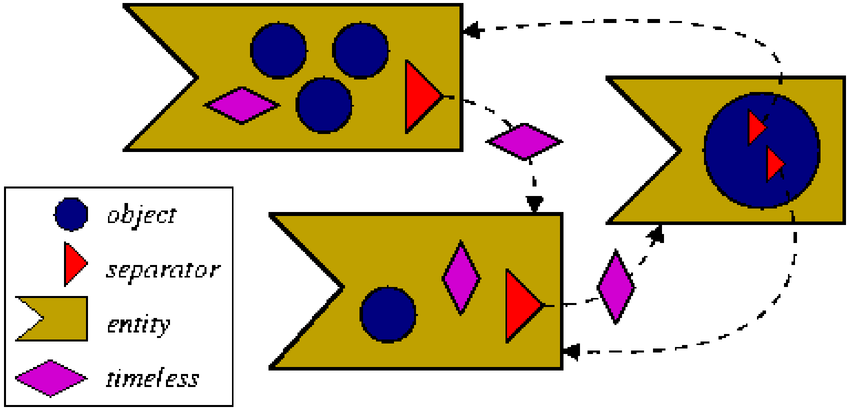
IPC	no context switch, no serialization, zero-copy
Java kernel	cross-layer optimization
rewriting	no source-code access required
distribution	provides a single system image abstraction
concurrency	model supports parallel and speculative execution

- hardware-oriented benefits

cost	COTS hardware and clusters
portability	runs on everything

zero-copy semantics

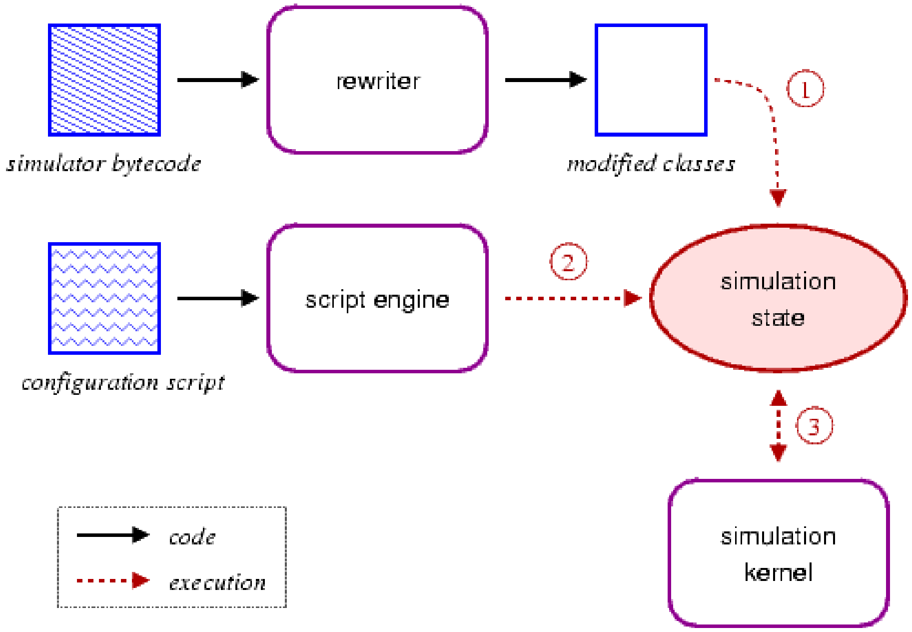
- timeless object: a temporally stable object
  - inferred statically as open-world immutable
  - or tagged explicitly with the `Timeless` interface
- benefits
  - pass-by-reference saves memory copy
    - zero-copy semantics for inter-entity communication
  - saves memory for common shared objects
    - 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**

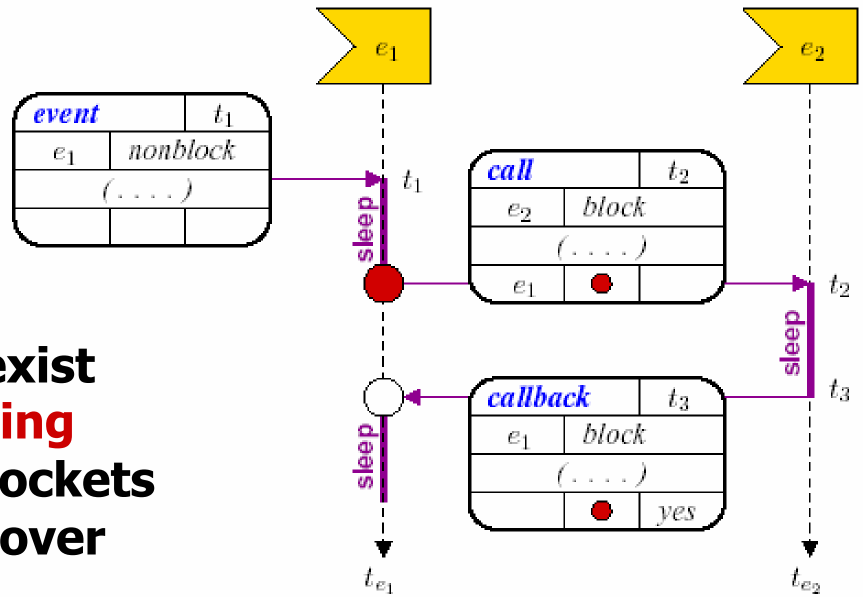
- **support for multiple scripting languages by reflection**
  - **no additional code**
  - **no memory overhead**
  - **no performance hit**
  - **Bsh - scripted Java**
Jython - Python
  - **Smalltalk, Tcl, Ruby, Scheme and JavaScript**



simulating with Java network 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

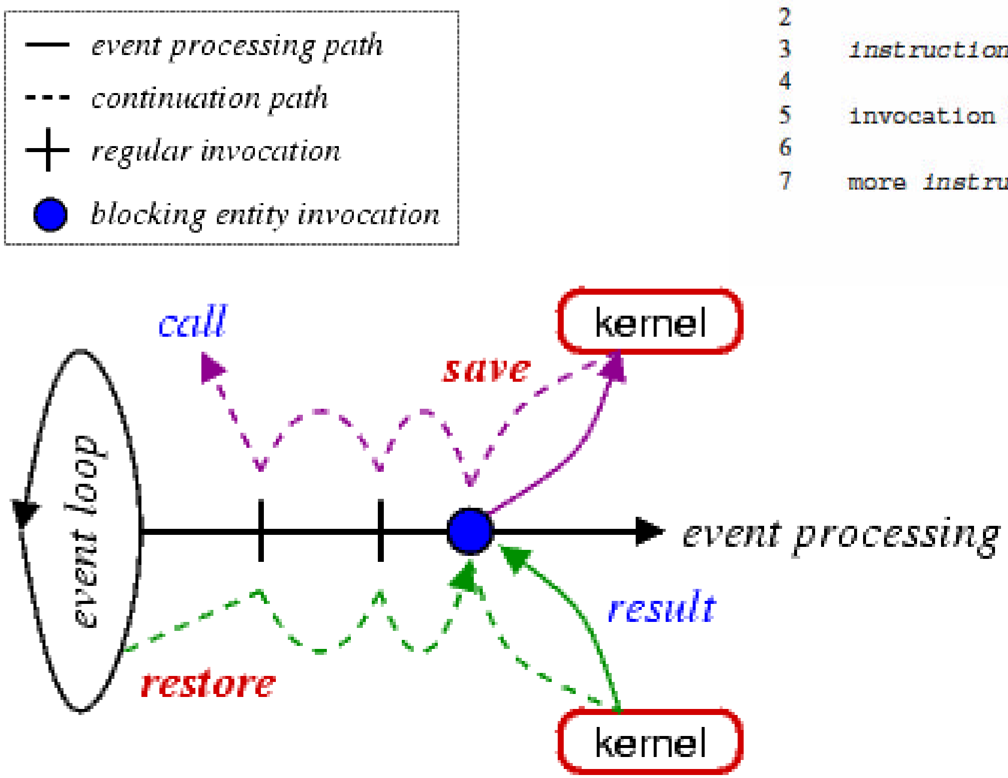


- benefits
  - no explicit process
  - blocking and non-blocking coexist
  - akin to **simulation time threading**
  - can build simulated network sockets
  - can **run standard applications** over these simulated sockets

*

capturing continuations

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



Before CPS transform:

```
METHOD continuable:

  instructions

  invocation BLOCKING

  more instructions
```

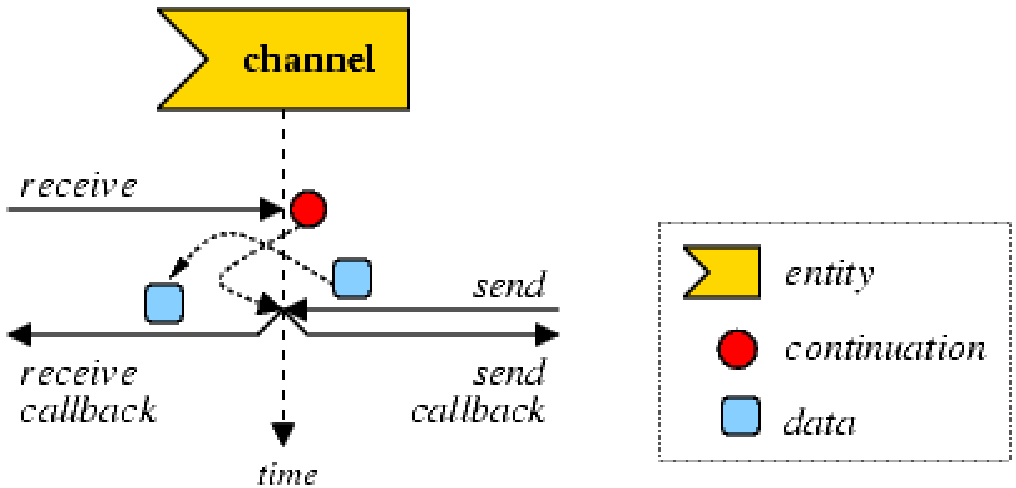
After CPS transform:

```
METHOD continuable:
  if jist.isRestoreMode:
    jist.popFrame f
    switch f.pc:
      case PC1:
        restoreLocals f.locals
        restoreStack f.stack
        goto PC1
      ...

  instructions

  setPC f.pc, PC1
  saveLocals f.locals
  saveStack f.stack
PC1:
  invocation BLOCKING
  if jist.isSaveMode:
    jist.pushFrame f
    return

  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, …**



rewriter flexibility

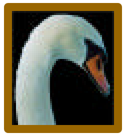
- **simulation time transformation**
  - extend Java object model with entities
  - extend Java execution model with events
  - language-based simulation kernel

- **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]**

summary

- **JiST – Java in Simulation Time**
  - convert virtual machine into simulation platform
  - 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 simulation research platform
  - merges systems-based and language-based approaches to simulator construction
    - efficient, transparent and standard



- **SWANS – Scalable Wireless Ad hoc Network Simulator**
  - built atop JiST, proof of concept
  - component-based framework
  - runs standard Java networking applications
  - uses hierarchical binning to perform signal propagation
  - scales to networks of a million nodes on a uni-processor

JiST –
Java in Simulation Time

for the

Scalable Simulation of Mobile Ad hoc Networks

THANK YOU.

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

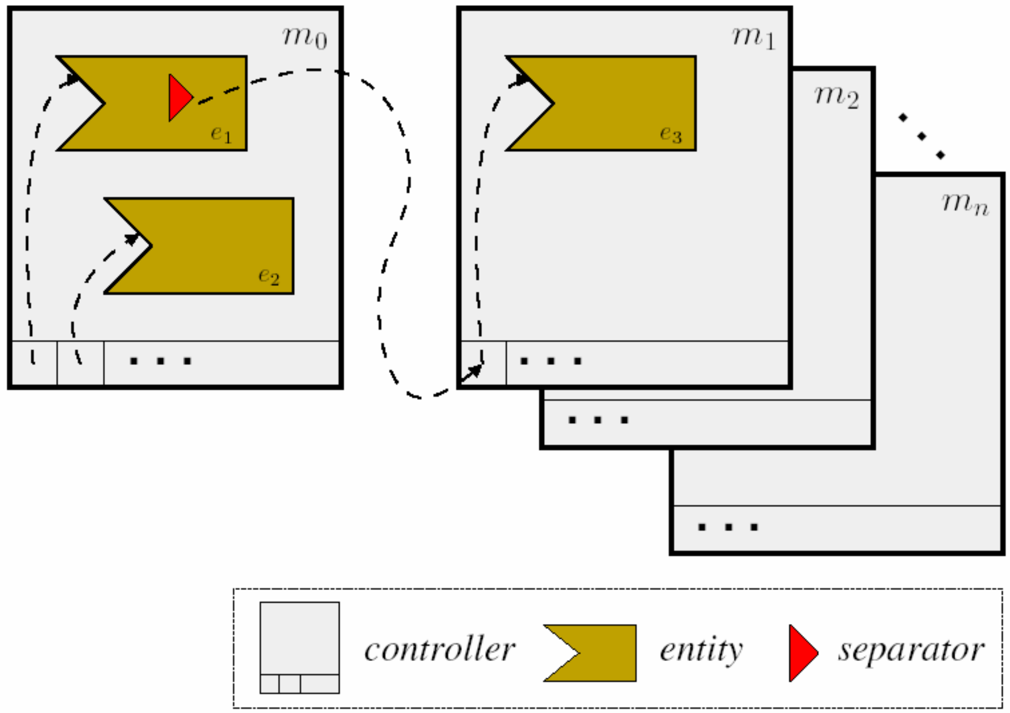
- **SWANS is a JiST application**
  - **entity invocation** **tracking time**
    no context switching; zero-copy; cross-layer optimizations; type-safety; implicit event structures and types
  - **timeless objects** **packets**
    saves memory; simplifies memory management
  - **proxy entities** **network 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

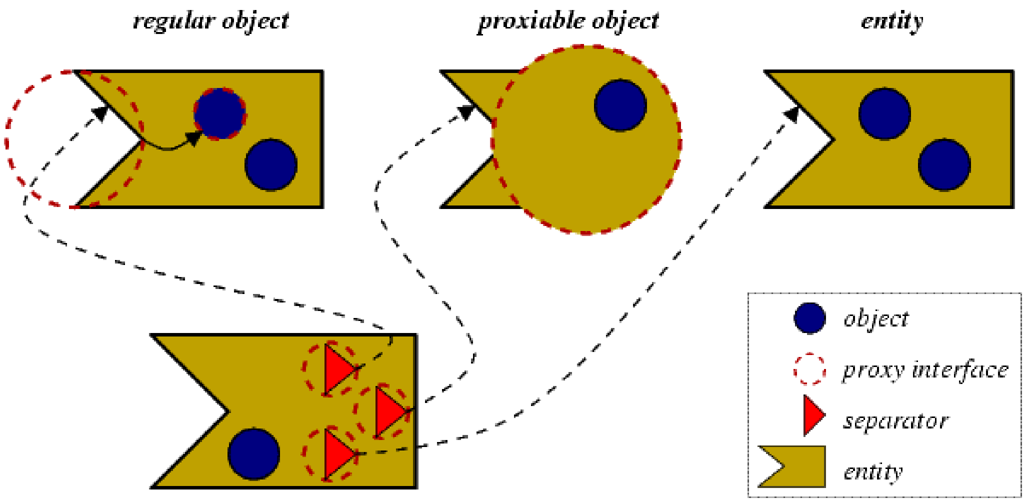
distribution and concurrency

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



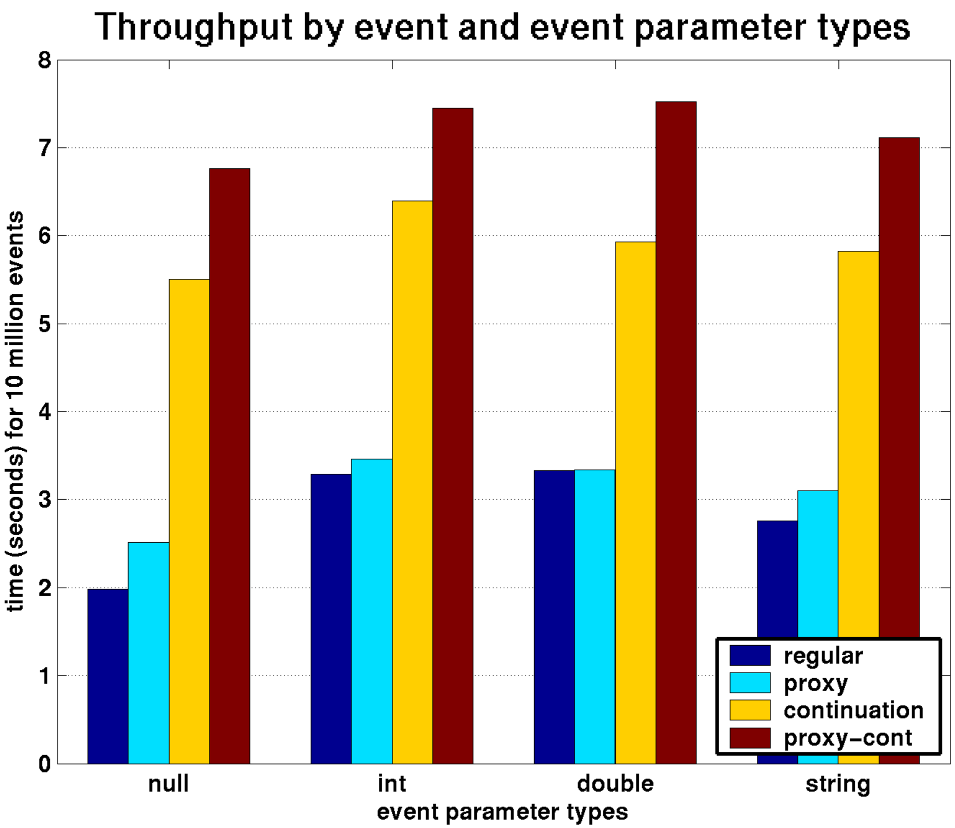
proxy entities

- proxy entities **relay events to a target**
  - possible targets: regular object, **proxiable** object, entity
  - proxiable: any object tagged with `Proxiable` 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



*

java deficiencies

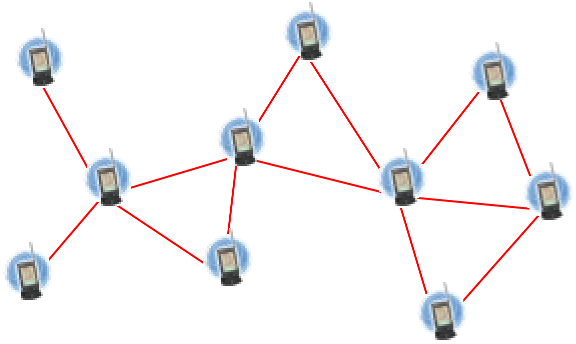


- manually need to box Java primitive types
- tail invocations not properly detected
- need API for type-safe stack access
- exceptions are very expensive

simulating MANETs

why simulate ad hoc networks?

- **scale:**
  - large number of nodes
  - expensive to own, maintain, charge...
  - distribution of control
  - aggregation of experimental data
  - node mobility
  - isolating experiment from interference
- **complexity:**
  - simple protocols vs.
    aggregate network behavior
  - repetition



jist api

JistAPI.java

```java
package jist.runtime;

public class JistAPI
{
 public static interface Entity { }
 public static class Continuation extends Error { }
 public static interface Timeless { }

 public static long getTime() { ... }
 public static void sleep(long n) { }
 public static void end() { }
 public static void endAt(long t) { }

 public static JistAPI.Entity THIS;
 public static EntityRef ref(Entity e) { ... }

 public static interface Proxiable { }
 public static Object proxy(Object proxyTarget, Class proxyInterface) { ... }
 public static Object proxyMany(Object proxyTarget, Class[] proxyInterface) { ... }

 public static final int RUN_CLASS = 0;
 public static final int RUN_BSH = 1;
 public static final int RUN_JPY = 2;
 public static void run(int type, String name, String[] args, Object properties) { }

 public static Channel createChannel() { ... }

 public static void setSimUnits(long ticks, String name) { }

 public static interface CustomRewriter {
  JavaClass process(JavaClass jcl);
 }
 public static void installRewrite(CustomRewriter rewrite) { }
}
```

example: hello world

```java
import jist.runtime.JistAPI;

class hello implements JistAPI.Entity
{
  public static void main(String[] args)
  {
    System.out.println("simulation start");
    hello h = new hello();
    h.myEvent();
  }

  public void myEvent()
  {
    JistAPI.sleep(1);
    myEvent();
    System.out.println("hello world, t="
      +JistAPI.getTime());
  }
}
```

example: scripts

hello.bsh

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

BeanShell – scripted Java

hello.jpy

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

Jython – Python

*

example: proxy entities

```
import jist.runtime.JistAPI;

public class proxy
{
  public static interface myInterface extends JistAPI.Proxiable
  {
    void myEvent();
  }

  public static class myEntity implements myInterface
  {
    private myInterface proxy =
      (myInterface)JistAPI.proxy(this, myInterface.class);
    public myInterface getProxy() { return proxy; }

    public void myEvent()
    {
      JistAPI.sleep(1);
      proxy.myEvent();
      System.out.println("myEvent at t="+JistAPI.getTime());
    }
  }

  public static void main(String args[])
  {
    myInterface e = (new myEntity()).getProxy();
    e.myEvent();
  }
}
```

*

example: blocking methods

```
import jist.runtime.JistAPI;

public class cont implements JistAPI.Entity
{
  public void blocking() throws JistAPI.Continuation
  {
    System.out.println("called at t="+JistAPI.getTime());
    JistAPI.sleep(1);
  }

  public static void main(String args[])
  {
    cont c = new cont();
    for(int i=0; i<3; i++)
    {
      System.out.println("i="+i+" t="+JistAPI.getTime());
      c.blocking();
    }
  }
}
```

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	15340	(5.8%)	6817	(5.6%)	7223	(63.6%)	29380	(7.4%)
code, etc.	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.