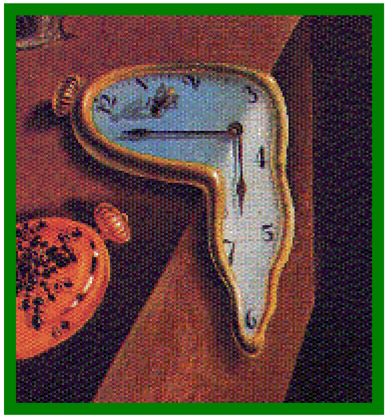


JiST: Java in Simulation Time

for

Scalable Simulation of Mobile Ad hoc Networks (MANETs)

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

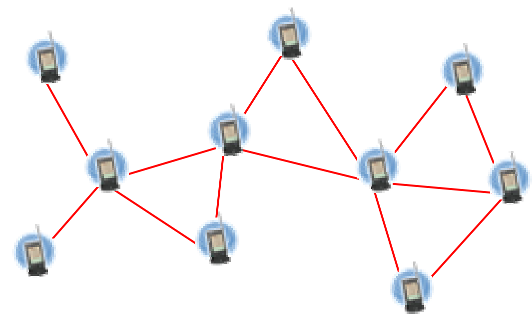
WNL meeting
24 September 2003

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

simulation scalability

- **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**

  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**
- **Sensor networks, smart dust, Ubicomp**
  - **Many thousands of wireless devices in environment**

Simulation scalability is important

what is a simulation?

- **discrete event simulations**
  - **state of the simulated world**
  - **discrete events in simulated time**
  - **discretized simulation model**
  - **temporally ordered event queue, event loop**
  - **work through simulation time as quickly as possible**
- **desirable properties**
  - **correctness**
    - **valid simulation results**
  - **efficiency**
    - **performance: throughput, memory**
  - **transparency**
    - **implicit optimization, concurrency, distribution, portability, robustness, fault-tolerance**

a *brief* history of simulation

- **unstructured simulation: computers compute**
- **structured: event-oriented vs. process-oriented**

systems	***languages***

systems

- **TimeWarp OS**
  - **processes run in virtual time**
  - **control scheduling, IPC**

  👍 **transparency** 👎 **efficiency**
- **simulation libraries**
  - **move functionality to user-space for performance**
  - **usually event-oriented**

  👎 **transparency** 👍 **efficiency**

languages

- **Simula**
  - **entities, messages**
  - **event-oriented**
- **Parsec (latest)**
  - **C-like language**
  - **process-oriented (logically)**
  - **simulation time concurrency**

◇? **transparency** ◇? **efficiency**

👍👎 **new language**

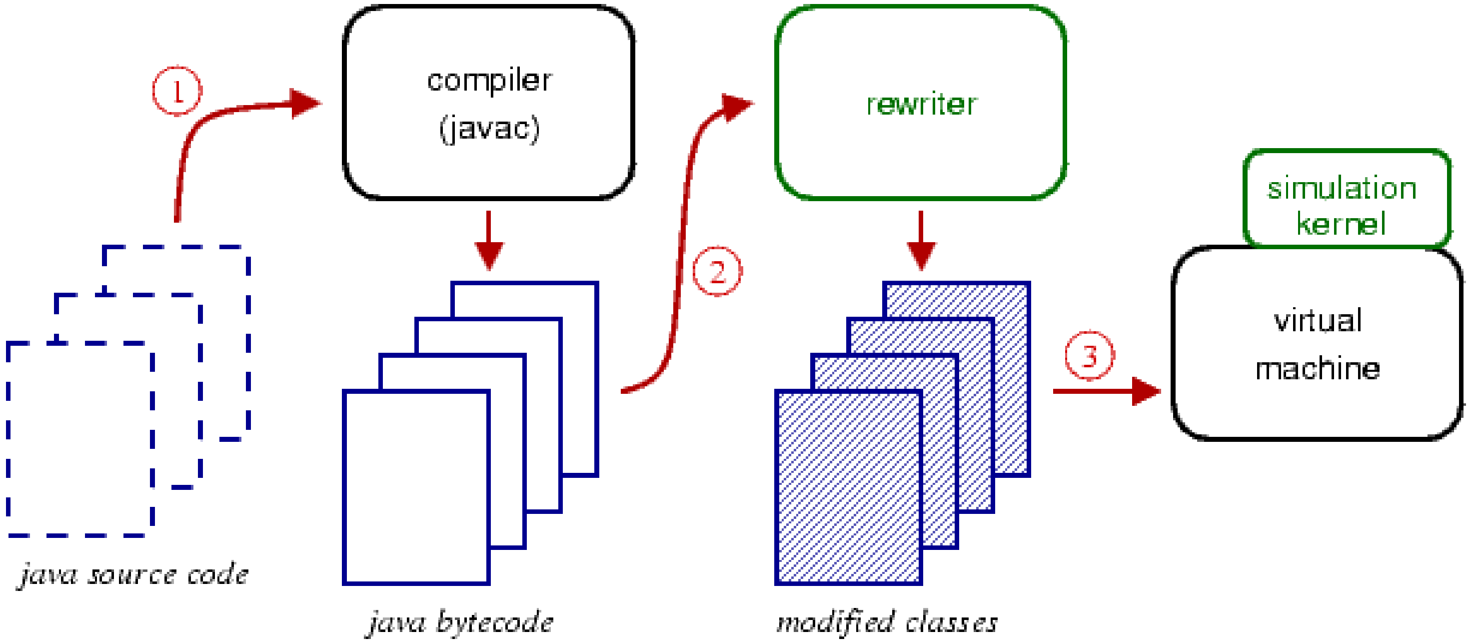
virtual machines

the jist approach

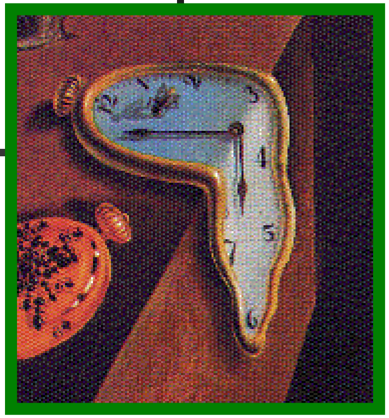
- **JiST – Java in Simulation Time**
  - converting a virtual machine into a simulation platform
  - no new language, no new library, no new runtime
  - merges modern language and simulation semantics
    - combines systems-based and languages-based approaches
- **overview**
  - system architecture
  - simulation time transformation
  - and more: timeless objects, proxy entities, reflection, debugging, continuations, concurrency, distribution
  - applications
    - SWANS – Scalable Wireless Ad hoc Network Simulator
  - conclusion

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.
3. **Rewritten program interacts with simulation kernel.**



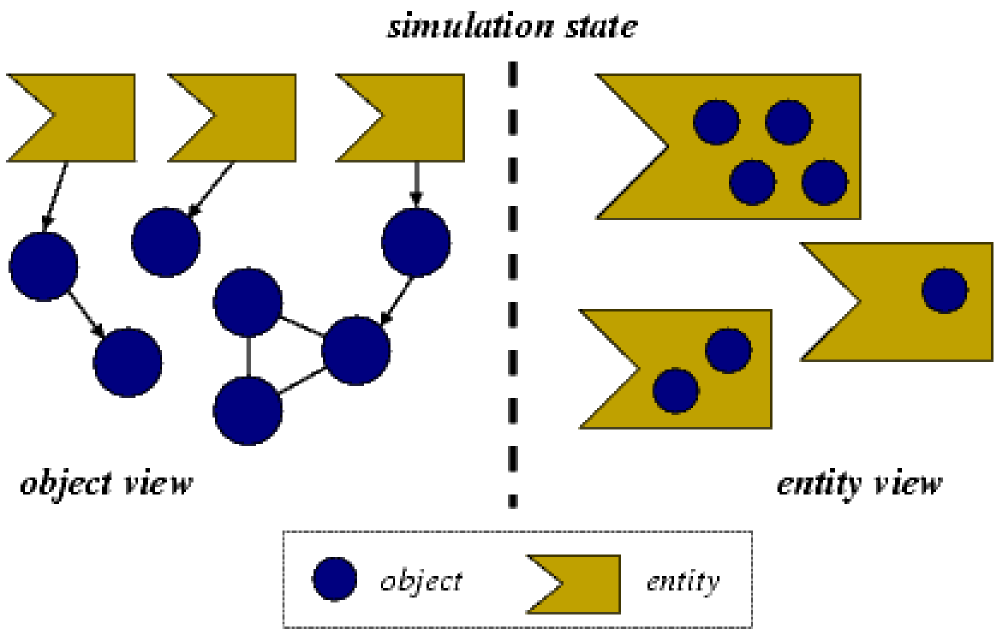
simulation time



- **actual time**
  - **progress of program *independent* of time**
- **real time**
  - **progress of program is *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`**
  - **simulation event loop embedded in virtual machine**

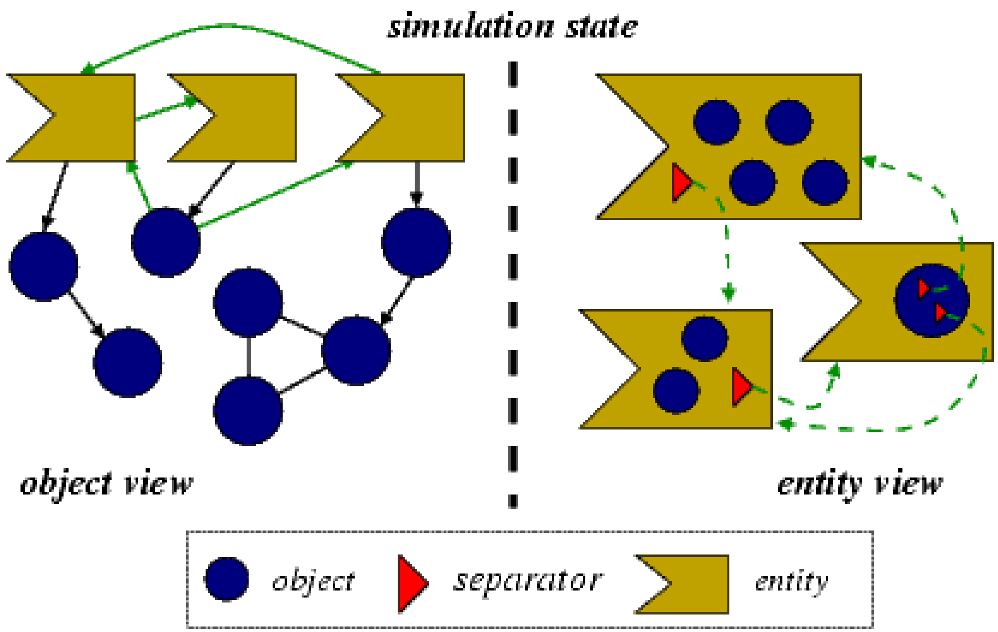
extended object model

- program state contained in **objects**
- objects contained in **entities**
  - each entity runs at its own simulation *time*
  - as with objects, entities do not share state
  - think of an entity as a simulation component



extended execution semantics

- entity references replaced with **separators**
  - event channels; act as **state-time boundary**
- entity methods are an event interface
  - **simulation time invocation**
  - **non-blocking**; invoked at caller entity time; no continuation



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.`

*

jist api

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

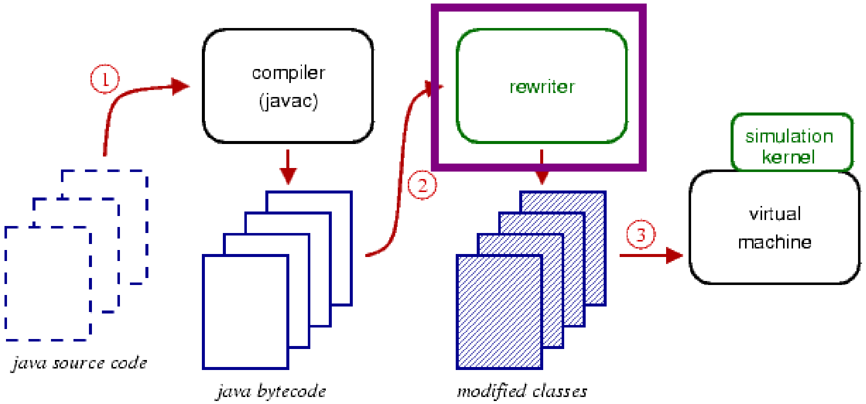
```
// used in hello example
interface Entity               - tag object as entity
long getTime()                 - return simulation time
void sleep(long ticks)         - advance simulation time

// others, to be introduced shortly
interface Timeless             - tag object as timeless
interface Proxiable            - tag object as proxiable
EntityRef THIS                 - this entity reference
EntityRef ref(Entity e)        - reference of an entity
... and a few more
```

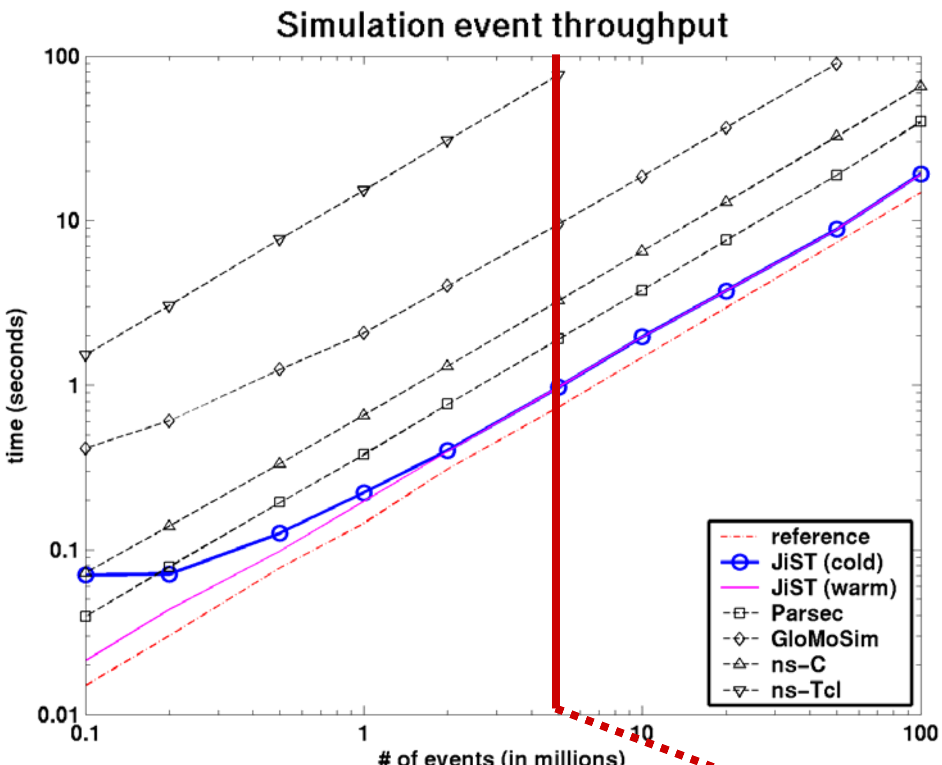
*

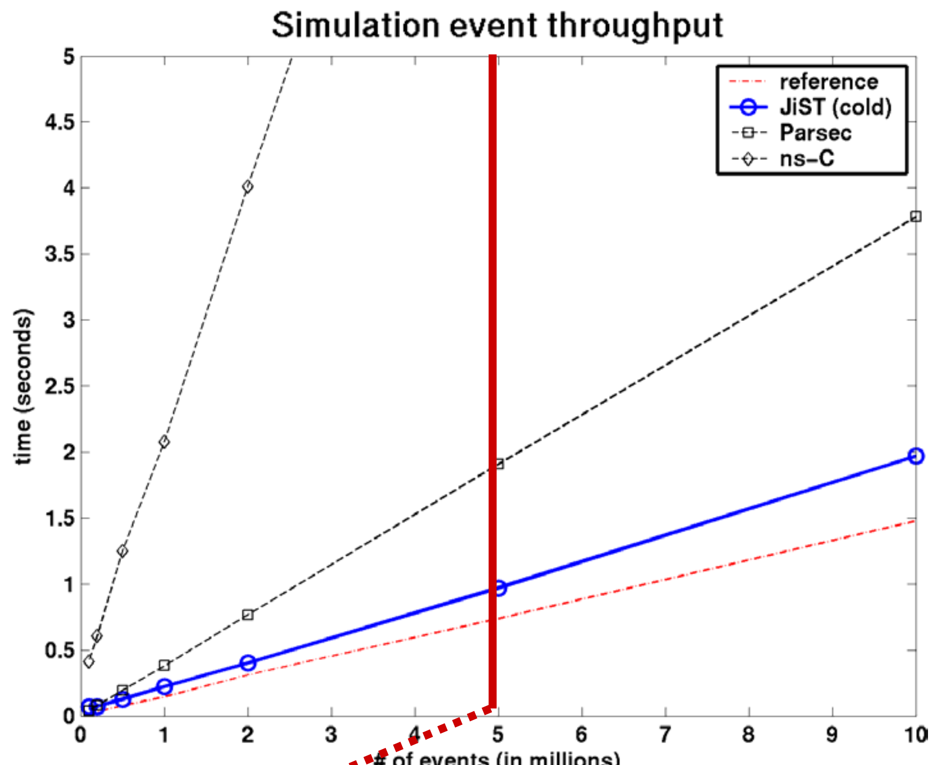
simulation time rewriter

- dynamic class loader
  - no source code access
  - uses Apache Byte Code Engineering Library (BCEL)
  - ignores non-application packages
- rewriting phases
  - 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
  - translate JiST API calls



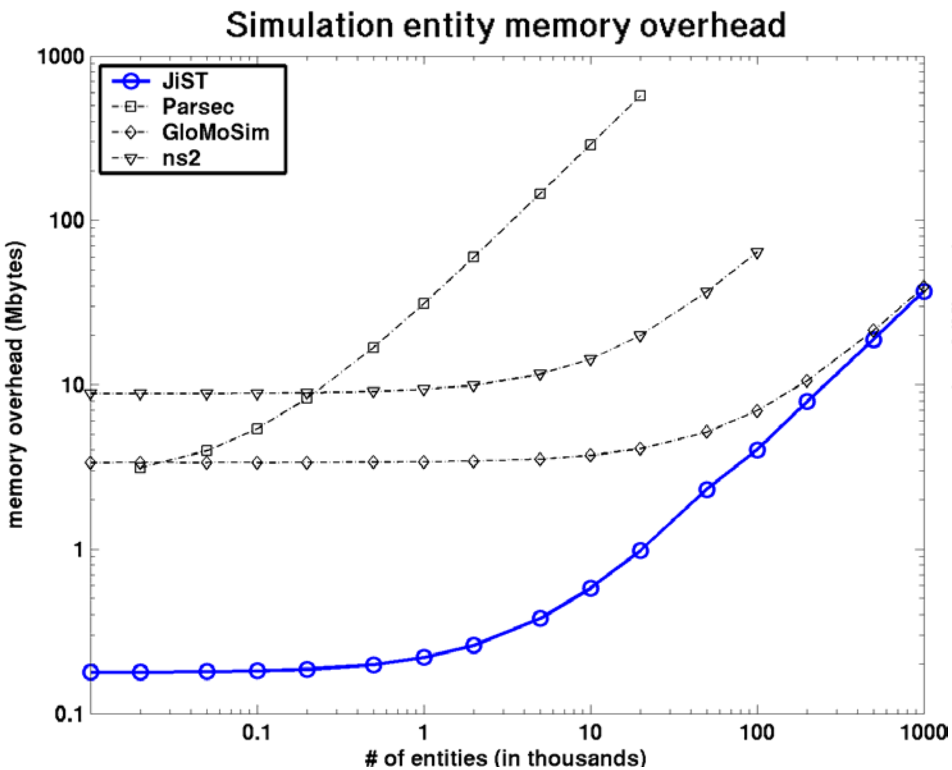
evaluation: event throughput

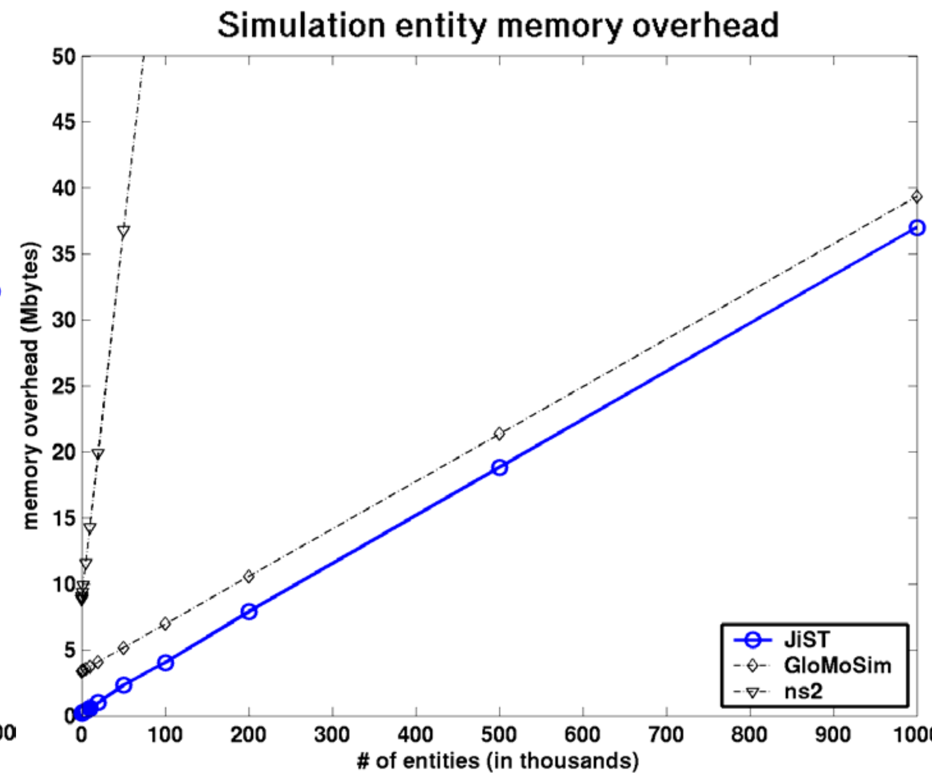




5x10^6 events	time (sec)	vs. reference	vs. JiST
reference	0.74		0.76x
JiST	**0.97**	1.31x	
Parsec	1.91	2.59x	1.97x
ns2-C	3.26	4.42x	3.36x
GloMoSim	9.54	12.93x	9.84x
ns2-Tcl	76.56	103.81x	78.97x

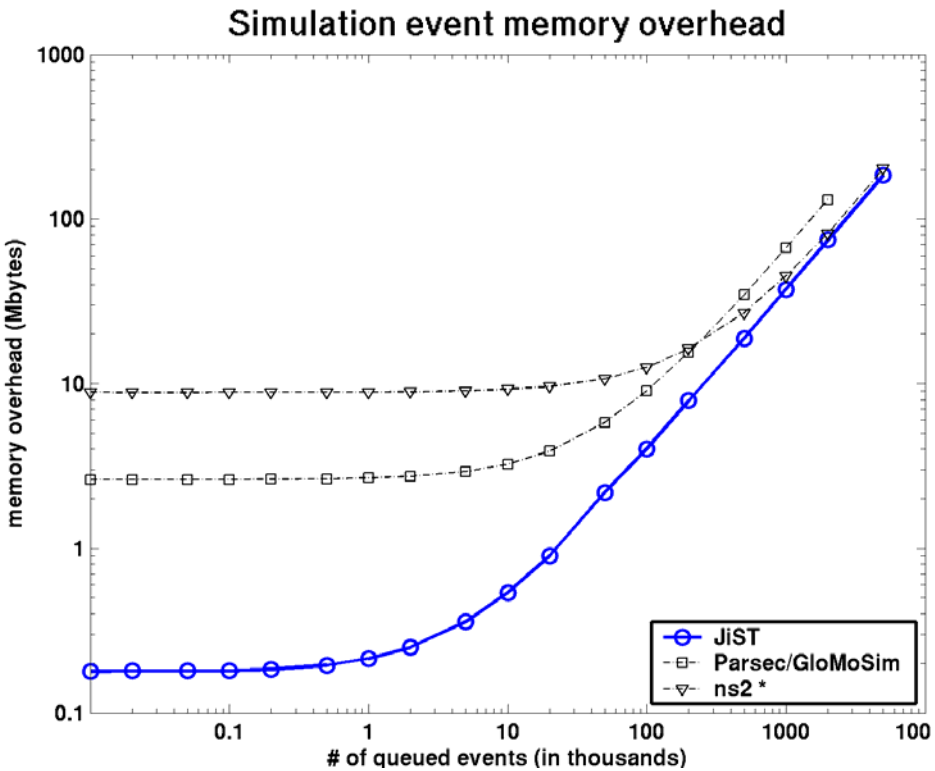
evaluation: memory overhead of entities

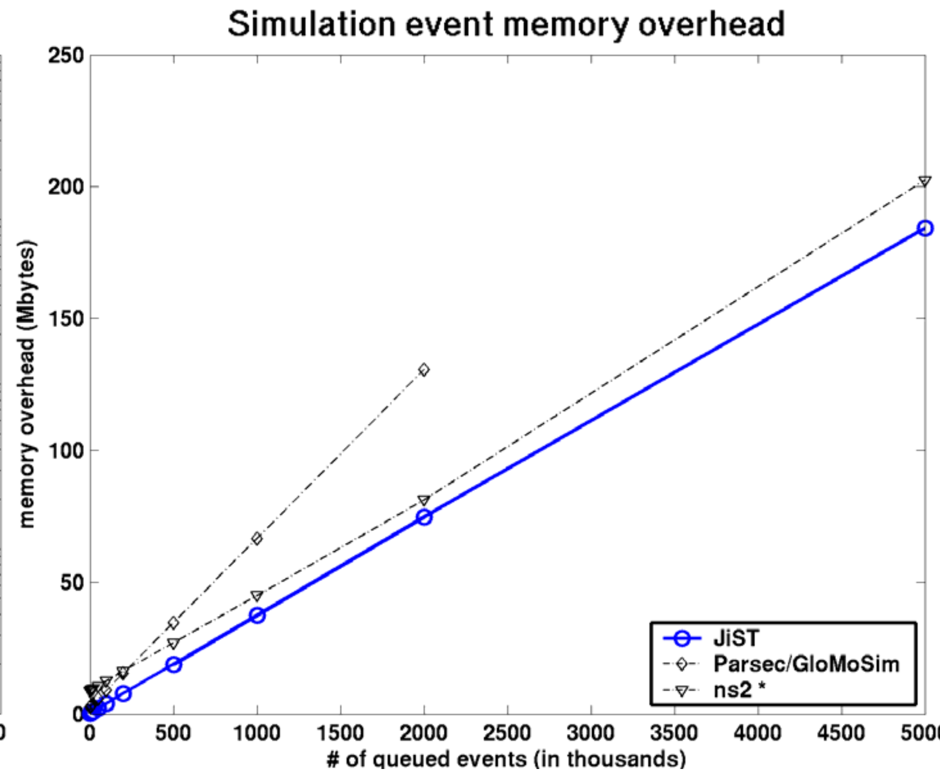




memory	per entity	per event	10K nodes sim.
JiST	**36 B**	**36 B**	**21 MB**
GloMoSim	36 B	64 B	35 MB
ns2 *	544 B	40 B	74 MB
Parsec	28536 B	64 B	2885 MB

evaluation: memory overhead of events





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

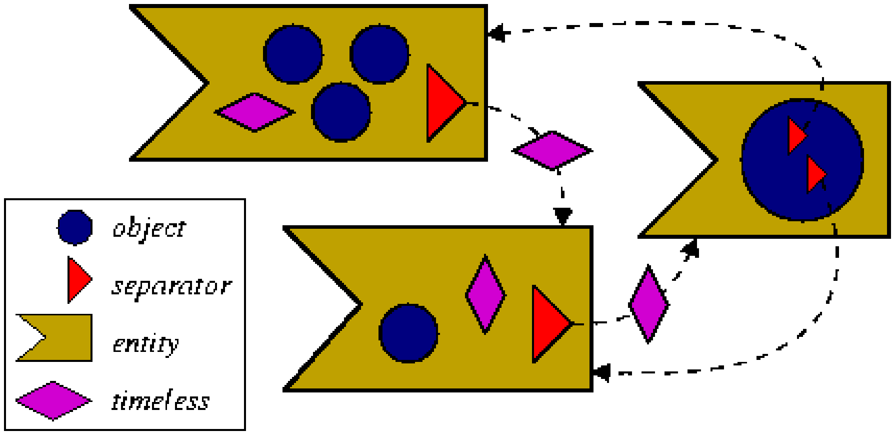
extensions to the model

- **entities**: extend object model and execution semantics
- **simulation time invocation**: event-based invocation

- **timeless objects**: pass-by-reference to avoid copy
- **proxy entities**: interface-based entity creation
- **reflection**, **tight event coupling**: scripting, debugging
- **continuations**: call and callback, blocking methods
- **simulation time concurrency**: threads, channels…
- **distribution**: location independence of entities
- **parallelism**: concurrent and speculative execution

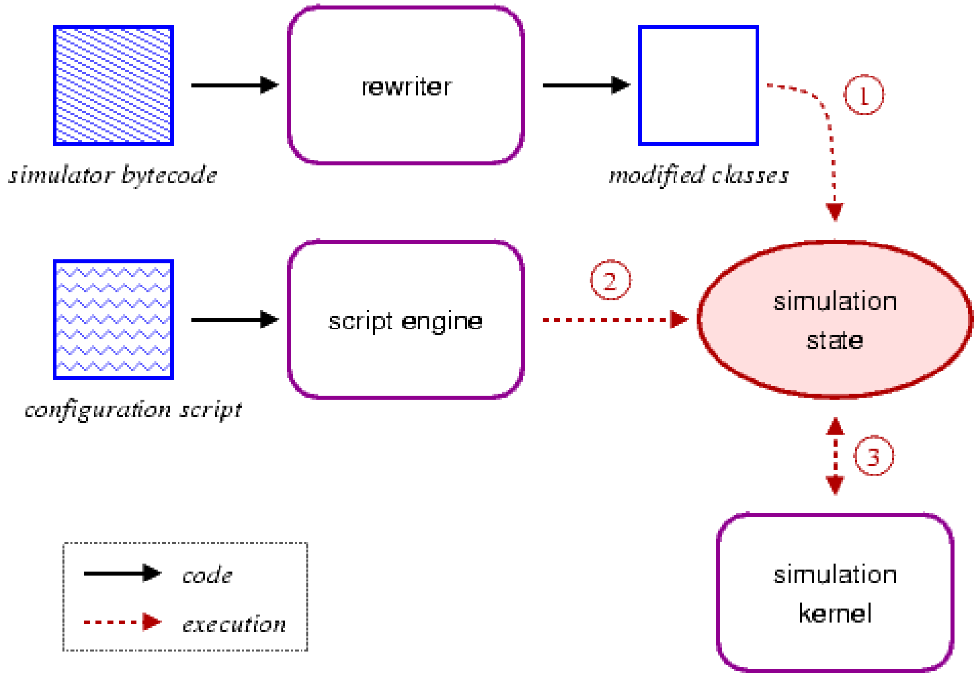
timeless objects

- timeless object: a temporally stable object
  - can be inferred statically as open-world immutable
  - or tagged explicitly with the `Timeless` interface
- benefits
  - pass-by-reference saves memory copy
  - saves memory for common shared objects; e.g. packets
    - can even substitute `hashcons` for `new` of common types



reflection

- **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**
  - reflection-based
  - no additional code
  - no memory overhead
  - no performance hit
  - Bsh - scripted Java
    Jython - Python
  - Bistro - Smalltalk
    Jacl - Tcl
    JRuby - Ruby
    Kawa - Scheme
    Rhino - JavaScript

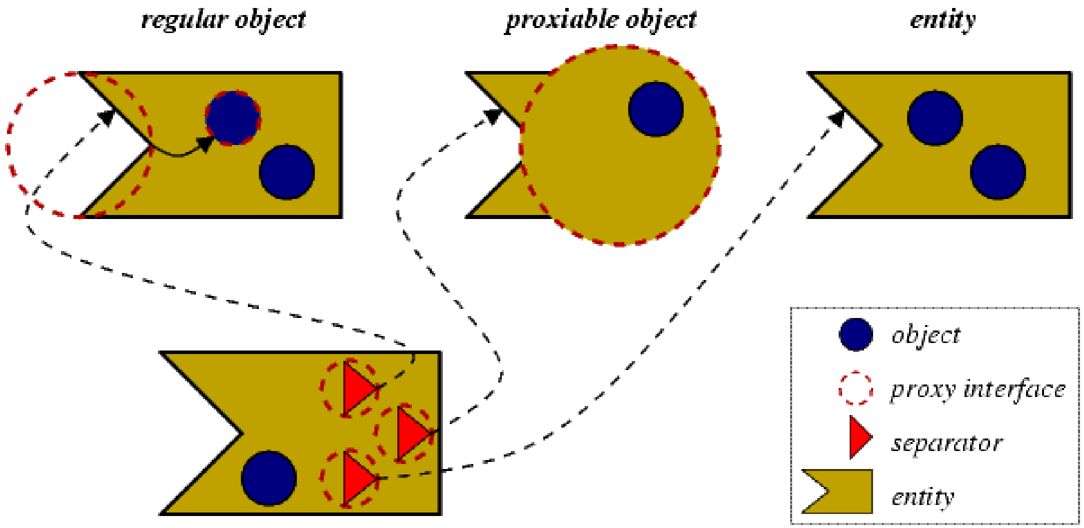


tight event coupling

- tight coupling of event dispatch and delivery provides 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

proxy entities

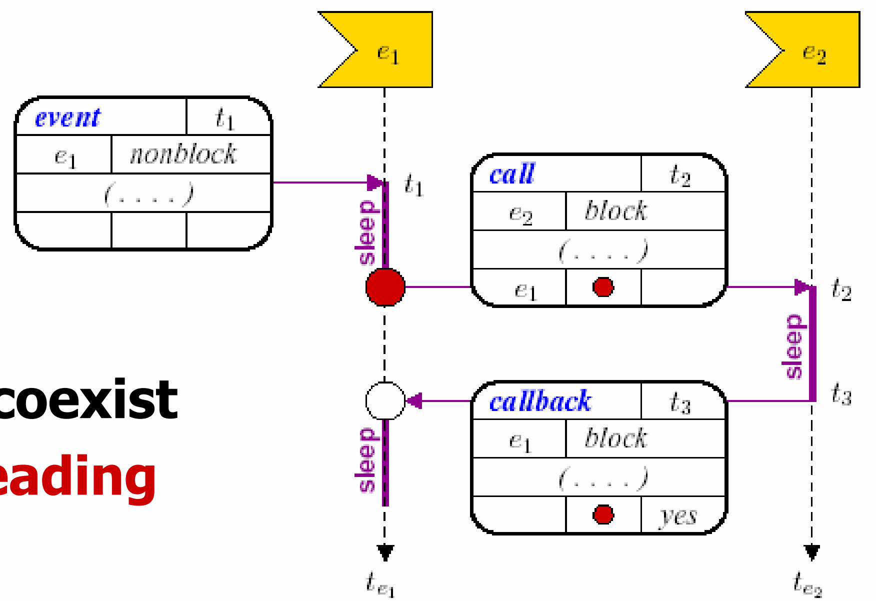
- proxy entities **relay events to a target**
  - possible targets: regular object, proxiable object, entity
- benefits
  - **equivalent performance**: `JistAPI.proxy(target, intfce)`
  - **interface-based**: does not interfere with object hierarchy
  - mix simulation time invocations with regular invocations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



*

continuations

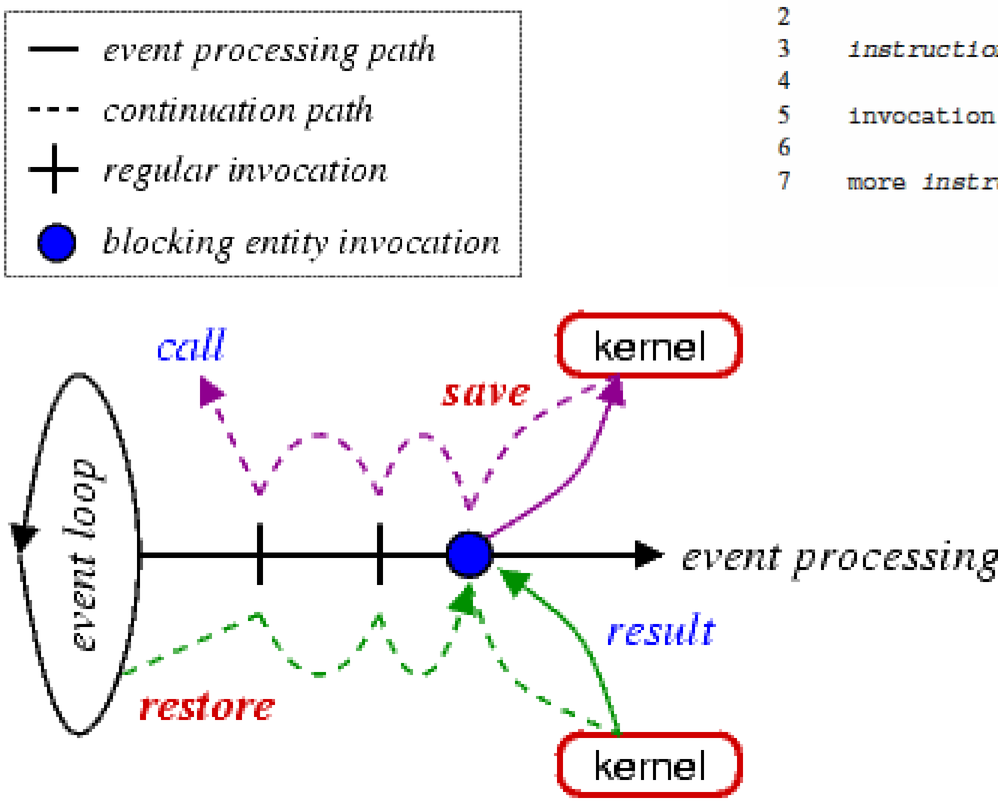
- simulation time entity method invocation
  - can easily write event-driven entities
  - what about process-oriented simulation?
- blocking methods
  - an entity method that declares a `Continuation` exception
  - event processing frozen at invocation
  - continues after call event completes, at some later simulation time
- benefits
  - no explicit process
  - blocking and non-blocking coexist
  - akin to simulation time threading



*

continuations: implementation

- **saving and restoring the frame is non-trivial in Java!**



Before CPS transform:

```
METHOD continuable:

  instructions

  invocation BLOCKING

  more instructions
```

After CPS transform:

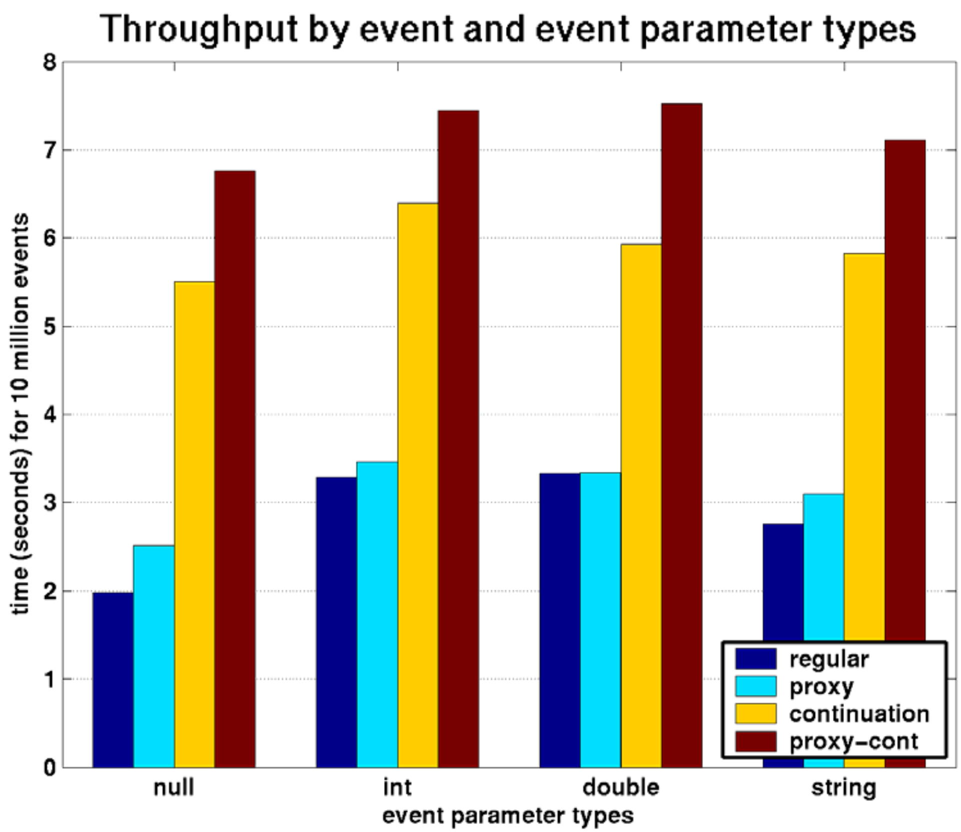
```
METHOD continuable:
  if jist.isRestoreMode:
    jist.popFrame f
    switch f.pc:
      case PC1:
        restoreLocals f.locals
        restoreStack f.stack
        goto PC1
      ...

  instructions

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

  more instructions
```

evaluation: proxy entities and continuations

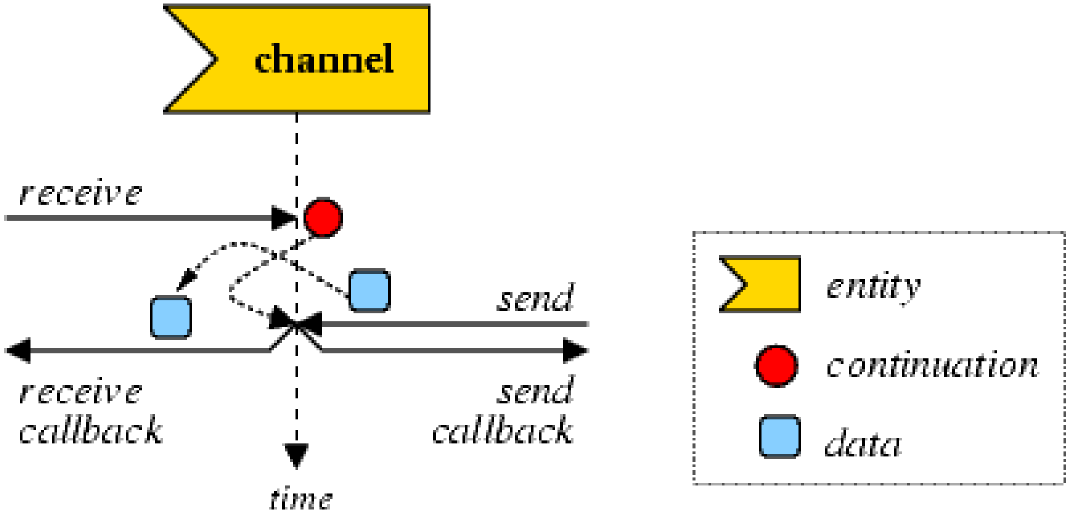


- observations
  - Java primitive type boxing could be faster
  - proxy invocation equivalent to regular invocation
  - continuations within 2-3x regular event invocation
    - overhead proportional to stack height
    - need stack access API

simulation time concurrency

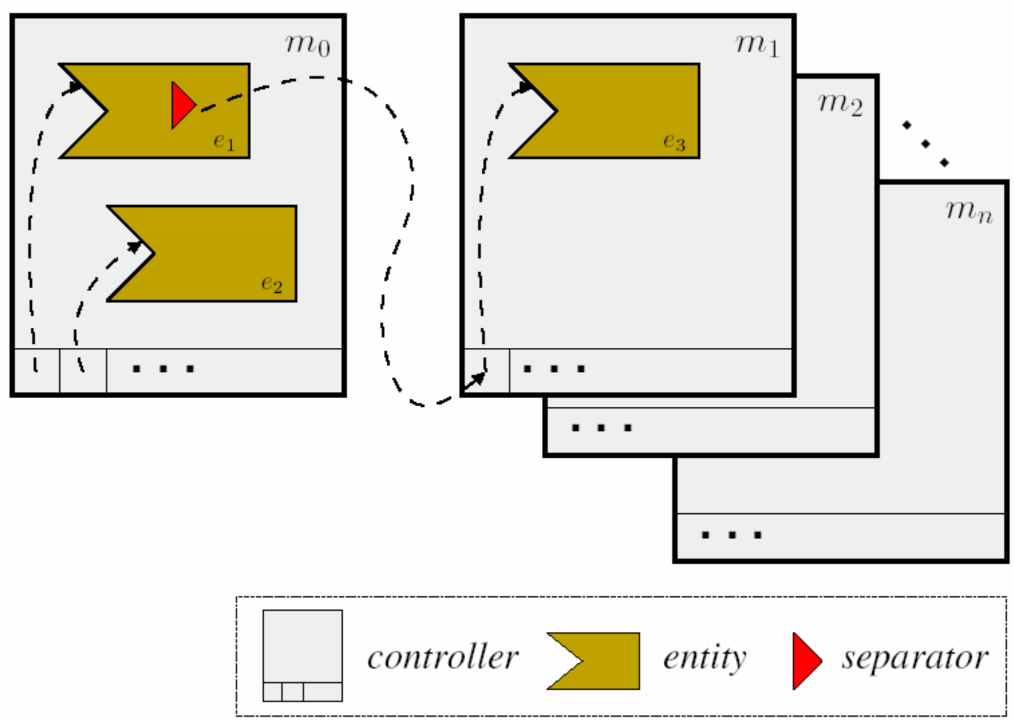
using continuations...

- simulation time threads
  - cooperative and also pre-emptive
- simulation time concurrency primitives
- CSP Channel: `JistAPI.createChannel()`
- locks, semaphores, barriers, monitors, FIFOs, ...



distribution and concurrency (coming soon)

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



benefits of the jist approach

- **more than just performance…**
- **application-oriented benefits**

type safety	source-target statically checked
event types	not required (implicit)
event structures	not required (implicit)
debugging	dispatch location and state available

- **language-oriented benefits**

garbage collection	memory savings, cleaner code
reflection	script-based configuration of simulations
safety	fine granularity of isolation
Java	standard language, compiler, runtime

- **system-oriented benefits**

IPC	no context switch; no serialization
Java kernel	cross-layer optimization
robustness	no memory leaks, no crashes
rewriting	no source-code access required
concurrency	supports parallel and speculative execution
distribution	provides a single system image abstraction

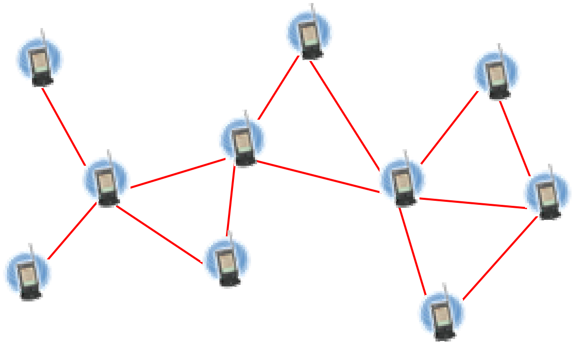
- **hardware-oriented benefits**

cost	COTS hardware, clusters (NOW)
portability	runs on everything

- **simulation research platform**

application: simulating MANETs

- 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



existing alternatives

ns2 **is the *gold standard***
- C++ with Tcl bindings, O(n²)
- used extensively by community
- written for TCP simulation
- modified for ad hoc networks
- processor and memory intensive
- sequential; max. ~500 nodes

PDNS **– parallel distributed ns2**
- event loop uses RTI-KIT
- needs fast inter-connect
- distribute memory, ~1000 nodes

OpNet **– popular commercial option**
- good modeling capabilities
- poor scalability

custom-made **simulators**
- fast, specialized computation
- lack sophisticated execution and also *credibility*

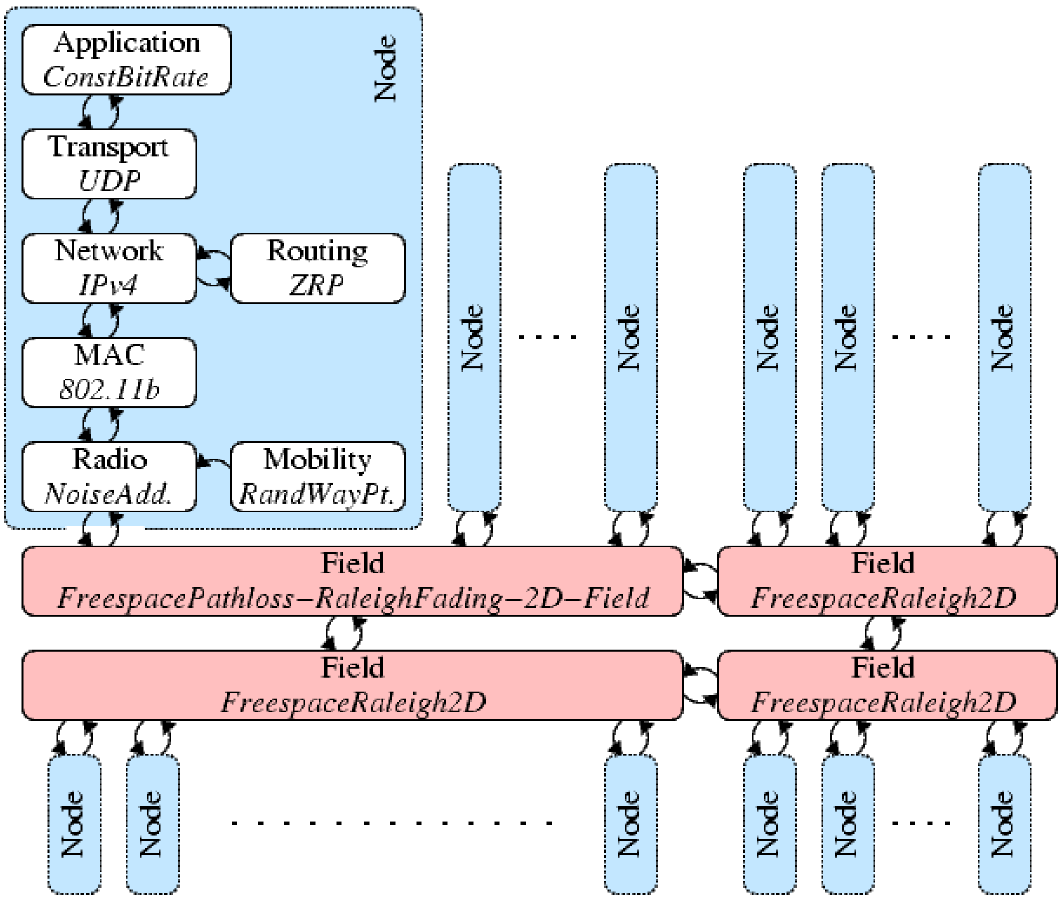
GloMoSim
- implemented in Parsec, a custom C-like language
- entities are memory intensive
- requires "node aggregation," which imposes conservative parallelism, loses Parsec benefits
- shown ~10,000 nodes on NUMA machine (SPARC 1000, est. $300k)

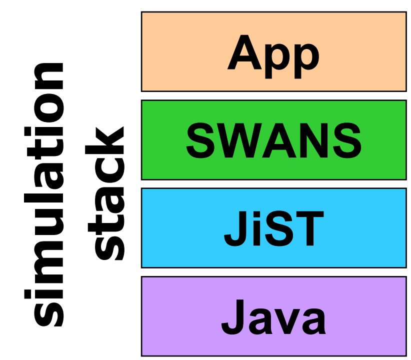
SWAN
- implemented atop the parallel, distributed DaSSF framework
- similar to GloMoSim

Simulation approaches
- languages (e.g. Parsec, Simula)
- libraries (e.g. Yansl, Compose)
- systems (e.g. TWOS, Warped)

SWANS

- **Scalable Wireless Ad hoc Network Simulator**
  - **runs standard Java network applications**
  - **allows vertical *and* horizontal aggregation**





	files	classes	lines
JiST	26	69	9673
SWANS	61	127	14646
Other	20	36	2816
	107	**232**	**27135**

JiST features in SWANS

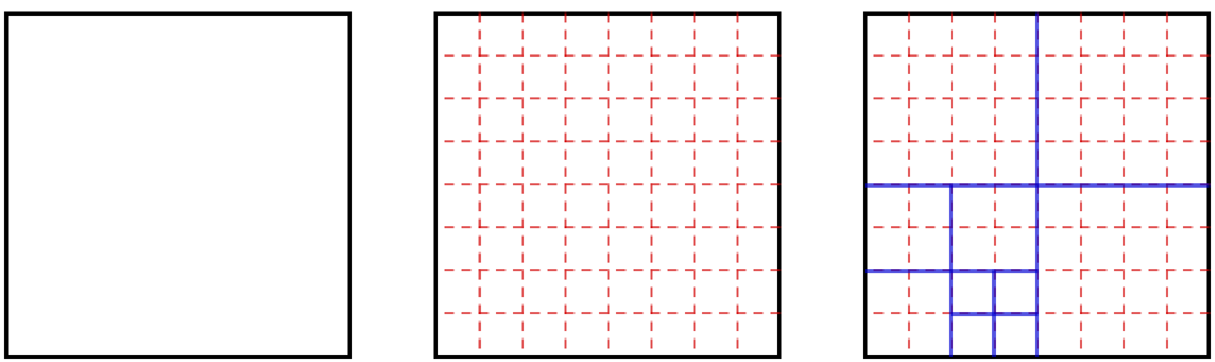
- **SWANS is a JiST application**
  - **entity invocation** — **tracking time**
  - **timeless objects** — **packets**
    saves memory; simplifies memory management
  - **proxy entities** — **network stack**
  - **reflection** — **script-based configuration**
  - **continuations** — **sockets**
    run standard Java network applications over simulated network

field

- signal propagation
  - linear (ns2), grid-based (GloMoSim), hierarchical binning
- fading models
  - none, Raleigh, Rician
- path-loss models
  - free-space, two-ray (i.e. with ground reflection)
- placement models
  - uniformly random
- mobility models
  - static, random-waypoint

hierarchical binning

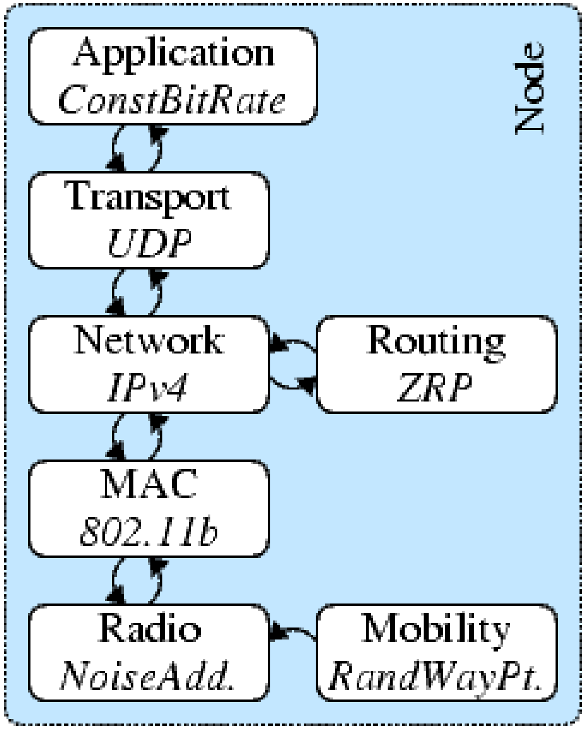
- signal propagation
  - find radios within a given radius
  - critical to performance and scalability
  - optimal algorithm in amortized expectation
    - location update — amortized expected constant time
    - neighborhood search — linear time, O(result set)
  - alternative approaches
    - linear scan — ns2
    - flat binning — GloMoSim, ns2’ (MSWiM '03)
    - function caching — ns2’ (WSC '03)



linear lookup *flat binning* *hierarchical binning*

the network stack

- radio
  - independent (ns2) and additive (GloMoSim) noise models
- link
  - 802.11b, dumb, wired
- routing
  - ZRP (in progress)
  - DSR – Ben Viglietta
  - AODV – Clifton Lin
- transport
  - UDP
  - TCP – Kelwin Tamtoro
- applications
  - any Java networking application!



performance: SWANS

- **simulation configuration**
  - **field** 5x5km^2; free-space path loss; no fading
  - **mobility** random waypoint: v=2-5m, p=10s
  - **radio** additive noise; standard power, gain, etc.
  - **link** 802.11b
  - **network** IPv4
  - **transport** UDP
  - **application** heartbeat neighbor discovery
- **ran on:**
  - PIII 1.1GHz laptop
  - only **384 MB RAM**
  - Sun JDK 1.4.2
- **memory consumption:**
  - **1.2KB per simulated node!**

	nodes		
	1,000	10,000	100,000
ns2	✓	✗	✗
Glomo	✓	✓	✗
SWANS	✓	✓	✓

status

- JiST done
  - functionally complete and performs well
  - user manual available
- SWANS almost finished
  - a bit more development
  - create component library
  - performance and experimental results
- Parallel JiST coming soon
  - develop parallel simulation kernel

summary

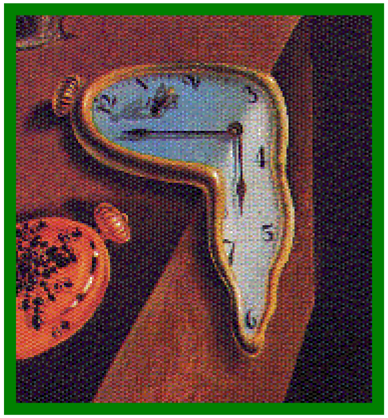
- **JiST – Java in Simulation Time**
  - **convert virtual machine into simulation platform**
  - **efficient both in terms of throughput and memory**
  - **flexible: timeless objects, reflection, debugging, proxy entities, continuations and blocking methods, simulation time concurrency, distribution**
  - **merges systems and languages approaches to simulation**

- **SWANS – Scalable Wireless Ad hoc Network Sim.**
  - **built atop JiST as proof of concept**
  - **component-based framework for wireless simulation**
  - **runs standard Java networking applications**

JiST:
Java in Simulation Time

for

Scalable Simulation of Mobile Ad hoc Networks



THANKS!

jist api

JistAPI.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

```java
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

```java
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();
    }
  }
}
```

introduction to jist

- **JiST – Java in Simulation Time**
  - ***extends*** **object model and execution semantics**
    - **simulations written in plain Java**
    - **compiled classes are modified at load time**
  - **… to run discrete event simulations *efficiently***
    - **reduces serialization and context-switching overhead**
    - **allows parallel and speculative simulation execution**
  - **merges modern language and simulation semantics**
    - **runs Java programs in simulation time**

- **proof of concept**
  - **SWANS – Scalable Wireless Ad hoc Network Simulator**
  - **ideas not specific to Java**

rewriting phase: verification

- verification
  - entity state private and non-static
  - no native, abstract, non-static methods in entities
  - no continuations after entity invocations
    - entity methods should return void
    - exceptions escaping entities cause simulation failure

```
public class MyEntity implements JistAPI.Entity
{
  public void event1(...) {
    ...
  }
}
```

rewriting phase: add entity self reference

- **add entity self reference to entity**
  - **add self reference field**
  - **initialize self reference in constructor**
  - **implement `jist.runtime.Entity` interface**

```
class MyEntity implements JistAPI.Entity {
  private EntityRef _jistField__ref;
  public MyEntity(...)
  {
    super(...);
    this._jistField__ref =
      jist.runtime.Controller.registerEntity(this);
    ...
  }
}
```

*

rewriting phase: intercept state access

- intercept entity state (field) access
  - all entity fields made private
  - get and set accessor methods added for entity fields
  - `get/set-field/static` into method invocations

```
public class MyEntity implements JistAPI.Entity
{
   //public int i;
     private int i;
     public void _jistMethod_Set_i(int i) { this.i = i; }
     public int _jistMethod_Get_i() { return i; }
}
```

rewriting phase: add method stub fields

- **add entity method stub fields to entity**
  - **statically initialized**

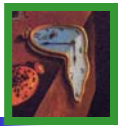
```
class MyEntity implements JistAPI.Entity
{
  public void myEvent(...) { ... }
  public static Method _jistMethodStub_myEvent$signature$;
  static
  {
    jist.runtime.Rewriter.MethodStubInit("MyEntity");
  }
}
```

*

rewriting phase: intercept entity invocations

- **intercept entity invocations**
  - **convert into method call to JiST runtime**
  - **pack arguments into object array (type safety)**
  - **pass correct method stub instance and entity instance**

```
...
//myentity.event1(1, "foo");
jist.runtime.Controller.entityInvocation(
  MyEntity._jistMethodStub_event1$28ILjava$2elang$2eString$3b$29V,
  myentity,
  new Object {
    new Integer(1),
    "foo"
  });
...
```



rewriting phase: modify entity creation

- **modify entity creation**
  - **creates a new entity**
  - **returns entity reference to new entity**

```
...
//MyEntity f = new MyEntity(...);
  EntityRef f = (new MyEntity(...))._jistField__ref
...
```

*

rewriting phase: modify entity references

- modify entity references
  - field entity types
  - method parameter entity types
  - method return entity types

```
public class MyEntity implements JistAPI.Entity
{
  //public SomeEntity entity;
    public EntityRef entity2
  //public void event(SomeEntity e, int i) {
    public void event(EntityRef e, int i) {
      ...
    }
}
```

*

rewriting phase: modify types; translate JistAPI

- **modify typed instructions**
  - type casts
- **translate JiST API calls**
  - `sleep(), getTime(), THIS, ref()`

```
class MySim implements JistAPI.Entity {
 //public void myEvent(MySim sim) {
   public void myEvent(EntityRef sim) {
   //JistAPI.sleep(1);
     JistAPI_Impl.sleep(1);
     if ( JistAPI_Impl.getTime() < 100 )
       //sim.myEvent((MySim)JistAPI.THIS));
         sim.myEvent((EntityRef)JistAPI_Impl.getTHIS());
     System.out.println("myEvent, time="+
     JistAPI_Impl.getTime());
  }
}
```