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. Zygmunt J. Haas

19 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 - \( \sim 500 \) 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

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

Simulation scalability is important
# existing wireless simulators

<table>
<thead>
<tr>
<th>ns2</th>
<th>GloMoSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>written in C++ with Tcl bindings</td>
<td>implemented in Parsec, a custom C-like language</td>
</tr>
<tr>
<td>created for TCP simulation, modified for wireless networks</td>
<td>implements “node aggregation,” to conserve memory</td>
</tr>
<tr>
<td>processor and memory intensive</td>
<td>shown ~10,000 nodes on NUMA machine (SPARC 1000, est. $300k)</td>
</tr>
<tr>
<td>sequential; max. ~500 nodes</td>
<td></td>
</tr>
<tr>
<td>recently “fixed” for ~5000 nodes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OpNet – popular commercial option</th>
<th>custom-made simulators</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast, specialized computation</td>
<td>fast, specialized computation</td>
</tr>
<tr>
<td>poor scalability</td>
<td>lack sophisticated execution and also credibility</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PDNS – parallel distributed ns2</th>
<th>SWAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>event loop uses RTI-KIT</td>
<td>parallelized and distributed using the DaSSF framework</td>
</tr>
<tr>
<td>uses fast inter-connect to distribute memory requirements</td>
<td>similar capabilities to GloMoSim</td>
</tr>
<tr>
<td>shown ~100,000 nodes</td>
<td>shown ~100,000 nodes</td>
</tr>
</tbody>
</table>

**rule of thumb:** extra 10x in scale, using at least 10x hardware and cost
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** - write program in a **standard** language
    - 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

- **transparency**

- **efficiency**

- **application-specific languages**
  - e.g. Apostle, TeD

- **transparency**

- **efficiency**

  - **new language**

virtual machines
the jist approach

- **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

<table>
<thead>
<tr>
<th></th>
<th>kernel</th>
<th>library</th>
<th>language</th>
<th>JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>transparent</td>
<td>++</td>
<td></td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>efficient</td>
<td></td>
<td>+</td>
<td>+</td>
<td>++</td>
</tr>
<tr>
<td>standard</td>
<td>++</td>
<td>++</td>
<td></td>
<td>++</td>
</tr>
</tbody>
</table>
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

```java
// 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

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

- demo!

<table>
<thead>
<tr>
<th>Java</th>
<th>JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack overflow @hello</td>
<td>hello world, time=1</td>
</tr>
<tr>
<td></td>
<td>hello world, time=2</td>
</tr>
<tr>
<td></td>
<td>hello world, time=3</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
</tr>
</tbody>
</table>
SWANS

- **Scalable Wireless Ad hoc Network Simulator**
  - 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 signal propagation
  - component-based simulation architecture written in Java
### SWANS components

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

<table>
<thead>
<tr>
<th></th>
<th>files</th>
<th>classes</th>
<th>lines</th>
<th>semi</th>
</tr>
</thead>
<tbody>
<tr>
<td>SWANS</td>
<td>67</td>
<td>152</td>
<td>17809</td>
<td>5263</td>
</tr>
<tr>
<td>Other</td>
<td>26</td>
<td>53</td>
<td>3994</td>
<td>1602</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>118</td>
<td>300</td>
<td>33068</td>
<td>9757</td>
</tr>
</tbody>
</table>

**Legend**

- **Application**: ConstBitRate
- **Transport**: UDP
- **Network**: IPv4
- **Routing**: ZRP
- **MAC**: 802.11b
- **Mobility**: RandWayPt.
SWANS performance

- 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

Time for 15 minute NDP simulation

Memory for NDP simulation

\[
\begin{array}{|c|c|c|}
\hline
 t=15m & ns2 & GloMoSim \\
\hline
 nodes & time & memory & time & memory & time & memory \\
\hline
 500 & 7136.3 s & 58761 KB & 81.6 s & 5759 KB & 53.5 s & 700 KB \\
5000 & 6191.4 s & 27570 KB & 3249.6 s & 4887 KB & 43.1 s & 1101 KB \\
50000 & & & & & 433.0 s & 5284 KB \\
\hline
\end{array}
\]
SWANS performance

Time for 2 minute NDP simulation

<table>
<thead>
<tr>
<th>network size (nodes)</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1s</td>
</tr>
<tr>
<td>10^2</td>
<td>10s</td>
</tr>
<tr>
<td>10^3</td>
<td>1m</td>
</tr>
<tr>
<td>10^4</td>
<td>1h</td>
</tr>
<tr>
<td>10^5</td>
<td>1h</td>
</tr>
<tr>
<td>10^6</td>
<td>10h</td>
</tr>
</tbody>
</table>

- **SWANS (hier)**
- **SWANS (scan)**

<table>
<thead>
<tr>
<th>$t=2m$</th>
<th>nodes</th>
<th>initial memory</th>
<th>avg. memory</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>10,000</td>
<td>13 MB</td>
<td>45 MB</td>
<td>2 m</td>
</tr>
<tr>
<td></td>
<td>100,000</td>
<td>100 MB</td>
<td>160 MB</td>
<td>25 m</td>
</tr>
<tr>
<td></td>
<td>1 million</td>
<td>1000 MB</td>
<td>1200 MB</td>
<td>5.5 h</td>
</tr>
<tr>
<td></td>
<td>1.0 KB</td>
<td>1.2 KB</td>
<td>20 ms</td>
<td></td>
</tr>
</tbody>
</table>
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
jist micro-benchmark: event throughput

<table>
<thead>
<tr>
<th>5x10^6 events</th>
<th>time (sec) vs. reference vs. JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>reference</td>
<td>0.74</td>
</tr>
<tr>
<td>JiST</td>
<td><strong>0.97</strong></td>
</tr>
<tr>
<td>Parsec</td>
<td>1.91</td>
</tr>
<tr>
<td>ns2-C</td>
<td>3.26</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>9.54</td>
</tr>
<tr>
<td>ns2-Tcl</td>
<td>76.56</td>
</tr>
</tbody>
</table>
jist micro-benchmark: memory overhead

<table>
<thead>
<tr>
<th></th>
<th>per entity</th>
<th>per event</th>
<th>10K nodes sim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JIST</td>
<td>36 B</td>
<td>36 B</td>
<td>21 MB</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>36 B</td>
<td>64 B</td>
<td>35 MB</td>
</tr>
<tr>
<td>ns2 *</td>
<td>544 B</td>
<td>40 B</td>
<td>74 MB</td>
</tr>
<tr>
<td>Parsec</td>
<td>28536 B</td>
<td>64 B</td>
<td>2885 MB</td>
</tr>
</tbody>
</table>
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

- 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
**JiST – Java in Simulation Time**

**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!
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- and language-based approaches to simulator construction
    - efficient, transparent and standard

<table>
<thead>
<tr>
<th></th>
<th>kernel</th>
<th>library</th>
<th>language</th>
<th>JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>transparent</td>
<td>++</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>efficient</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td></td>
</tr>
<tr>
<td>standard</td>
<td>++</td>
<td>++</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **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.