**JiST:**
Java in Simulation Time

for

Scalable Simulation of Mobile Ad hoc Networks

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

MURI Poster Session
26 August 2003

http://www.cs.cornell.edu/barr/repository/jist/
the world today...

- **Transparent** Parallel and **Optimistic** Execution of Discrete Event **Simulations** of MANETs in **Java**

- **discrete event simulations are useful and needed**
- **but, most published ad hoc network simulations**
  - lack network **size** \(\sim 250\) nodes; or
  - compromise **detail** packet level; or
  - curtail **duration** few minutes; or
  - are of sparse **density** tens of nodes/ \(km^2\); or
  - etc...

- i.e. **limited simulation scalability**
the world today... in perspective

- A university **campus**
  - Cornell students ~ 30,000
  - Wireless devices per student average ~ 1
  - Main campus < 4 km²

- The United States **military**
  - Troops deployed in Iraq 100-150,000 (in clusters)
  - Wireless devices per soldier ???
  - Territory 400,000km²

- And, predictions of
  - smaller devices, better radios and chips
  - smart dust, wearable/disposable/ubiquitous computing

Simulation **scalability** is important.
introduction to jist

- J
c
ava-based simulation framework
  - J
iST extends object model and execution semantics
  - ... to run discrete event simulations
  - transparently
    - simulations written in plain Java
    - compiled classes are modified at load time
  - and 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
system architecture

2. Run simulation within JiST (within Java). Simulation classes are dynamically rewritten to introduce simulation time semantics.
3. Rewritten program interacts with simulation kernel.
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**
  - allows vertical and horizontal aggregation

### Table

<table>
<thead>
<tr>
<th></th>
<th>JiST</th>
<th>SWANS</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>files</td>
<td>26</td>
<td>52</td>
<td>16</td>
</tr>
<tr>
<td>classes</td>
<td>65</td>
<td>115</td>
<td>26</td>
</tr>
<tr>
<td>lines</td>
<td>9278</td>
<td>12871</td>
<td>2042</td>
</tr>
</tbody>
</table>

- larger than JiST code-base
- simpler than GloMoSim and ns2 implementations
- developed in <3 months
**performance: event throughput**

![Graph showing event throughput performance comparison between different tools.

### Ratio Table

<table>
<thead>
<tr>
<th># events</th>
<th>JiST</th>
<th>GloMoSim</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>0.044s</td>
<td>0.435s</td>
<td>10%</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.262s</td>
<td>2.938s</td>
<td>9%</td>
</tr>
<tr>
<td>$10^7$</td>
<td>2.301s</td>
<td>28.04s</td>
<td>8%</td>
</tr>
<tr>
<td>$10^8$</td>
<td>22.48s</td>
<td>278.4s</td>
<td>8%</td>
</tr>
</tbody>
</table>

Serial throughput increase of 12x

<table>
<thead>
<tr>
<th>$5 \times 10^6$ events</th>
<th>time (sec)</th>
<th>vs. baseline</th>
<th>vs. JiST</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>1.640</td>
<td>1.0x</td>
<td>0.8x</td>
</tr>
<tr>
<td>JiST</td>
<td>1.957</td>
<td>1.2x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Parsec</td>
<td>3.705</td>
<td>2.3x</td>
<td>1.9x</td>
</tr>
<tr>
<td>ns2-C</td>
<td>5.151</td>
<td>3.1x</td>
<td>2.6x</td>
</tr>
<tr>
<td>GloMoSim</td>
<td>23.720</td>
<td>14.5x</td>
<td>12.1x</td>
</tr>
<tr>
<td>ns2-Tcl</td>
<td>160.514</td>
<td>97.9x</td>
<td>82.0x</td>
</tr>
</tbody>
</table>
performance: memory overhead

<table>
<thead>
<tr>
<th>memory</th>
<th>entity</th>
<th>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>36 B*</td>
<td>72 MB*</td>
</tr>
<tr>
<td>Parsec</td>
<td>28536 B</td>
<td>64 B</td>
<td>2885 MB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory</th>
<th>Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>JiST</td>
<td>36 bytes &gt; 10^6 entities</td>
</tr>
<tr>
<td>Parsec</td>
<td>28536 bytes ~ 10^4 entities</td>
</tr>
</tbody>
</table>

JiST scales to more entities per process
performance: SWANS

- simulation configuration
  - field: 5x5km²; 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!

<table>
<thead>
<tr>
<th>nodes</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns2</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Glomo</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>SWANS</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
and lots more!

- **timeless objects**: pass-by-reference to avoid copy
- **proxy entities**: interface-based entity creation
- **continuations**: call and callback, blocking methods
- **concurrency**: channel, threads, monitors, locks...
- **distribution**: separators track entities across machines
- **scripting**: embed engines for Java, Python, Tcl, etc...
benefits of the jist approach

• more than just scalability.
• 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  pure Java; “runs everywhere”