JiST:
Java in Simulation Time

Transparent Parallel and Optimistic Execution of Discrete Event Simulations (PDES) of Mobile Ad hoc Networks (MANETs)

Rimon Barr
barr@cs.cornell.edu

Wireless Network Lab
Cornell University
3 March 2003

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

• discrete event simulations are useful
  • physics, chemistry, genomics, proteomics
  • geology, meteorology, astronomy
  • processors, networks

• basic structure
  • simulation time
  • state (partitioned)
  • events, event queue

• ad hoc network simulations
  • lack scalability, or
  • compromise detail
simulating ad hoc wireless 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
existing MANET simulators

- ns2
- PDNS
- OPNet
- GlomoSim
- custom-made
design space

- interpreted execution
  vs. compiled

- general purpose language
  vs. domain specific

- shared nothing architecture
  vs. shared everything
design space

• interpreted execution  
  vs. compiled

pros:
  • runtime inspection, interrupts, debugging
  • simulation composition and configuration
  • reflection facilitates debugging

cons:
  • rely on JIT for performance
design space

• general purpose language
  vs. domain specific

pros:
• credibility, code re-use
• general compiler advances
• familiarity, other software engineering arguments

cons:
• specific compiler optimizations
design space

- shared nothing architecture vs. shared everything

pros:
  - inexpensive (COTS)
  - portable, simple

cons:
  - synchronization
  - serialization
existing MANET simulators

• **ns2** is the gold standard
  • Tcl-based, with C++ bindings
  • used extensively within research community
  • initially developed for detailed TCP simulations
  • Monarch – modified to support ad hoc networks
  • processor and memory intensive, sequential
  • $O(n^3)$
  • max. ~ 250 nodes

• **PDNS** – parallel distributed ns2
  • perform event loop over Georgia Tech. RTI-KIT
  • requires fast inter-connect
  • helps with memory limits

• **OPNet**
existing MANET simulators

- **Glomosim**
  - written in Parsec, a custom C-like language
  - entities map to processes
  - messages map to IPCs
  - “node aggregation”
    - imposes conservative parallelism
  - max. ~10,000 nodes
    - but on NUMA: Sun SPARCserver 1000, est. $300,000

- **custom-made simulators**
  - fast, specialized computation
  - lack sophisticated execution, parallelism
  - credibility
java in simulation time

• Java-based simulation framework

• runs discrete event simulations
  • efficiently
    • in parallel
    • optimistically
  • transparently
    • simulations written in plain Java
    • compiled class files are modified at load time

• proof of concept
  • SWANS – Scalable Wireless Ad hoc Network Simulation
  • ideas not specific to Java
simulation time

• program time
  • progress of program independent of time

• real time
  • progress of program dependent on time

• simulation time
  • progress of time dependent on program
  • simulation event loop embedded in virtual machine
  • simulation time is advanced by the program
an example

- the “hello world” of discrete event simulations

```java
class MySim implements JistAPI.Entity
{
    private int data = 0;
    public void myEvent()
    {
        JistAPI.sleep(1);
        myEvent();
        System.out.println("myEvent, sim-time=" + JistAPI.getTime() + " data=" + (data++));
    }
}
```

- result: one event per time step
**jist api**

- JistAPI class provides application `calls’ to the JiST runtime
- permits standard Java compilation

```java
interface Entity {
    - tag object as entity
    long getTime() - return simulation time
    void sleep(long t) - advance simulation time
    EntityRef THIS - reference to this entity
    EntityRef ref(Entity e) - reference to entity
}
```

- runs on standard Java VM
entities and separators

- program state contained in objects
- objects partitioned into entities
- entities must not share state
  - live objects are referenced by exactly one entity
- therefore, each entity has its own entity time

- think, component interface...
entities and separators

- separators (or entity references)
  - separate application state, application time
  - provide location independence

- method calls to entities
  - non-blocking
  - invoked at caller entity time
  - no continuation
    - neither return, nor exception
rewriting simulations

- **dynamic class loader**
  - no source code access
  - uses Apache Byte Code Engineering Library (BCEL)
  - ignores non-application packages

- **steps**
  - verification
  - add entity self reference
  - intercept entity state (field) access
  - add method stub fields
  - intercept entity invocations
  - modify entity creation
  - modify entity references
  - modify typed instructions
  - translate JiST API calls
rewriting simulations

- 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

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

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

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

- 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

```java
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 simulations

- add entity method stub fields to entity
  - statically initialized

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

  public static Method _jistMethodStub_myEvent$signature$;

  static {
    jist.runtime.Rewriter.MethodStubInit("MyEntity");
  }
}
```
rewriting simulations

- 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$L2ILjava$lang$3b$29V,
    myentity,
    new Object {
        new Integer(1),
        "foo"
    }
);
...

rewriting simulations

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

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

...
rewriting simulations

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

```java
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 simulations

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

```java
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();
    }
}
```
benefits of rewriting approach

- standard Java compilation
- standard Java Virtual Machine
- no source code access required
- retains type-safety properties
- rewriting occurs once, at load time
- partitioning of application state into entities
- location independence of entities
- event loop embedded into virtual machine
- transparent parallel and optimistic execution
- dynamic and interpreted environment
  - yet, efficient...
benefits of rewriting approach

- micro-benchmark of event throughput
  - includes garbage collection
  - includes event scheduling
  - includes message passing
  - ... still preliminary!

<table>
<thead>
<tr>
<th># events</th>
<th>JiST</th>
<th>GlomoSim</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^5$</td>
<td>0.22s</td>
<td>0.48s</td>
<td>45%</td>
</tr>
<tr>
<td>$10^6$</td>
<td>1.44s</td>
<td>3.18s</td>
<td>45%</td>
</tr>
<tr>
<td>$10^7$</td>
<td>13.55s</td>
<td>30.46s</td>
<td>44%</td>
</tr>
<tr>
<td>$10^8$</td>
<td>130.6s</td>
<td>292.5s</td>
<td>45%</td>
</tr>
</tbody>
</table>

serial throughput increase of 2.2x
serial simulation time

- program state in objects
- objects partitioned among entities
- separators of state, time
- serial execution

- inject orthogonal code
  - inspection, node mobility, debugging
current status

- rewriter working
- simulation infrastructure working
  - serial execution
- building swans
  - designing physical layer
- parallelization
- optimistic execution
  - check-pointing and undo
  - bounding, load balancing
coming soon: a real application

• **SWANS**
  • Scalable Wireless Ad hoc Network Simulator

• **implement**
  • physical: propagation, reception
  • link: 802.11b
  • routing: DSR or ZRP
  • application: CBR
  • mobility model
coming soon: parallel simulation time

- lock-step in simulation time
  - concurrent events
  - conservative
- separators
  - location-independence
  - entity tracking

- balance load
- minimize network overhead
coming soon: optimistic simulation time

- checkpoint entities
- cascade undo
  - cancellation
  - propagation
- rollback interface
  - automatic rollback method generation
- balance forward progress of time
JIST: Java In Simulation Time

Transparent Parallel and Optimistic Execution of PDES of MANETs

Thank you.

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