JiST– Java in Simulation Time
User Guide and Tutorial

Rimon Barr
barr@cs.cornell.edu

Introduction

Discrete-event simulators are important scientific tools, and the focus of a vast body of computer science research that is directed at their efficient design and execution. The JiST system, which stands for **J**ava **i**n **S**imulation **T**ime, follows a long line of simulation frameworks, languages and systems. JiST is a new Java-based discrete-event simulation engine, with a number of novel and unique design features. The purpose of this document is to expose those features with examples, to describe the overall functioning of the system, and to leave the reader with an understanding of how to use JiST to construct efficient, robust and scalable simulations.

Architecture

The JiST system architecture, depicted in Figure 1, consists of four distinct components: a compiler, a bytecode rewriter, a simulation kernel and a virtual machine. One writes JiST simulation programs in plain, unmodified Java, and compiles them to bytecode using a regular Java language compiler. These compiled classes are then modified, via a bytecode-level rewriter, to run over a simulation kernel and to support the *simulation time* semantics described shortly. The simulation program, the rewriter and the JiST kernel are all written in pure Java. Thus, this entire process occurs within a standard, unmodified Java virtual machine (JVM).

The benefits of this approach are numerous. Embedding the simulation semantics within the Java language allows us to reuse a large body of work, including the Java language itself, its standard libraries and existing compilers. JiST benefits from the automatic garbage collection, type-safety, reflection and many other properties of the Java language. This approach also lowers the learning curve for users and facilitates the reuse of code for building simulations. The use of a standard virtual machine provides an efficient, highly-optimized and portable execution platform, and allows for important cross-layer optimization between the simulation kernel and running simulation. Furthermore, since the kernel and simulation are both running within the same process space we reduce serialization and context switching overheads. In summary, a key benefit of the JiST approach is that it allows for the efficient execution of simulation programs, within the context of a modern, popular language. JiST combines simulation semantics, found in custom simulation languages and simulation libraries, with modern language capabilities. This design results in a system that is convenient to use, robust and efficient.

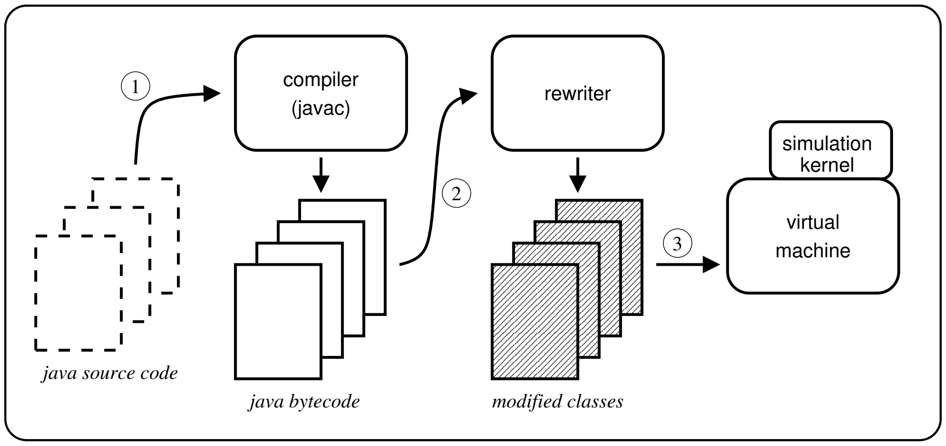


Figure 1: The JiST system architecture – simulations are compiled (1), then dynamically instrumented by the rewriter (2) and finally executed (3). The compiler and virtual machine are standard Java language components. Simulation time semantics are introduced by the rewriting classloader and supported at runtime by the Java-based simulation kernel.

Hello world

We begin with a customary example of a JiST program: the 'Hello World!' of simulations, listed in Figure 2. The first thing to note is that this is a valid Java program. You can compile it with a Java compiler: `javac hello.java`; and run it as a regular Java program: `java hello`. Scanning the source listing, you should also notice the repeated use of the `JistAPI`, on lines 3, 14 and 17. This `JistAPI` class represents the application interface exposed by the simulation kernel. Of course, if we run our `hello` program without the JiST runtime, simply under a regular Java runtime, there will be no active simulation kernel. In this case, the entire `JistAPI` acts as a dummy class that merely facilitates type-safe execution: the `Entity` interface is empty, the `sleep` call does nothing and the `getTime` function returns zero. In fact, the correct way to think about the `JistAPI` is that it marks the program source in a manner that both respects Java syntax and is preserved through the compilation process. It allows type-safe compilation of JiST simulation programs using a conventional Java compiler.

However, when we run our `hello` simulation under JiST: `java jist.runtime.Main hello`, we *will* have a simulation kernel loaded and running. Among other things, this kernel installs a class loader into the JVM, which dynamically rewrites our `hello` bytecode as it is loaded. The various `JistAPI` markings within the compiled simulation program serve to direct the code transformations that introduce the simulation time semantics into the bytecode. The entire JiST functionality is exposed to the simulation developer in this manner, via `jist.runtime.JistAPI`.

Now, returning to our `hello` example, you will notice that the call to `myEvent` on line 15 is recursive. Running the program on its own, without the JiST runtime, will produce a stack overflow, as expected, on exactly that line.

hello.java

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

Figure 2: The simplest of simulations consists of a single entity that emits a message at each time step.

However, executing the same application under JiST produces the output:

```
> simulation start
> hello world, t=1
> hello world, t=2
> etc.
```

In essence, our method call has been transformed into a simulation event on the `hello` *entity*. It is scheduled and invoked by the simulation kernel in *simulation time*. Note, also, that `hello` is an entity, not a regular object, since it implements the `JistAPI.Entity` interface. You may also observe, from the output, that the `sleep` serves to advance the simulation time of the system. And, clearly, there is no stack overflow. Thus, although we reuse the Java language, as well as its compiler and virtual machine, we extend the object model of the language and execution semantics of the virtual machine, so that simulation programs will run in simulation time.

Simulation time

The standard Java virtual machine is a stack-based Java byte-code execution engine with well-defined execution semantics [6]. In this standard execution model, which we refer to as *actual time* execution, the passing of time is not dependent on the progress of the application. In other words, the system clock advances regardless of how many byte-code instructions are processed. Also, the program can advance at a variable rate, since it depends not only on processor speed, but also on other unpredictable things, such as interrupts and application inputs. Moreover, the JVM certainly does not make strong guarantees regarding timely program progress: it may decide, for example, to perform garbage collection at any point.

To solve such problems, much research has been devoted to executing applications in *real time* or with more predictable performance models, wherein the runtime can guarantee that instructions or sets of instructions will meet given deadlines. Thus, the rate of application progress is made dependent on the passing of time.

In JiST, the opposite is true: that is, the progress of time depends on the progress of the application. The application clock, which represents simulation time, does not advance to the next discrete time point until all processing for the current, discrete simulation time has been completed.

In *simulation time* execution, individual primitive application byte-code instructions are processed sequentially, following the standard Java control flow semantics, but the application time remains unchanged. Application code can advance time only via the `sleep(`n`)` system call. In essence, every instruction takes zero time to process except `sleep`, which advances the simulation clock forward by exactly n simulated time quanta. The JiST runtime processes an application in its simulation-temporal order until all the events are exhausted or until a pre-determined ending time is reached, whichever comes first. We summarize these different execution models in Table 1.

actual time	-	program progress and time are independent
real time	-	program progress depends on time
simulation time	-	time depends on program progress

Table 1: Relationship between progress progress and time under different execution models

Object model and execution semantics

JiST simulation programs are written in plain Java, an object-oriented language. As in any object-oriented language the entire simulation program comprises numerous classes that collectively implement the logic of the simulation model. During its execution, the state of the program is contained entirely within individual objects. These objects communicate by passing messages, which are represented as object method invocations in the language.

In order to facilitate the design of simulations, JiST extends this traditional programming model with the notion of simulation *entities*. Programmatically, entities are defined as instances of classes that implement the `JistAPI.Entity` interface. Although entities are regular objects to the JVM, they serve to logically encapsulate application objects, as shown in Figure 3, and demarcate independent simulation components. Thus, every object should be logically contained within an entity. Conversely, the state of an entity is the combined state of all the objects reachable from it.

To enforce this strict partitioning of a simulation into entities, and prevent object communication across entity boundaries, each (mutable) object in the system must belong to a single entity and must be entirely encapsulated within it. In Java, this merely means that all references to an object must originate either directly or indirectly from a single entity. This condition suffices to ensure simulation partitioning, since Java is a safe language.

The JiST kernel manages a simulation at the granularity of its entities. Instructions and method invocations *within* an entity follow the regular Java control flow and semantics, entirely opaque to the JiST infrastructure. The vast majority of this code is involved with encoding the logic of the simulation model and is entirely unconnected to the notion of simulation time. All the standard Java class libraries are available and behave as expected. In addition, the simulation developer has access to a few basic JiST primitives, including functions such as `getTime` and `sleep`, which allow for interactions with the simulation kernel.

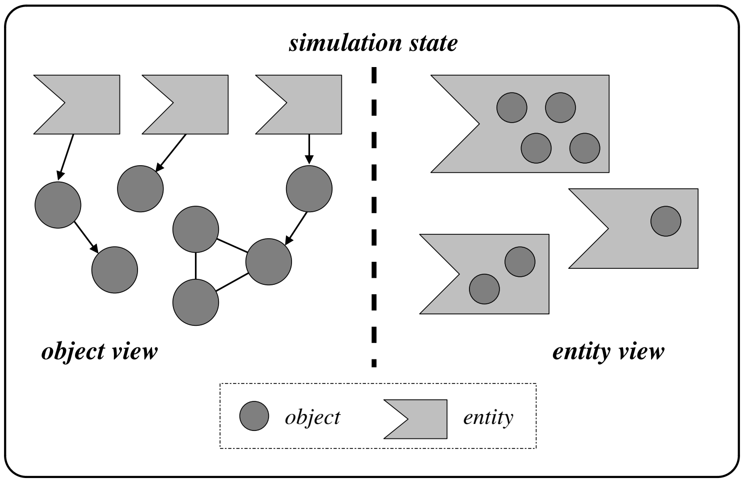


Figure 3: Simulation programs are partitioned into entities along object boundaries. Thus, entities do not share any application state and can independently progress through simulation time between interactions.

In contrast, invocations *across* entities are executed in simulation time. This means that the invocation is performed on the callee (or target) entity when its state is at the same simulation time as the calling (or source) entity. In this manner, simulation events are intuitively represented as method invocations across entities. This is a convenient abstraction in that it eliminates the need for an explicit simulation event queue. The execution semantics are that method invocations on entities are non-blocking. They are merely queued at their point of invocation, not invoked. It is the JiST kernel that actually runs the event loop, which processes the simulation events, invoking the appropriate method for each event dequeued in its simulation time order, and executing the event to completion without continuation. In other words, cross-entity method invocations act as synchronization points in simulation time. Or, from a language-oriented perspective, an entity method is like a coroutine, albeit scheduled in simulation time.

A complete separation of entities, as discussed above, is not possible without an additional consideration. That is, in order to invoke a method on another entity – to send it an event – the caller entity *must* hold some kind of reference to the target entity, as depicted in Figure 4. We, therefore, distinguish between object references and entity references, and expand on the constraints stated above. All references to a given (mutable) object must originate from within the same entity. However, references to entities are free to originate from any entity. The rationale is that object references imply inclusion within the state of an entity, whereas entity references represent channels along which simulation events are transmitted. As a direct consequence, entities do not nest, just as regular Java objects do not.

We reintroduce the separation of entities at runtime, by transparently replacing all entity references within the simulation bytecode with special objects, called *separators*, as shown on the right of Figure 4. The separator object identifies a particular entity, but without referencing it directly. Rather, separators store a unique entity identifier that is generated by the kernel for each entity during its initialization. Separators can be held in local variables, stored in fields or objects or passed as method parameters, just like the regular object references that they replace. Since the replacement occurs across the entire simulation bytecode, it remains type-safe.

Due to this imposed separation, we guarantee that interactions among entities can *only* occur via the JiST kernel. Furthermore, since entities do not share any application state, each entity may actually progress through simulation

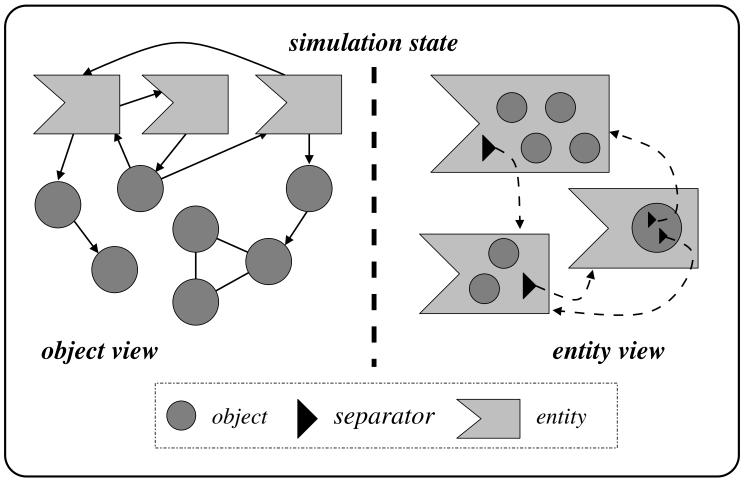


Figure 4: At runtime, entity references are transparently replaced with separators. This both preserves the separation of entity state, and also serves as a convenient point to insert additional functionality. For example, separators can provide the abstraction of a single system by tracking the location of remote entities and relaying events to them.

time independently between interactions. The separators, in effect, represent an application state-time boundary around each entity, similar to a TimeWarp [5] process but at finer granularity. Thus, by tracking the simulation time of each individual entity, these separators allow for concurrent execution. By adding the ability to checkpoint entities, via Java serialization, for example, the system can support speculative execution as well. Finally, separators also provide a convenient point for the distribution of entities across multiple machines. In a distributed simulation, the separators function also as remote stubs, and transparently maintain a convenient abstraction of a single system image. Separators transparently store and track the location of entities as they migrate among machines in response to fluctuating processor, memory and network loads.

The role of the simulation developer, then, is to codify the simulation model in regular Java, and to partition the state of the simulation not only into objects, but also into a set of independent entities along reasonable application boundaries. The JiST infrastructure will transparently execute the program efficiently, while retaining the simulation time semantics.

Simulation time invocation

To illustrate the extended JiST object model and simulation time invocation, we return to our `hello` program and describe its execution step-by-step. Omitting unnecessary complexity in the interests of clarity, below are the basic operations that occur when one executes: `java jist.runtime.Main hello`.

- The JVM is loaded and initialized.
- The JiST simulation kernel is loaded into the JVM and initialized. Among other activities, a dynamic class loader is installed, so that JiST verification and rewriting of application occurs on demand.

- The kernel creates an anonymous, bootstrap simulation entity. The kernel then creates and enqueues a bootstrap simulation event at time $t = 0$, which will invoke the simulation program's `main` method with any command-line arguments, when the simulation event loop begins.
- The kernel enters the simulation event loop and our `hello` simulation begins.
- Process the bootstrap event at time $t = 0$:
  - Line 7: Output `''simulation start''`.
  - Line 8: Create and initialize a new instance of `hello`. Note that `hello` is an *entity*, not a regular object, because it implements the `JistAPI.Entity` interface on line 3. Thus, at runtime, local variable `h` will actually contain a separator object.
  - Line 9: Perform a simulation time invocation of `myEvent`. In other words, *schedule* an event at the current local simulation time $t = 0$ to invoke `myEvent` on the entity referenced by `h`.
  - End of event.
- Process the `myEvent` event at time $t = 0$.
  - Line 14: Advance the local simulation time by one time quantum to $t = 1$.
  - Line 15: Perform a simulation time invocation of `myEvent`. That is, *schedule* an event at current local simulation time $t = 1$ to invoke `myEvent` on the current entity.
  - Line 16: Emit the `''hello world''` message, with the current local simulation time.
  - End of event.
- Process the next scheduled `myEvent` event at time $t = 1, 2, 3, \cdots$, as above.

There are a number of additional observations to make regarding the execution of our primitive simulation program. Note, for example, that there is no mention of concurrency. In fact, there may be multiple threads that share the simulation event queue and execute the events in parallel. The JiST kernel, however, does guarantee that events are atomic. Thus, events on the same entity are executed to completion without interruption. Entity invocations do not interrupt event processing; the invocation is merely enqueued. Also, other events may not interrupt processing, even during a `sleep` call. The `sleep` merely advances the *local* simulation clock. For instance, take two events t_1 and t_2, at times $t = 1$ and $t = 2$, respectively. Assume that the first event, during the course of its execution, calls `sleep` for 3 time quanta. Nevertheless, t_2 will be invoked only after t_1 processing is complete. In effect, the `sleep` call is used for scheduling outgoing invocations with delay, but not for delaying internal operations. Thus, all operations on the state of an entity occur atomically, at the simulation time of the current event.

There are also some intricacies with respect to the extended object model. For instance, only invocations on *public* methods of entities are rewritten into event dispatches. While there are technical motivations for this decision that stem from the Java bytecode semantics, it also matches conceptually with the general Java design. An entity represents an event-based interface around some internal state. This external interface, as with regular Java objects, is considered to be the set of public methods. Non-public methods are considered to be regular object methods, not entity methods, and have the regular Java invocation semantics. Furthermore, since entity method invocations are event-based, they are performed without continuation. Consequently, there can be no continuation defined. In other words, public entity methods must return `void` and can not statically declare any throwable exceptions. If, at runtime, an exception nevertheless occurs and escapes outside the boundary of an entity to the kernel event loop, then the system will immediately terminate the faulty simulation. For a similar reason, entities may not expose public fields. Specifically, a remote field access is equivalent to an accessor method, with a non-`void` return type. Likewise, static fields represent state shared across entities, and are disallowed. Finally, entity constructors are treated specially. They instantiate the desired entity, but return a separator object referencing the newly created entity, in place of the object itself. In fact, it is not the constructor that is actually modified, rather the constructor calling-point right after a corresponding `new` instruction.

Finally, while the semantics of simulation time invocation are sufficient to use the system, it is important to understand the underlying implementation of simulation time invocation, at least at a high level. The simulation time invocations are introduced by the rewriter, which inspects all method invocation instructions to find invocations on targets that are entities. At this program location, the stack will contain the target reference (in fact, the separator object), and any arguments. The rewriter inserts bytecode instructions to pack the arguments into an object array, and to rearrange the stack to hold three items: the target, a method reflection object, and the invocation argument array. The invocation instruction is then converted into a call to the simulation kernel, which creates and schedules a simulation event with this information. For performance, method reflection objects are pre-generated statically, object arrays are pooled to avoid object instantiation as are the event objects, the expensive Java reflection access checks are performed once and then safely disabled, and synchronization locks are escalated, wherever possible. Unfortunately, the Java type system and reflection interface forces us to wrap primitive types in their object counterparts to be placed in the argument array, which incurs a small, but notable overhead. It would be nice to have the JVM extended to support the efficient "boxing" of primitive types, as in the CLR. Simulation developers may wish to note this fact and pass along objects instead of primitives, but only when this can be achieved without an additional object instantiation.

Timeless objects

Having exhausted our `hello` simulation and discussed the central notion of simulation time invocation, we now introduce more advanced JiST features and add complexities to the basic model. Our first extension is the notion of *timeless* objects, introduced purely for reasons of performance. A timeless object is defined as one that will not change over time. Knowing that a value is temporally stable, allows the system to safely pass it across entities by reference, rather than by copy.

The system may be able to infer that an object is open-world immutable, through static analysis, and automatically add the timeless labels. In some cases, the analysis can be overly conservative due to restrictions of the Java type system. The programmer can then explicitly define an object to be timeless, by tagging the object with the empty `JistAPI.Timeless` interface. This tag implies that the object will not be modified in the simulation's future, even though it technically could be.

The timeless tag may also be useful for sharing state among entities in an effort to reduce simulation memory consumption, as depicted in Figure 5. This facility should be exercised with care. Since entities may progress at different rates through simulation time, any shared state *must* actually be timeless. Objects with temporally unstable values shared across entities will create temporal inconsistencies in the state of the simulation. Conversely, the simulation developer should be aware that non-timeless objects are passed across entities by copy. Thus, it is not

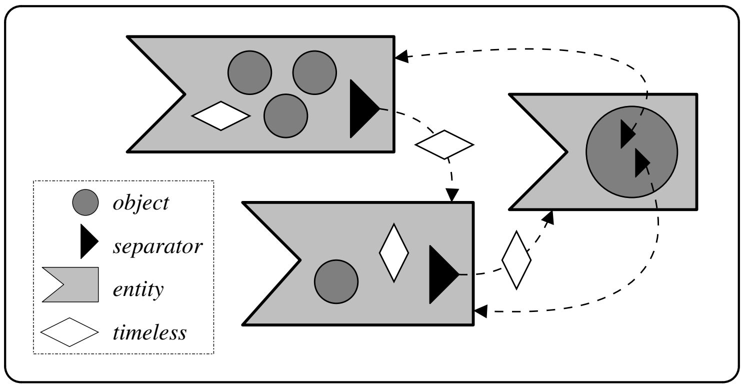


Figure 5: Objects passed across entities should be timeless – in other words, should hold temporally stable values – to prevent temporal inconsistencies in the simulation state. Sharing timeless objects among entities is an effective way to conserve simulation memory.

possible to transmit information back to the caller using an object parameter in an event. Two separate events should be used instead.

Proxy entities

Entities encapsulate state and present an event-oriented interface encoded as methods. However, methods also represent regular Java invocations on objects. The developer of an entity class must, therefore, remember that public methods are invoked in simulation time while non-public methods are regular invocations, that only non-public and non-static fields are allowed, etc. This can be confusing, and it imposes restrictions on the developer and can lead to awkward coding practices within entity classes. In other words, there exists a clash of functionality between entity and object invocation semantics at the syntactic level.

To address this issue, we introduce *proxy* entities. As their name implies, proxy entities exist only to relay incoming events onto a target object. As with any entity method invocation, there are still two phases: event dispatch and event delivery. And, even though the internal mechanisms used for both event dispatch and delivery of proxied invocations may be totally different from regular entity invocations, there are no syntactic differences visible to the developer. Furthermore, the performance is equivalent.

Proxies are created via the `proxy` API call, which accepts two parameters. The first parameter is the proxy target, and can be one of three types: an entity, a regular object or a *proxiable* object. The proxying of each of the three possible target types is illustrated in Figure 6, and described below. The second parameter to the system call is the proxy interface, indicating the methods that are to be exposed and relayed to the target. Clearly, the proxy target must implement the proxy interface, and this is also verified by the system.

- **regular object** proxy: Regular objects do not contain the machinery to receive events from the system. They need to be contained within entities. Therefore, we proxy an object by creating a new entity, which implements the given interface and relays methods to the given target object. In other words, we wrap the object, and all objects reachable from it, with a new entity.

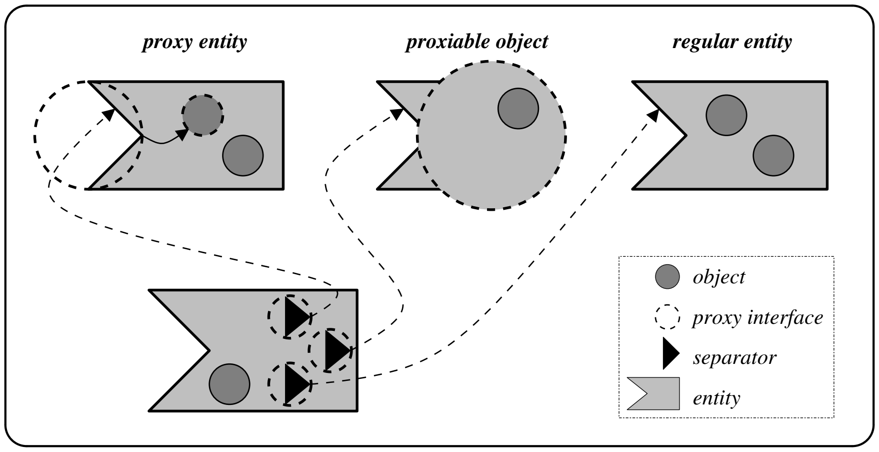


Figure 6: Proxy entities use an interface-based approach to identify entity methods, thus simplifying the development of entities. The proxy system call behavior corresponds to the type of object passed to it.

Note that this approach inserts the overhead of an additional method invocation into the event delivery process. Note also that if one invokes the *proxy* call twice on the same object, the result is two new entities that share that object in their state, as well as all those reachable from it. We can, however, eliminate this indirection through the use of *proxiable* objects.

- **proxiable object** proxy: A proxiable object is any object that implements the `JistAPI.Proxiable` interface. This interface, like the `JistAPI.Entity` interface, is empty and serves only as a marker to the rewriter. The effect of this tag is to introduce the necessary machinery required to receive events from the kernel. In this manner, the extra method call overhead of an intermediate relaying entity is eliminated, so this approach is always preferred over proxying regular objects, when source code is available. It merely requires the addition of a proxiable tag to the target.
- **entity** proxy: Finally, the last kind of object that we can proxy is an existing entity. An entity already includes all the machinery required to receive and process events. Thus, the event delivery path is unchanged. However, the `proxy` call does serve an important function on the dispatch side. The result of the `proxy` call, in this and in all the cases above as well, is a new separator object that implements and relays *only* the methods of specified interface. Thus, the system call serves to restrict the events allowed from a particular source. One should consider a proxy separator to be a *capability* (in the systems sense), since it is unforgeable: the separator can not be cast to other interfaces at runtime.

Proxy entities simplify development. They allow an object the flexibility of combining both event-driven and regular method invocation, without loss of performance. They are interface-based, so they do not interfere with the object hierarchy. And, they allow for a capability-like isolation of functionality, which is useful in larger simulation programs. Finally, although the proxy separators are completely different in implementation, they behave in the same way as regular separators, both in terms of execution semantics and performance.

The code listing in Figure 7 shows how to use proxy entities with a basic example, similar to the earlier `hello` example. Note that `myEntity` is proxiable, because it implements the `myInterface` on line 10, which inherits `JistAPI.Proxiable` on line 5. The proxy separator is defined on line 13 using the `JistAPI.proxy` call with proxy.java

```
 1 import jist.runtime.JistAPI;
 2
 3 public class proxy
 4 {
 5   public static interface myInterface extends JistAPI.Proxiable
 6   {
 7     void myEvent();
 8   }
 9
10   public static class myEntity implements myInterface
11   {
12     private myInterface proxy =
13       (myInterface)JistAPI.proxy(this, myInterface.class);
14     public myInterface getProxy() { return proxy; }
15
16     public void myEvent()
17     {
18       JistAPI.sleep(1);
19       proxy.myEvent();
20       System.out.println("myEvent at t="+JistAPI.getTime());
21     }
22   }
23
24   public static void main(String args[])
25   {
26     myInterface e = (new myEntity()).getProxy();
27     e.myEvent();
28   }
29 }
```

Figure 7: An example illustrating a simple use of proxy entities.

the target proxiable instance and appropriate interface class. The invocations on this proxy, on lines 19 and 27, occur in simulation time.

Reflection

An important consideration in the design of many popular simulators is configurability, and the desire is to reuse the same simulation code for many different experiments. A few design approaches are possible. The first is source-level reuse. This approach involves the recompilation of the simulation program before each run with hard-coded simulation parameters and possibly also with a small driver program for simulation initialization. While the source-based approach is flexible and also runs efficiently, it is cumbersome to require recompilation on each run. A second approach is to use a configuration file that is parsed by a more sophisticated driver program as it initializes the simulation. This eliminates the need for recompilation in many circumstances, but it it is a brittle and inflexible option, since it is limited to pre-defined configuration options. A third approach is to integrate a scripting language into the simulation for purposes of configuration, an approach championed by ns2. The scripting language interpreter – Tcl, in the case of ns2, – is backed by the compiled simulation runtime, so that script variables are linked

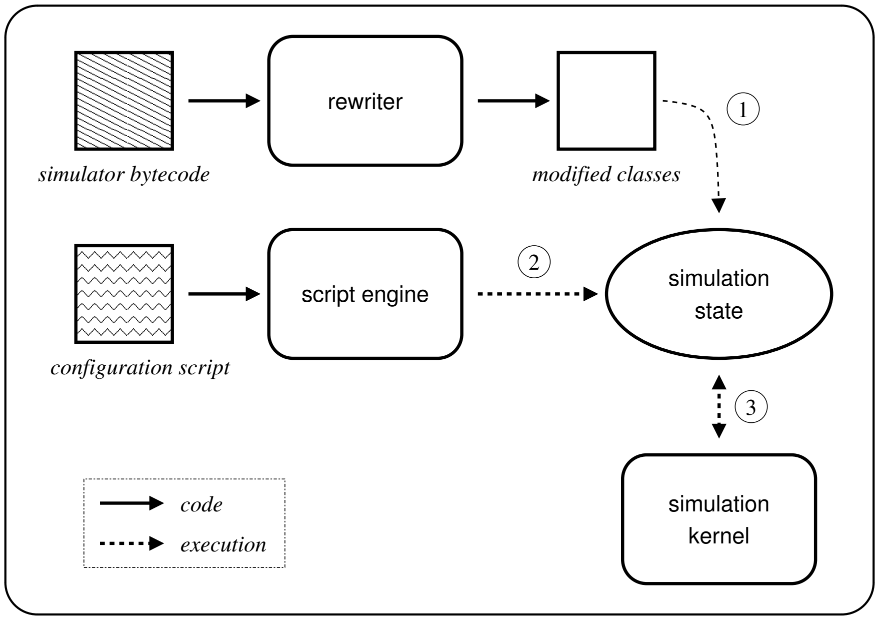


Figure 8: JiST can easily provide multiple scripting interfaces to configure its simulations without source modification, memory overhead, or loss of performance. (1) As before, the simulation classes are loaded and rewritten on demand. (2) The script engine configures the simulation, using reflection, and may even dynamically compile the script to bytecode for performance. (3) The simulation then runs, as before, interacting with the kernel as necessary.

to simulation values. The script is used to instantiate and initialize the various pre-defined simulation components. This is an attractive, flexible approach, since it allows the simulation to be configured with a script.

Unfortunately, the linkage between the compiled simulation components and the interpreted simulation configuration scripts is difficult to establish. It is achieved manually via a programming pattern called *split objects*, which requires interface functions that channel information in objects within the compiled space to and from objects in the interpreted space. This not only clutters the core simulation code, but it is also inefficient in that it duplicates information within the process memory. Furthermore, the script performance depends heavily on these interfaces. The choice of binding ns2 with Tcl, for example, requires excessive string manipulation, and leads to long configuration times.

In contrast, the scripting functionality comes “for free” in JiST. It does not require *any* additional code in the simulation components. It does not incur *any* additional memory overhead. And, the script can configure a simulation just as quickly as a custom driver program. This is all possible because Java is a dynamic language that supports reflection. A script engine can query and update simulation values by reflection, and can also dynamically pre-compile the driver script directly to bytecode for efficient execution. As illustrated in Figure 8, the access that the script engine has to the simulation state is just as efficient and expressive as the compiled driver program.

A number of robust Java scripting engines already exist. JiST supports the BeanShell engine, with its Java syntax, and also the Jython engine, which interprets Python scripts. If desired, other engines, such as Bistro (Smalltalk), Jacl (Tcl), JRuby (Ruby), Kawa (Scheme), Rhino (JavaScript) or a custom simulation definition language interpreter can easily be integrated as well, and they can even co-exist.

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

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

Figure 9: Basic example of BeanShell and Jython script-based simulation drivers. One can use Java-based scripting languages to configure and initialize simulations from existing components.

As is the case with compiled driver programs, the driver script is invoked within its host script engine by the JiST kernel. The script is the first, bootstrap event of the simulation. In addition, this functionality is exposed via the `JistAPI`, so that even simulation programs can embed configuration scripts that are specific to those simulations. For example, a wireless network simulator written atop JiST, called SWANS and described later, uses scripts to configure network topology and node protocol stacks.

Tight event coupling

Encoding an event-dispatch as a method invocation has a number of additional benefits, which can *significantly* reduce the amount of simulation code required, as well as improve its clarity, without affecting runtime performance. These benefits are summarized in Table 2. The first benefit of this encoding is type-safety, which eliminates a common source of error: the source and target of an event are statically checked by the Java compiler. The event type information is also managed automatically at runtime, which completely eliminates the many event type constants and associated event type-cast code that is otherwise required. A third benefit is that marshalling of event parameters is performed automatically. In contrast, simulators written against event-driven libraries often require a large number of event structures and much code to simply pack and unpack parameters from these structures. Finally, debugging event-driven simulators can be onerous, because simulation events arrive at target entities from the simulation queue without context. Thus, it can be difficult to determine the cause of a faulty or unexpected incoming event. In JiST, an event can automatically carry its context information: the point of dispatch (with line numbers, if source information is available) as well as the state of the source entity. These contexts can then be chained to form an event causality trace, the equivalent of a stack trace, which may be helpful in debugging. For performance reasons, this information is collected only in JiST's debug mode.

The tight coupling of event dispatch and delivery in the form of a method invocation also has important performance implications. Tight event-loops, which can be determined only at runtime, can be dynamically optimized across the kernel-simulation boundary. The tight coupling also abstracts the simulation event queue behind simulation time semantics. This, in turn, allows the JiST runtime to execute the simulation efficiently – in parallel and optimistically – without requiring any modifications to the simulation code. Also the distribution of simulation entities across different machines can occur transparently with respect to the running simulation.

type safety	-	source and target of event statically checked by compiler
event typing	-	not required; events automatically type-cast as they are dequeued
event structures	-	not required; event parameters automatically marshalled
debugging	-	event dispatch location and state available
execution	-	transparently allows for parallel, optimistic and distributed execution

Table 2: Benefits of tight event coupling achieved through simulation time method invocations

Continuations

Up to this point, we have described how entity method invocations can be used to model simulation events. This facility provides us with all the functionality of an explicit event queue, which is all that many existing simulators use, plus the ability to transparently execute the simulation more efficiently. However, it remains cumbersome to model simulation *processes*, since they must be written as event-driven state machines. While many entities, such as network protocols or routing algorithms, naturally take this event-oriented form, other kinds of entities do not. For example, an entity that models a file-transfer is more readily encoded as a process than as a sequence of events. Specifically, one would rather use a tight loop around a blocking send routine than dispatch *send* events to some transport entity, which will eventually dispatch *send-complete* events in return. To that end, we introduce *simulation time continuations*.

In order to invoke an entity method with continuation, the simulation developer merely needs to declare that a given entity method is *blocking*. Syntactically, an entity method is blocking if and only if it declares that it throws a `JistAPI.Continuation` exception. This exception is not actually thrown. It acts merely as a method tag, and as an indication to the JiST rewriter. Moreover, it need not be explicitly declared further up the call-chain, since it is a sub-class of `Error`, the root of an implicit exception hierarchy.

The semantics of a blocking entity method invocation, as shown in Figure 10, are a natural extension atop the existing event-based invocation. We first save the call-stack of the calling entity and attach it to the outgoing event. When the call event is complete, we notice that it has caller information, so we dispatch a callback event to the caller, with its continuation information. Thus, when the callback event is eventually dequeued, the state is restored and execution will continue right after the point of the blocking entity method invocation. In the meantime, however, the local simulation time has progressed to the simulation time at which the calling event was completed, and other events may have been processed against the entity.

This approach has a number of advantages. First, it allows blocking and non-blocking entity methods to co-exist. Methods can arbitrarily be tagged as blocking, and we extend the basic event structure to store the call and callback information. Secondly, there is no notion of an explicit process. Unlike process-oriented simulation runtime, which must allocate a fixed stack-space for each real or logical process, the JiST stack space is unified across all its active entities, reducing memory consumption. The continuation stacks actually exist in the heap along with the events that contain them, and any objects that they reference. Moreover, our model is actually closer to threading, in that multiple continuations can exist simultaneously for a single entity. Finally, there is no system context switching required. The parallelism occurs only in simulation time, so the underlying events may be executed sequentially within a single thread of control.

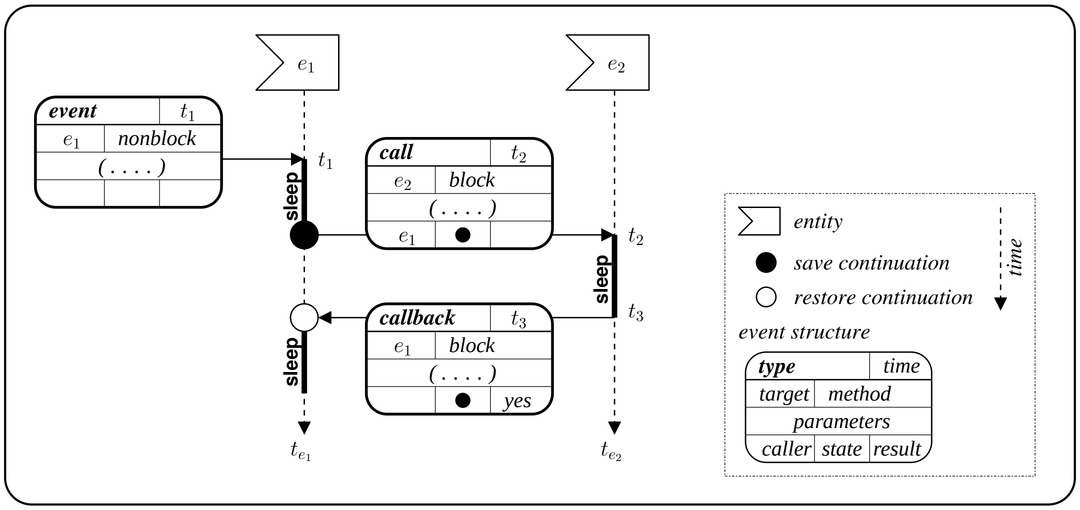


Figure 10: The addition of blocking methods allows simulation developers to regain the simplicity of process-oriented development. When a blocking entity method is invoked, the continuation state of the current event is saved and attached to a call event. When this call event is complete, we schedule a callback event to the caller. The continuation is restored and the caller continues its processing, albeit at a later simulation time.

The code listing in Figure 11 shows an entity with a single blocking method. Notice that `blocking` is a blocking method, since it declares that it may throw a `JistAPI.Continuation` exception on line 5. Otherwise, `blocking` would be a regular entity method, invoked in simulation time. The effect of the blocking tag is best understood by comparing the output for the program both with and without the blocking semantics, i.e. both with an without the blocking tag on line 5. Note how the timestamps are affected, in addition to the ordering of the events.

blocking	non-blocking
`> i=0 t=0`	`> i=0 t=0`
`> called at t=0`	`> i=1 t=0`
`> i=1 t=1`	`> i=2 t=0`
`> called at t=1`	`> called at t=0`
`> i=2 t=2`	`> called at t=0`
`> called at t=2`	`> called at t=0`

Unfortunately, saving and restoring the call-stack for the purposes of continuation is not a trivial task in Java [10]. The fundamental difficulty arises from the fact that stack manipulations are not supported at either the language, library or bytecode level. We have implemented and evaluated a number of ways to address this problem. The specific choice of implementation does not affect the execution semantics of continuations. However, understanding the underlying implementation may help simulation developers produce more efficient simulations.

Our implementation of continuations draws on ideas in the JavaGoX [11] and PicoThreads [3] projects, which also need to save the stack, albeit in a radically different context. We introduce an important new idea to these approaches, which eliminates the need to modify method signatures. This fact is significant, since it allows our implementation to function even across the standard Java libraries and enables us, for example, to run standard, cont.java

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

Figure 11: A basic entity which illustrates the use of a blocking method.

unmodified Java network applications directly over our wireless simulator. This design also eliminates the use of exceptions to carry state information and is considerably more efficient for our simulation needs.

Since we are not allowed access to the call-stack in Java, we instead convert parts of the original simulation program into a continuation-passing style (CPS). The first step is to determine which parts of the simulation code need to be transformed. For this purpose, the JiST rewriter incrementally produces a call-graph of the simulation at runtime as it is loaded, and uses the blocking method tags to compute all *continuable* methods. Continuable methods are those methods that could exist on a call stack at the point of a blocking entity method invocation. Or, more precisely, a continuable method is defined recursively as any method that contains:

- an *entity* method invocation instruction, whose target is a *blocking* method; or
- a regular *object* method invocation instruction, whose target is a *continuable* method.

Note that the continuable property does not spread recursively to the entire application, since the second, recursive half of the continuable definition does not cross entity boundaries.

Each method within the continuable set undergoes a basic CPS transformation, as shown in Figure 12. We scan the method for continuation points and assign each one a *program location number*. We then perform a data-flow analysis of the method to determine the types of the local variables and stack slots at that point, and use this information to generate a custom class that will store the continuation frame of this program location. These classes, containing properly typed fields for each of the local variables and stack slots in the frame, will be linked together to form the preserved stack. Finally, we insert both saving and restoration code for each continuation point. The saving code marshals the stack and locals into the custom frame object, and pushes it onto the event continuation stack via the kernel. The restoration code does the opposite, and then jumps right back to the point of the blocking invocation.

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
```

Figure 12: These pseudocode-bytecode listings show the basic CPS transformation that occurs on all continuable methods. The transformation instruments the method to allow it to either: a) save and exit, or b) restore and start; from any continuable invocation location.

All of this must be done in a type-safe manner. This requires special consideration not only for the primitive types, but also for arrays and null-type values. There are other restrictions that stem from the bytecode semantics. Specifically, the bytecode verifier will allow uninitialized values on the stack, but not in local variables or fields. The bytecode verifier will also not allow an initializer (constructor) to be invoked more than once. We eliminate both of these possibilities by statically verifying that no constructor is continuable.

Finally, the kernel functions as the continuation trampoline, as shown in Figure 13. When the kernel receives a request to perform a call with continuation, it registers the call information, switches to save mode, and returns to the caller. The stack then unwinds, and eventually returns to the event loop, at which point the call event is dispatched with the continuation attached. When the call event is received, it is processed, and a callback event is dispatched in return with both the continuation and the result attached. Upon receiving this callback event, the kernel switches to restore mode and invokes the appropriate method. The stack then winds up to its prior state, and the kernel receives a request for a continuation call yet again. This time, the kernel simply returns the result of the call event, and allows the event processing to continue from where it left off.

The performance of a blocking call with a short stack is only around 2-3 times slower than two regular events. Thus, continuations present a viable, efficient way to reclaim process-oriented simulation functionality within an event-oriented simulation framework.

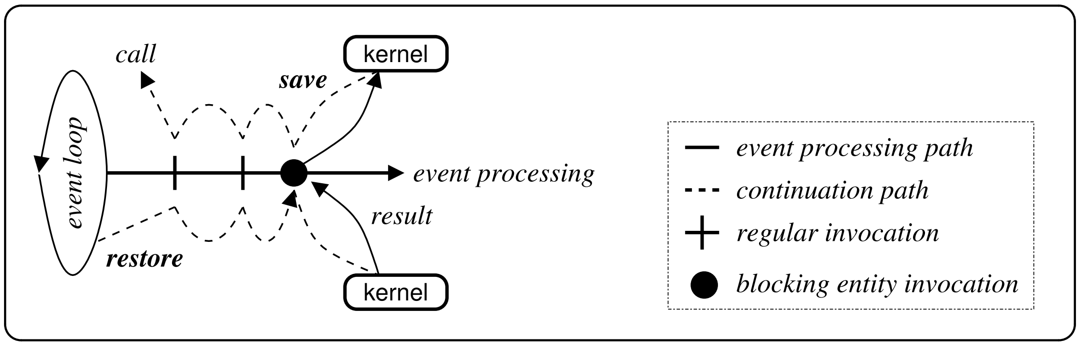


Figure 13: The JiST event loop also functions as a continuation trampoline. It saves the continuation state on a blocking entity method invocation, and restores it upon receiving the callback. Due to Java constraints, the stack must be manually unwound and preserved.

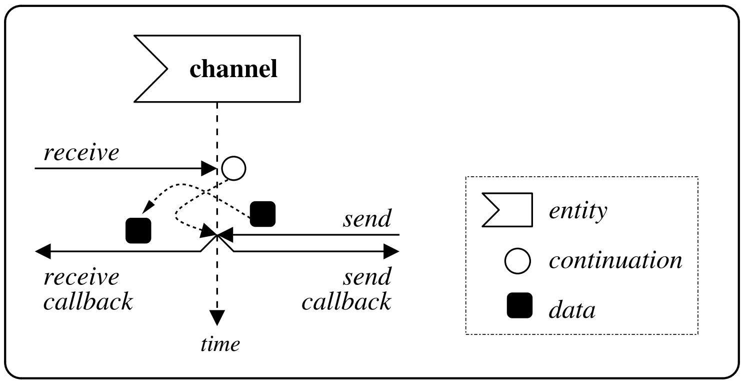


Figure 14: A blocking CSP Channel is built using continuations. Other simulation synchronization primitives can similarly be constructed.

Concurrency primitives

Using continuations, we also can recreate various concurrency primitives in simulation time. As an example, we construct the Channel primitive from Hoare's Communicating Sequential Processes (CSP) language [4]. Other primitives, such as threads, locks, semaphores, barriers, monitors, FIFOs, etc., can readily be built either atop Channels, or directly within the kernel.

As shown in Figure 14, the CSP channel blocks on the first receive (or send) call and stores the continuation. When the matching send (or receive) arrives, then the data item is transferred and control returns to both callers. A JiST channel is created via the `createChannel` system call. It supports both the CSP semantics, as well as non-blocking sends and receives.

The JiST API

Many of the important kernel functions have already been mentioned. For completeness, we include the full JiST application interface in Figure 15 and describe it below.

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) { }
}
```

Figure 15: The JiST application programming interface is exposed at the language level via the `JistAPI` object. The rewriter replaces default noop implementations of the various functions and interfaces with their *simulation time* equivalents.

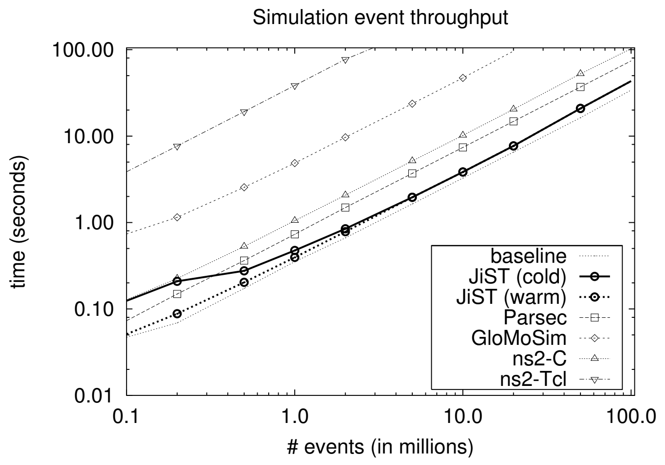
JiST API function	**explanation**
Entity interface	tags a simulation object as an entity, which means that invocations on this object follow simulation time semantics. e.g. `jist.swans.mac.MacEntity`.
Continuation exception	tags an entity method as blocking, which means that these entity method invocations will be performed in simulation time, but with continuation. e.g. `jist.swans.app.UdpSocket.receive(DatagramPacket)`.
Timeless interface	explicitly tags a simulation object as timeless, which means that it will not be changed across simulation time and thus need not be copied when transferred among entities. e.g. `jist.swans.node.Message`.
getTime()	returns the local simulation time. The local time is the time of the current event being processed plus any additional `sleep` time.
sleep(time)	advance the local simulation time.
end()	end simulation after the current time-step.
endAt(time)	end simulation after given absolute time.
THIS	entity self-referencing separator, analogous to Java `this` object self-reference. Should be type-cast before use.
ref(entity)	returns a separator of a given entity. All statically detectable entity references are automatically converted into separator stubs by the rewriter, so this operator should not be needed. It is included only to deal with rare instances of creating entity types dynamically, and for completeness.
Proxiable interface	an interface used to tag objects that may be proxied. It serves to improve proxying performance by eliminating the need for a relaying wrapper entity.

JiST API function	explanation
proxy(target, interface)	Returns a proxy separator for the given target object and interface class. The proxying approach depends on whether the target is an existing entity, a regular object or a proxiable object. The result is an object whose methods will be relayed to the target in simulation time.
proxyMany(target, interface[])	Same as proxy call, and allows for multiple interfaces.
run(type, name, args, prop)	Start a new simulation with given name and arguments at the current time. The supported simulation loader types are Java applications (RUN_CLASS), BeanShell scripts (RUN_BSH), and Jython scripts (RUN_JPY). The properties object carries simulation type-specific information directly to the simulation loader.
createChannel()	Create a new CSP Channel Entity.
setSimUnits(ticks, name)	Set the simulation time unit of measure and length in simulation ticks. The default is 1 tick.
CustomRewriter interface	Defines an installable rewriting phase.
installRewriter(rewriter)	Installs a custom class rewriting phase at the beginning of the JiST rewriting machinery. Used for simulation specific rewriting needs. For example, SWANS uses this interface to rewrite the networking library calls of applications to operate over the simulated network.

JiST performance

Based on conventional wisdom [1], one would argue against implementing the JiST system in Java, for performance reasons. In fact, the vast majority of existing simulation systems have been written in C and C++, or their derivatives. However, JiST actually out-performs the popular ns2 [7] and also the scalable GloMoSim [12] simulators, even on sequential benchmarks. Aggressive profile-driven optimizations combined with the latest Java runtimes result in a simulation system that can even match or exceed the performance of the highly-optimized Parsec [2] runtime! In this section, we present micro-benchmark results comparing JiST against other simulation systems.

The following measurements were taken on a 1133 MHz Intel Pentium III uni-processor machine with 128 MB of RAM and 512 KB of L2 cache, running the version 2.4.20 stock Redhat 9 Linux kernel with glibc v2.3. We used the publicly available versions of Java 2 JDK (v1.4.1), Parsec (v1.1.1), GloMoSim (v2.03) and ns2 (v2.26). Each data point shown represents an average of at least two runs, with deviations of less then 1% in all cases. All



5×10^6 events	time (sec)	vs. baseline	vs. JiST
baseline	1.640	1.0x	0.8x
JiST	**1.957**	**1.2x**	**1.0x**
Parsec	3.705	2.3x	1.9x
ns2-C	5.151	3.1x	2.6x
GloMoSim	23.720	14.5x	12.1x
ns2-Tcl	160.514	97.9x	82.0x

Figure 16: JiST has higher event throughput, and comes within 20% of the baseline lower bound. The kink in the JiST curve (at left) in the first fraction of a second of simulation is evidence of JIT compilation at work. The table (at right) shows the time to perform 5 million events, and also normalized against both the baseline and JiST performance.

tests were also performed on a second machine – a more powerful and memory rich dual-processor – with identical absolute memory or relative performance results.

Event throughput

Computational throughput is important for simulation scalability. Thus, in the following experiment, we measured the performance of each of the simulation engines in performing a tight simulation event loop. We began the simulations at time zero, with an event scheduled to generate another identical event for the subsequent simulation time step. We ran each simulation for n simulation time quanta, over a wide range of n, and measured the actual time elapsed.

Equivalent, efficient benchmark programs were written in each of the systems. The JiST program looks similar to the "hello world" program presented earlier. The Parsec program sends null messages among native Parsec entities using the special `send` and `receive` statements. The GloMoSim test considers the overhead of the node aggregation mechanism built over Parsec, which was developed to reduce the number of entities and save memory for scalability (discussed shortly). The GloMoSim test is implemented as an application component, that circumvents the node communication stack. Both the Parsec and GloMoSim tests are compiled using using `pcc -O3`, which is the most optimized setting. ns2 utilizes a split object model, allowing method invocations from either C or Tcl. The majority of the performance critical code, such as packet handling, is written in C, leaving mostly configuration operations for Tcl. However, there remain some important components, such as the mobility model, that depend on Tcl along the critical path. Consequently, we ran two tests: the ns2-C and ns2-Tcl tests correspond to events scheduled from either of the languages. ns2 performance lies somewhere between these two, widely divergent values, depending on how frequently each language is employed for a given simulation. Finally, we developed a baseline test to obtain a lower bound on the computation. It is a program, written in C and compiled with `gcc -O3`, that inserts and removes elements from an efficient implementation of an array-based heap.

The results are plotted in Figure 16. Please note the log-log scale of this and subsequent plots. As expected, all the simulations run in time linear with respect to n. An unexpected result, since Java is interpreted, is that JiST out-performs all the other systems, including the compiled ones. It also comes within 20% of the baseline measure of the lower bound. This is due to the impressive JIT dynamic compilation and optimization capabilities of the modern Java runtime. The optimizations can actually be seen as a kink in the JiST curve during the first fraction of a second of simulation. To confirm this, we warmed the JiST test with 10^6 events, and observed that the kink disappears. The table shows the time taken to perform 5 million events in each of the measured simulation systems, and these figures normalized against the baseline and the performance of JiST. JiST is twice as fast as both Parsec and ns2-C. GloMoSim and ns2-Tcl are one and two orders of magnitude slower, respectively.

Message-passing overhead

Alongside event throughput, it is important to ensure that inter-entity message passing scales well with the number of entities. For simplicity of scheduling, many (inefficient) parallel simulation systems utilize kernel threads or processes to model entities, which can lead to severe degradation with scale.

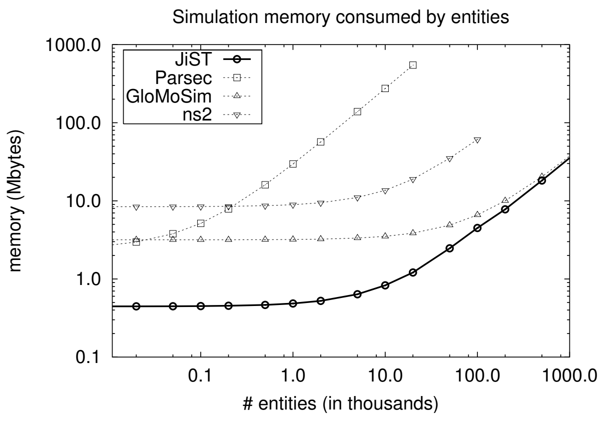
The systems that we have considered do not exhibit this problem. ns2 is a sequential simulator, so this issue does not arise. Parsec, and therefore also GloMoSim, models entities using logical processes implemented in user space and use an efficient simulation time scheduler. JiST implements entities as concurrent objects and also uses an efficient simulation time scheduler. The overheads of both Parsec and JiST were empirically measured. They are both negligible, and do not represent a scalability constraint.

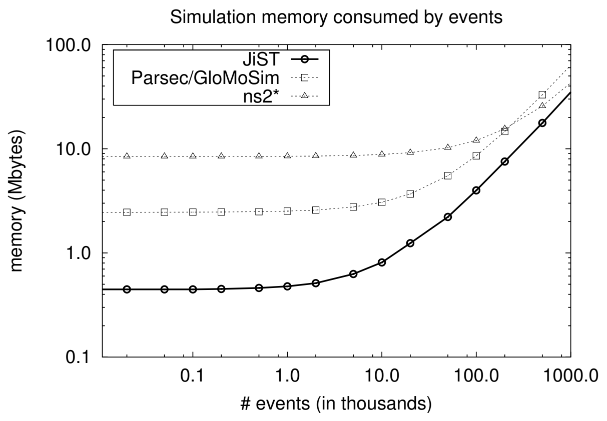
Memory utilization

Another important resource that limits scalability is memory. We, therefore, measured the memory consumed by entities and by queued events in each of the systems. Measuring the memory footprint of entities involves the allocation of n empty entities and observing the size of the operating system process, for a wide range of n. In the case of Java, we invoke a garbage collection sweep and then request an internal memory count. Analogously, we queue a large number of events and observe their memory requirements. The entity and event memory results are plotted in Figure 17. The base memory footprint of each of the systems is less than 10 MB. Asymptotically, the process footprint increases linearly with the number of entities or events, as expected.

JiST performs well with respect to memory requirements for simulation entities. It performs comparably with GloMoSim, which uses node aggregation specifically to reduce Parsec's memory consumption. A GloMoSim "entity" is merely a heap-allocated object containing an aggregation identifier and an event-scheduling heap. In contrast, each Parsec entity contains its own program counter and logical process stack, the minimum stack size allowed by Parsec being 20 KB. In ns2, we allocate the smallest split object possible, an instance of `TclObject`, responsible for binding values across the C and Tcl memory spaces. JiST achieves the same dynamic configuration capability without requiring the memory overhead of split objects.

JiST also performs well with respect to event memory requirements. Though they store slightly different data, the C-based ns2 event objects are exactly the same size as JiST events. On the other hand, Tcl-based ns2 events





memory	entity	event	10K nodes sim.
JiST	**36 B**	**36 B**	**21 MB**
GloMoSim	36 B	64 B	35 MB
ns2	544 B	36 B*	72 MB*
Parsec	28536 B	64 B	2885 MB

Figure 17: JiST allocates entities efficiently (at left): comparable to GloMoSim at 36 bytes per entity, and over an order of magnitude less that Parsec or ns2. JiST also allocates events efficiently (at right): comparable to ns2 (in C) at 36 bytes per queued event, and half the size of events in Parsec and GloMoSim. (*) Events scheduled in ns2 via Tcl will allocate a split object and thereby incur the same memory overhead as above. The table shows per entity and per event memory overhead, along with the system memory overhead for a simulation scenario of 10,000 nodes, *without* including memory for any simulation data. (*) Note that the ns2 split object model will affect its memory footprint more adversely than other systems when simulation data is added.

require the allocation of a new split object per event, thus incurring the larger memory overhead above. Parsec events require twice the memory of JiST events.

The memory requirements per entity, mem_{entity}, and per event, mem_{event}, in each of the systems are tabulated in Figure 17. We also compute the memory footprint within each system for a simulation of 10,000 nodes, assuming approximately 10 entities per node and an average of 5 outstanding events per entity. In other words, we compute: $mem_{sim} = 10^4 \times (10 \times mem_{entity} + 50 \times mem_{event})$. Note that these figures do not include the fixed memory base for the process, nor the actual simulation data. These are figures for empty entities and events alone, thus showing the *overhead* imposed by each system.

Note also that adding simulation data would doubly affect ns2, since it stores data in both the Tcl and C memory spaces. Moreover, Tcl encodes this data internally as strings. The exact memory impact thus varies from simulation to simulation. As a point of reference, regularly published results of a few hundred wireless nodes occupy more than 100 MB, and simulation researchers have scaled ns2 to around 1,500 non-wireless nodes using a process with a 2 GB memory footprint [9, 8].

Advantages of the JiST approach

Aside from performance and scalability, we have mentioned various benefits of the JiST design throughout the explanations. For reference, we summarize these advantages below.

application-oriented benefits

type safety	-	source and target of simulation events are statically type-checked by the compiler, eliminating a large class of errors
event types	-	numerous constants and the associated type-casting code are not required; events are implicitly typed
event structures	-	numerous data structures used to carry event payloads and the associated event marshalling code can be eliminated; event payloads are implicitly marshalled
debugging	-	event dispatch location and source entity state are available during event processing; can generate event causality trace to determine the origin of a faulty event

language-oriented benefits

reflection	-	allows script-based simulation configuration, debugging and tracing in a manner that is transparent to the simulation implementation
safety	-	allows for an object-level isolation of state between entities; ensures that all simulation time calls pass through the kernel; allows sharing of immutable objects; provides flexibility in entity state aggregation
garbage collection	-	memory for objects with long and variable lifetimes, such as network packets, is automatically managed; facilitates memory savings through sharing of immutable objects; avoids memory leaks and the need for complex memory protocols
Java	-	JiST reuses the standard language, libraries, compiler and runtime

system-oriented benefits

inter-process communication	-	since entities share the same process space, we pass a pointer, and there is no serialization; there is also no context switch required
Java-based kernel	-	allows cross-layer optimization between kernel and running simulation for faster event dispatch and system calls
robustness	-	strict Java verification ensures that simulations will not crash; garbage collection protects against memory leaks over time
rewriting	-	operates at the bytecode level; does not require source-code access
concurrency	-	simulation object model and execution semantics support parallel and optimistic execution transparently with respect to the application
distribution	-	provides a single system image abstraction that allows for dynamic entity migration to balance load, memory or network load

hardware-oriented benefits		
portability	-	pure Java: “runs everywhere”
cost	-	runs on COTS clusters (NOW, grid, etc.), as well as more specialized architectures

Scalable Wireless Ad hoc Network Simulator - SWANS

Both as a proof of the JiST approach and as a research tool, we built SWANS, a **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator. The SWANS software is organized as groups of software components that can be composed to form complete wireless simulations, as shown in Figure 18. There are components that implement different types of applications; networking, routing and media access protocols; radio transmission, reception and noise models; signal propagation and fading models; and node mobility. Instances of each component class are shown italicized in the figure.

Every SWANS component is encapsulated as a JiST entity: it stores it own local state, and interacts with other components via exposed entity interfaces. This pattern simplifies simulation development by reducing the problem to creating relatively small, event-driven components, which are then hooked up to form complete simulations. It also restricts state access and the degree of inter-dependence, allowing components to be readily interchanged with suitable alternate implementations, and for each simulated node to be independently configured.

It is important to note that communication among entities is very efficient. The design incurs no serialization cost and no copy cost among co-located entities, since these messages are merely Java objects that are passed along via the simulation time kernel by reference. This facilitates both vertical and horizontal aggregation of simulation state. The messages themselves are a chain of nested objects that mimic the data encapsulation of the network stack. This not only saves memory, and memory copies, but also provides an additional convenience in that it allows for the use of polymorphism in message processing.

The component design also restricts the simulation communication pattern. For example, application or link-layer components of different nodes do not communicate directly; they can only pass messages along their own node stacks. Consequently, the elements of the simulated node stack above the *Radio* layer become trivially parallelizable, and may be distributed with low synchronization cost.

In the following sections, we discuss each of the component packages: field, radio, link, routing, network, transport, application and common. We highlight the more important elements of the implementation. Further detail should be gleaned directly from the source code and JavaDoc pages.

Field

The SWANS field components are responsible for modeling signal propagation, pathloss and fading, as well as node mobility. Radios make transmission downcalls to the field, and other radios on the field receive reception upcalls, if they are within range of the signal. Both the pathloss and fading models are functions that depend on the source and destination radio locations. Pathloss models include free-space, two-ray and table-driven pathloss. Fading models include zero, Raleigh and Rician fading.

We have implemented a naïve signal propagation algorithm, which uses a slow, $O(n)$, linear search to determine the node set within the reception neighborhood of the transmitter. This is equivalent to the ns2 implementation.

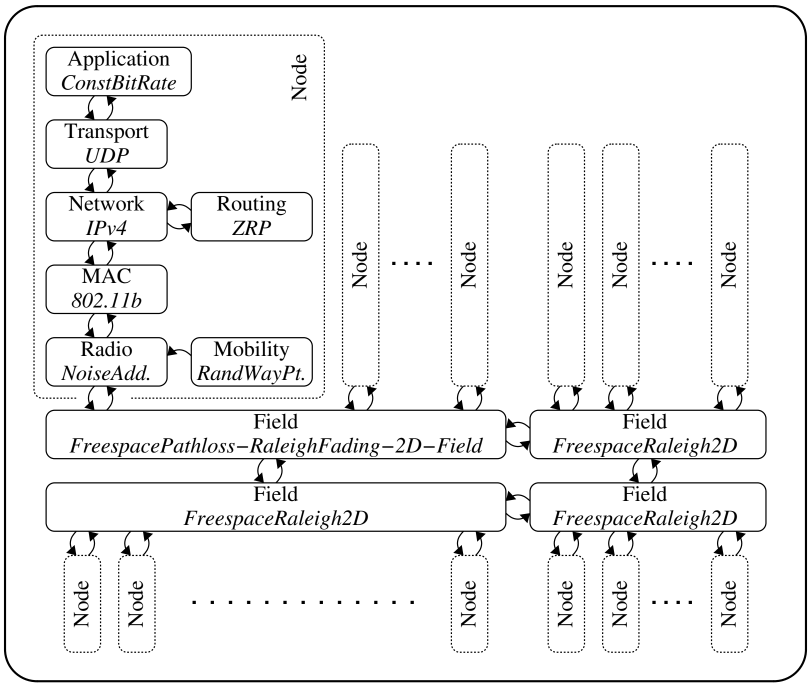


Figure 18: The SWANS simulator consists of event-driven components that can be configured and composed to form a meaningful wireless network simulation. Different classes of components are shown in a typical arrangement together with specific instances of component implementations in italics.

SWANS also supports a grid-based algorithm, as in GloMoSim. In addition, we have implemented a more advanced algorithm that uses *hierarchical* binning, an improvement on the *flat* binning algorithm implemented in GloMoSim and others.

In the flat binning approach, the field is sub-divided into a grid of node bins. A node location update requires constant time, since the bins divide the field in a regular manner. The neighborhood search is then performed by scanning all bins within a given distance from the signal source. While this operation is also of constant time, given a sufficiently fine grid, the constant is sensitive to the chosen bin size: bin sizes that are too large will capture too many nodes, and thus not serve their search-pruning purpose; bin sizes that are too small will require the scanning of many empty bins. The optimal bin size is proportional to the signal propagation radius, however this may change from radio to radio.

We improve on this approach. Instead of a flat sub-division, the hierarchical binning implementation *recursively* divides the field along both the x and y-axes. The node bins are the leaves of this balanced, spatial decomposition tree, which is of height equal to the number of divisions, or $log_2(\frac{field\ size}{bin\ size})$. This is similar to a quad-tree data structure, except that the division points are not the nodes themselves, but fixed coordinates. As before, updating a node location requires a constant number of operations, including the maintenance of inner node counts in the tree. However, this constant – the height of the tree – changes only logarithmically with changes in the bin size. Furthermore, since nodes move only a short distance between updates, the amortized height of the common parent of the two affected node bins is 1. The amortized number of inner nodes that require update is, therefore, also constant. When scanning for node neighbors, empty bins can be pruned as we descend spatially. Thus, the result can be computed in proportion to the size of the result. Since we will need to scan the result to propagate the signal, this is optimal; it will not benefit from function caching.

Finally, the node mobility is implemented as an interface-based discretized model. Upon each node movement, the model is queried to schedule the next movement. The mobility models that are implemented include static and random-waypoint.

jist.swans.field.*

interface	**class**	**description**
FieldInterface	Field	centralized node container that performs mobility and signal propagation with fading and pathloss
Fading	Fading.None	zero fading model
	Fading.Raleigh	Raleigh fading model
	Fading.Rician	Rician fading model
Pathloss	Pathloss.FreeSpace	pathloss model based purely on distance
	Pathloss.TwoRay	pathloss model that incorporates ground reflection
Spatial	Spatial.Linear	signal propagation and location update performed via linked list of radios
	Spatial.Grid	as above, but performed using a more efficient flat grid structure of small “Linear” bins
	Spatial.HierGrid	as above, but performed using a more consistently efficient hierarchical grid structure
Placement	Placement.Random	uniformly random initial node placement
Mobility	Mobility.Static	no mobility
	...RandomWaypoint	random waypoint mobility model

Radio

The SWANS radio receives upcalls from the field entity, and passes successfully received packets on to the link layer. It also receives downcalls from the link layer entity and passes them on to the field for propagation. We have implemented an independent interference radio, as in ns2, as well as an additive interference radio, as in GloMoSim. The independent interference model considers only signals destined for the target radio as interference. The additive model correctly considers all signals as contributing to the interference. Both radios are half-duplex. Radios are parameterized by frequency, transmission power, reception sensitivity and threshold, antenna gain, bandwidth and error model. Error models include bit-error rate and signal-to-noise threshold.

jist.swans.radio.*

interface	**class**	**description**
RadioInterface	RadioNoiseIndep	interference at radio consists only of other signals above a threshold that are destined for that same radio
	RadioNoiseAdditive	interference consists of all signals above a threshold
none	RadioInfo	unique and shared radio parameters

Link

The SWANS link layer entity receives upcalls from the radio entity and passes them to the network entity. It also receives downcalls from the network layer and passes them to the radio entity. The link layer entity is responsible for the implementation of a chosen medium access protocol, and for encapsulating the network packet in a frame. Link layer implementations include IEEE 802.11b and a "dumb" protocol. The 802.11b implementation includes the complete DCF functionality, with retransmission, NAV and backoff functionality. It does not include the PCF (access-point), fragmentation or frequency hopping functionality found in the specification, which is on par with the GloMoSim and ns2 implementations. The "dumb" link entity will only transmit a signal if the radio is currently idle.

jist.swans.mac.*

interface	**class**	**description**
MacInterface	MacDumb	transmits only if transceiver is idle
	Mac802_11	802.11b implementation
none	MacAddress	mac address
	MacInfo	unique and shared mac parameters

Network

The SWANS network entity receives upcalls from the link entity and passes them to the appropriate packet handler, based on the packet protocol information. The SWANS network entity also receives downcalls from the routing and transport entities, which it enqueues and eventually passes to the link entity. Thus, the network entity is the nexus of multiple network interfaces and multiple network packet handlers. The network interfaces are indexed sequentially from zero. The packet handlers are associated with IETF standard protocol numbers, but are mapped onto a smaller index space, to conserve memory, through a dynamic protocol mapper that is shared across the entire simulation. Each network interface is associated with a packet queue, which can handle packet priorities and perform RED. The packets are dequeued and sent to the appropriate link entity using a token protocol to ensure that only one packet is transmitted at a time per interface. The network layer sends packets to the routing entity to receive next hop information, and allows the routing entity to peek at all incoming packets. It also encapsulates message with the appropriate IP packet header. The network layer uses an IPv4 implementation. Loopback and broadcast are implemented.

jist.swans.net.*

interface	**class**	**description**
NetInterface	NetIp	IPv4 implementation
none	NetAddress	network address
	MessageQueue	outgoing message queues

Routing

The routing entity recieves upcalls from the network entity with packets that require next-hop information. It also receives upcalls that allow it to peek at all packets that arrive at a node. It sends downcalls to the network entity with next-hop information when it becomes available. SWANS implements the Zone Routing Protocol (ZRP), Dynamic Source Routing (DSR), and ad hoc On-demand Distance Vector Routing (AODV).

jist.swans.route.*

interface	**class**	**description**
RouteInterface	RouteZrp	Zone Routing Protocol
	RouteDsr	Dynamic Source Routing protocol
	RouteAodv	Ad hoc On-demand Distance Vector routing protocol

Transport

The SWANS transport entity receives upcalls from the network entity with packets of the appropriate network protocol, and passes them on to the appropriate registered transport protocol handler. It also receives downcalls from the application entity, which it passes on to the network entity. The two implemented transport protocols are UDP and TCP, which encapsulate packets with the appropriate packet headers. UDP socket, TCP socket and TCP server socket implementations actually exist within the application entity. The primary reason for this decision is that these implementations are modeled after corresponding Java classes, which force the use non-timeless objects. The `DatagramSocket`, for example, uses a mutable `DatagramPacket` to provide data. In all other respects, including correctness and performance, this decision, to move the socket implementations into the application entity, is inconsequential.

SWANS installs a rewriting phase that substitutes identical SWANS socket implementations for the Java equivalents within node application code. This allows existing Java networking applications to be run as-is over the simulated SWANS network. The SWANS implementations use continuations and a blocking channel in order to implement blocking calls. The entire application is conveniently frozen, for example, at the point that it calls `receive` until its packet arrives through the simulated network. Thus, we have a powerful Java simulation "sandwich": Java networking applications running over SWANS, running over JiST, running within the JVM.

There is an interesting complexity in this transformation that is worth mentioning. As discussed previously, since constructors can not be invoked twice, they may not be continuable. However, certain socket constructors, such as a TCP socket, have blocking semantics, since they require a connection handshake. We circumvent this problem by rewriting constructor invocations into two separate invocations. The internal socket implementation has a non-blocking constructor, which does nothing more than store the initialization arguments, and a second blocking method that will always be called immediately after the constructor. This second method can safely perform the required blocking operations.

jist.swans.trans.*

interface	**class**	**description**
TransInterface	TransUdp	UDP implementation, usually interacts with *`jist.swans.app.net.UdpSocket`*.
	TransTcp	TCP implementation, usually interacts with *`jist.swans.app.net.TcpServerSocket`* and *`.TcpSocket`* and various blocking streams implementations in *`jist.swans.app.io.*`*

Application

At the top of our network stack, we have the application entities, which make downcalls to the transport layer and receive upcalls from it, usually via SWANS sockets or streams that mimic their Java equivalents. The most generic, and useful kind of application entity is a harness for regular Java applications. These Java applications operate within a context that includes the correct underlying transport implementation for the particular node. Thus, these Java applications can open regular communication sockets, which will actually transmit packets from the appropriate simulated node, through the simulated network. SWANS implements numerous socket and stream types in the `jist.swans.app.net` and `jist.swans.app.io` packages. Applications can also connect to lower-level entities. The heartbeat node discovery application, for example, operates at the network layer. It circumvents the transport layer and communicates directly with the network entity.

jist.swans.app.*

interface	**class**	**description**
AppInterface	AppJava	versatile application entity that allows regular Java network applications to be executed within SWANS
	AppHeartbeat	runs heartbeat protocol for node discovery
item	**package**	**implementations**
socket	net	UdpSocket, TcpServerSocket, TcpSocket
stream	io	InputStream, OutputStream, Reader, Writer, InputStreamReader, OutputStreamWriter, BufferedReader, BufferedWriter

Common

There are various interfaces that are common across a number of SWANS layers and tie the system together. The most important interface of this kind is *Message*. It represents a packet transfered along the network stack, and it must be timeless (or immutable). Components at various layers define their own message structures. Many of these instances recursively store messages within their payload, thus forming a message chain that encodes the hierachical header structure of the message. Other common elements include a node, node location, protocol number mapper, and miscellaneous utilities.

jist.swans.misc.*

interface	**package**	**implementations**
Message	jist.swans.misc	MessageBytes, MessageNest
	jist.swans.mac	Mac802_11.RTS, .CTS, .ACK, .DATA, etc.
	jist.swans.net	NetIp.IpMessage
	jist.swans.route	RouteZrp.IARP, RouteDsr.RREQ, etc.
	jist.swans.trans	TransUdp.UdpMessage, TransTcp.TcpMessage, etc.

Code overview

For readers that intend to work on the project, this section explains the general organization of the codebase. The software distribution contains the JiST system, small regression test and benchmark simulations, the SWANS components, and simulation driver scripts. The software map below contains only the more significant files and is intended to be a navigational aid. It is correct as of September 2003.

	files	**classes**	**lines**
JiST	26	65	9302
minisim	10	22	1494
SWANS	52	113	12716
driver	8	8	862
	97	211	24556

`src` – source code

- `jist/runtime` – JiST simulation framework

rewriting	-	`Rewriter`, `RewriterFlow`, `RewriterVerify`, `ClassTraversal`
kernel	-	`Main`, `Controller`
events	-	`Event`, `EventLocation`, `Bootstrap`, `Heap`
entities	-	`EntityRef`, `Entity`, `ProxyEntity`
system calls	-	`JistAPI`, `JistAPI_Impl`
distributed	-	`ControllerRemote`, `Group`

- `jist/minisim` – regression and benchmark simulations
- `jist/swans` – wireless ad hoc network simulation components for SWANS
  - `jist/swans/field` – signal propagation, fading, pathloss and mobility
  - `jist/swans/radio` – radio signal transmission and reception
  - `jist/swans/mac` – link layer implementations
  - `jist/swans/net` – network layer implementations
  - `jist/swans/route` – routing layer implementations
  - `jist/swans/trans` – transport layer implementations
  - `jist/swans/app` – applications
  - `jist/swans/app/net` – application sockets
  - `jist/swans/app/io` – application streams
  - `jist/swans/misc` – common data structures and utilities

rewriting	-	`Rewriter`
runtime	-	`Main`

- `driver` – SWANS simulation drivers

`bin` – execution scripts

`lib` – required libraries

`memprof` – aggregating memory profiler

`bench` – Parsec, GloMoSim and ns2 benchmarks

References

[1] D. Bagley. The great computer language shoot-out, 2001. `http://www.bagley.org/~doug/shootout/`.

[2] R. L. Bagrodia, R. Meyer, M. Takai, Y. Chen, X. Zeng, J. Martin, and H. Y. Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, Oct. 1998.

[3] A. Begel, J. MacDonald, and M. Shilman. PicoThreads: Lightweight threads in Java. Technical report, UC Berkeley, 2000.

[4] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[5] D. R. Jefferson, B. Beckman, F. Wieland, L. Blume, M. D. Loreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. J. Wedel, H. Younger, and S. Bellenot. Distributed simulation and the Time Warp operating system. In *Proceedings of 12th ACM Symposium on Operating Systems Principles*, pages 77–93, Nov. 1987.

[6] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[7] S. McCanne and S. Floyd. NS (Network Simulator) at `http://www-nrg.ee.lbl.gov/ns`, 1995.

[8] D. M. Nicol. Comparison of network simulators revisited, May 2002.

[9] G. Riley, R. M. Fujimoto, and M. A. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of 7th Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, Mar. 1999.

[10] T. Sakamoto, T. Sekiguchi, and A. Yonezawa. Bytecode transformation for portable thread migration in Java. In *Proceedings of the 4th International Symposium on Mobile Agents*, 2000.

[11] T. Sekiguchi, T. Sakamoto, and A. Yonezawa. Portable implementation of continuation operators in imperative languages by exception handling. *Lecture Notes in Computer Science*, 2022:217+, 2001.

[12] X. Zeng, R. L. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, May 1998.