JiST: Embedding Simulation Time into a Virtual Machine

Rimon Barr
barr@cs.cornell.edu

Zygmunt J. Haas
haas@ee.cornell.edu

Robbert van Renesse
rvr@cs.cornell.edu

Computer Science and Electrical Engineering
Cornell University, Ithaca NY 14853

ABSTRACT

Since progress in many avenues of science depends heavily on simulated results, discrete event simulators have been the subject of much research into their efficient design and execution. This paper introduces JiST, a Java-based simulation framework that executes discrete event simulations both efficiently and transparently. Our system differs from existing work in that it embeds *simulation time* semantics into the Java execution model, but does so without inventing a new language, without requiring a specialized compiler and without utilizing a custom runtime. The result is a flexible simulation environment that allows sequential simulation execution and also transparently supports both parallel and optimistic execution with automatic checkpointing and rollback. The JiST approach uses a convenient single system image abstraction across a cluster of nodes, that allows for dynamic network and computational load-balancing and fine-grained migration of simulation state. The system provides standard benefits that the modern Java runtime affords, such as type-safety, garbage collection and portability. Nevertheless, JiST performs well, either matching or exceeding the performance of ns2 and the highly optimized GloMoSim runtime in both throughput and memory consumption. We illustrate the practicality of the JiST framework by applying it to the construction of SWANS, a scalable wireless ad hoc network simulator.

Keywords

discrete event simulation, simulation languages and environments, parallel and distributed simulation, Java, wireless networks

1. INTRODUCTION

From physics to biology, from forecasting the weather to predicting the performance of a new processor design, researchers in many avenues of science increasingly depend on software simulations to accurately model both realistic phenomena as well as hypothetical scenarios. Simulation facilitates the analysis of complex systems that may not be satisfactorily expressed analytically, nor easily reproduced and observed empirically, if at all.

For instance, and also as a running example for the remainder of this paper, the recent research focus of the networking and distributed systems communities on wireless ad hoc networks has fundamentally depended on simulation. Analytically quantifying the performance and complex behavior of even simple protocols in the aggregate is often imprecise. On the other hand, performing actual experiments is onerous: acquiring hundreds of devices, managing their software and configuration, controlling a distributed experiment and aggregating the data, possibly moving the devices around, finding the physical space for such an experiment, isolating it from interference and generally ensuring *ceteris paribus* are but some of the difficulties that make empirical endeavors daunting. Consequently, the vast majority of publications in this area are based entirely upon simulation.

Simulating an ad hoc network, or any other phenomenon, requires an accurate computational model that is nevertheless efficient. The fundamental trade-off is to develop a sufficiently abstract representation of the state and of the state changes that can nonetheless produce meaningful, reliable results. For example, simulation time can often be discretized to produce discrete event simulations, which can be readily encoded as event-driven programs. Events are time-stamped messages, processed in their temporal order. Processing an event involves updating the simulation state according to the given model, and potentially scheduling more events in the future.

Due to their popularity and widespread utility, discrete event simulators have been the subject of much research into their efficient design and execution (surveyed in [46, 22, 49, 24]). From a systems perspective, researchers have built many types of simulation libraries or execution runtimes spanning the gamut from the conservatively parallel to the aggressively optimistic, and from the shared memory to the message passing paradigms. From a modeling perspective, researchers have designed numerous simulation languages that codify event causality, execution semantics and simulation state constraints, which both simplify parallel simulation development and permit important static and dynamic optimizations.

Yet, despite a plethora of ideas and contributions to theory, languages and systems, the parallel simulation community has repeatedly asked itself "will the field survive?" under a perception that it had "failed to make a significant impact in the general simulation community," (see [23, 47, 8]

MSWiM '03 San Diego, California USA

and others). Even though a number of parallel discrete event simulation (PDES) environments have been shown to scale beyond 10^4 nodes [57], slow sequential simulators remain the norm. In particular, most published ad hoc network results are based on simulations of few nodes (usually only around 200 nodes), for a short duration (frequently just 90 seconds), and over a limited field. Larger simulations usually compromise on simulation detail or restrict node mobility.

These observations influenced the directions of the JiST project. Specifically, we decided to:

- *not* invent a simulation language – new languages, and especially domain-specific ones, are rarely adopted by the broader community;
- *not* create a simulation library – libraries often require developers to litter their code with PDES-specific, non-portable library calls and impose unnatural program structure to achieve performance and concurrency; and
- *not* develop a new language runtime – custom runtimes are rarely as optimized, reliable or portable as generic runtimes.

Instead, we hoped to bring simulation time execution semantics to a modern and popular virtual machine.

1.1 Objectives

JiST, which stands for **J**ava **i**n **S**imuation **T**ime, is a new discrete event simulation system that integrates some of the prior systems and languages approaches. Specifically, the key motivation behind JiST is to create a simulation system that can execute discrete event simulations both *efficiently* and *transparently*, yet to achieve this using only a *standard* systems language and runtime, where:

- **efficient** refers to a simulation runtime that compares favorably with existing, highly optimized simulation engines; the ability to execute a given simulation program in parallel and optimistically; dynamically optimizing the configuration of the simulation across the available computational resources to improve processing throughput, and; reasoning about simulation state and event causality constraints to improve throughput.
- **transparent** implies that simulation programs are automatically transformed to run with *simulation time* semantics; simulations are instrumented to support the various concurrency, consistency and reconfiguration protocols necessary for efficient sequential, parallel or speculative execution, without requiring programmer intervention or calls to specialized simulation libraries.
- **standard** denotes writing simulations in a *conventional* programming language and running these programs over a conventional runtime, where the term conventional describes a commonly used systems programming language, as opposed to a domain-specific language designed explicitly for simulation.

These three goals – the last one in particular – highlight an important distinction between JiST and previous simulation systems in that the simulation code that runs on JiST need not be written in a domain-specific language invented specifically for writing simulations, nor need it be littered with special-purpose system calls and call-backs to support concurrency, serialization, distribution or dynamic reconfiguration protocols. Instead, JiST transparently provides the performance benefits of parallel and optimistic simulation execution to simulation programs written in plain Java over an unmodified Java virtual machine. It is also important to clarify that JiST is *not* intended to simulate the execution of arbitrary Java programs. Rather, it is a simulation framework that can transparently and efficiently execute simulation programs over the Java platform.

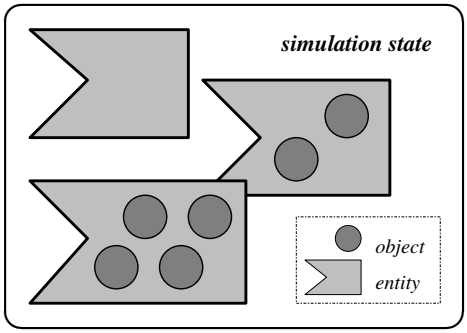

Figure 1: Simulation programs are partitioned into entities along object boundaries. Thus, entities do not share any application state and can independently progress through simulation time between interactions.

A more detailed discussion regarding the benefits of the JiST approach, including the choice of Java, various runtime design decisions and the use of JiST as a research platform, is left to Section 5. Next, we describe the design and implementation in Sections 2 and 3, followed by a performance evaluation of the system in Section 4. We discuss the related and prior work in Section 6.

2. DESIGN

The purpose of the JiST system is to run discrete event simulations both *efficiently* and *transparently* using a *standard* systems language and runtime. In this section, we elaborate on what it means to execute a program in *simulation time*, and provide an overview of how the JiST system supports this abstraction as well as wireless network simulation.

2.1 Simulation time execution

The Java virtual machine (JVM) [39] is a stack-based Java byte-code execution engine. 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 a garbage collection sweep 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

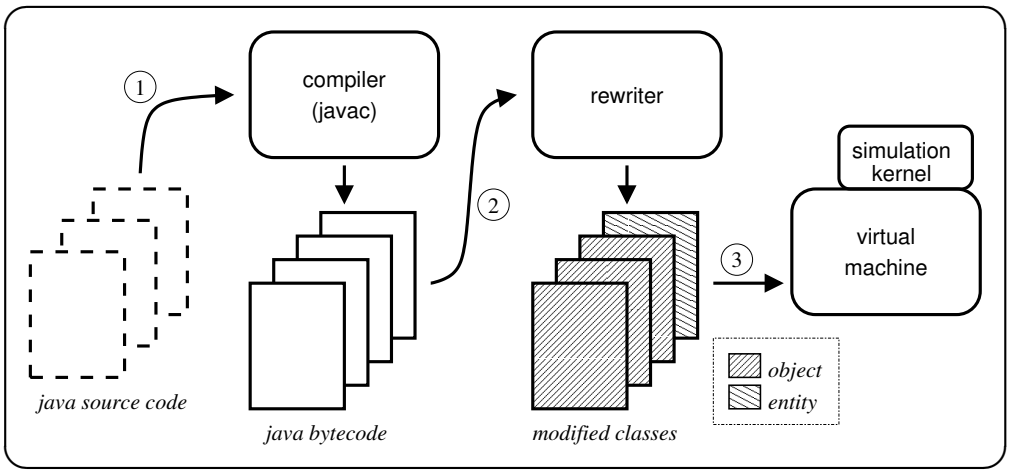

Figure 2: The JiST system architecture – simulations are (1) compiled, then (2) dynamically instrumented by the rewriter and finally (3) executed. The compiler and virtual machine are standard Java language components. Simulation time semantics are introduced by the rewriter, and supported at runtime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

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 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.

The notion of *simulation time* itself is not new: simulation program writers have long been accustomed to explicitly track the simulation time and explicitly schedule simulation events in time-ordered queues [45]. The simulation time concept is also integral to a number of simulation languages and simulation class libraries. The novelty of the JiST system lies in that it embeds the simulation time semantics into the Java execution model, which allows the system to transparently run the resulting simulations efficiently. The structure of these simulation programs is now described.

2.2 Simulation programs

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 within individual objects. These objects communicate by passing messages, 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 *entities*. Entities logically encapsulate application objects (Figure 1) and serve to demarcate independent simulation components. Every application object is contained within exactly one entity, and simulation events are then intuitively represented as method invocations among these entities. This is a convenient abstraction that not only eliminates the need for an explicit simulation event queue, but also enforces a clean partitioning of the simulation state.

To enforce this simulation partitioning 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 means that all references to an object must originate either directly or indirectly from a single entity. This condition suffices to ensure simulation partitioning, because Java is a safe language.

JiST manages a simulation at the granularity of its entities. Instructions and method invocations *within* an entity follow the regular Java 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 return the current simulation time and advance it, respectively.

In contrast, invocations *across* entities are executed in simulation time. This means that an invocation is performed on the callee entity when its state is at the same simulation time as the calling entity. Thus, cross-entity method invocations act as synchronization points in simulation time. The invocations are non-blocking. They are queued in simulation time order and are performed without continuation. Because of the simulation partitioning, interactions among entities can only occur via the JiST infrastructure.

Since entities do not share any application state, each entity can actually progress through simulation time indepen-

dently between interactions. Thus, by tracking the simulation time of each entity individually, the model can support concurrent execution. Each entity e_i represents a tuple $\langle state_i, time \rangle$, and the state of the entire simulation at any given time is the union of the state of all its entities at that time: $state(t) = \cup \langle state_i, t \rangle$. By adding entity checkpointing, the model can even support speculative execution.

The role of the simulation programmer 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. For example, a wireless simulation would be constructed from entities for nodes, radios, routing protocols, etc., as discussed in Section 5.4. The JiST infrastructure will then transparently and efficiently execute the program with simulation time semantics.

The JiST model of execution, known as the *concurrent object model*, is similar to, for example, the Compose [43] simulation library. It invokes a method for every message received, and executes it to completion. This is in contrast to the *process model* that is used, for example, in Parsec [8], wherein explicit send and receive operations are interspersed in the code. In the process model, each entity must store a program-counter and a stack as part of its state. Unlike Compose, message sending in JiST is embedded in the language and does not require a class library. Unlike Parsec, JiST operates within regular Java syntax, without requiring new language constructs and compilers.

2.3 Wireless network simulations

Although JiST can efficiently support general-purpose discrete event simulations, one of the primary motivations for its creation was for the scalable simulation of wireless ad hoc networks. It out-performs existing solutions in this space, owing to a number of design decisions, which we contrast with two popular solutions: ns2 [45] and GloMoSim [65].

The ns2 network simulator has a long history with the networking community, is widely trusted, and has been extended to support mobility and wireless networking protocols. It is designed as a monolithic, sequential simulator. ns2 uses a clever "split object" design, which allows Tcl-based script configuration of C-based object implementations. This approach is convenient for users. However, it incurs a substantial memory overhead and increases the complexity of simulation code. Researchers have extended ns2 to conservatively parallelize its event loop. However, this technique has proved primarily beneficial for distributing ns2's considerable memory requirements. Based on numerous published results, it is not easy to scale ns2 beyond a few hundred simulated nodes. Simulation researchers have shown ns2 to scale, with difficulty and substantial hardware resources, to simulations of a few thousand nodes.

GloMoSim is a newer simulator written in Parsec that has recently gained popularity within the wireless ad hoc networking community. It was designed specifically for scalable simulation by explicitly supporting efficient, conservatively parallel execution with lookahead. The sequential version of GloMoSim is freely available. The conservatively parallel version has been commercialized as QualNet. Due to Parsec's large per-entity memory requirements, GloMoSim implements a technique called "node aggregation," wherein state of multiple simulation nodes are multiplexed in a single Parsec entity. While this effectively reduces memory consumption, it incurs a performance overhead and also increases code complexity. The aggregation of state also renders speculative execution techniques impractical. GloMoSim has been shown to scale to 10,000 nodes on large, specialized multi-processor machines.

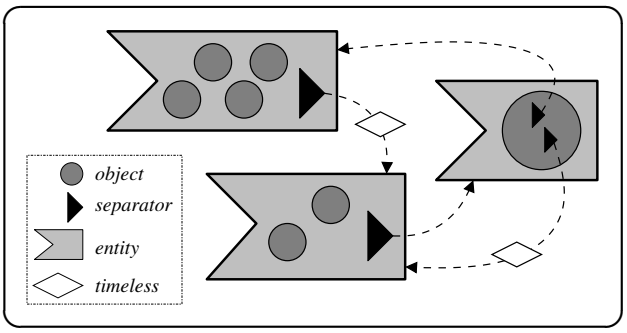

Figure 3: The rewriter partitions applications into entities. Entities reference one another only through separator stubs, and communicate by sending messages via the simulation time kernel.

In contrast, JiST is designed to transparently execute simulations in a concurrent, distributed and also speculative fashion, thereby achieving scalability. Wireless network simulations are particularly well-suited to optimistic execution models, since wireless nodes contain independent simulation state and the likelihood that one node causes a simulation rollback at another node decreases with the distance between them. Speculative execution also lowers the synchronization cost, which enables simulations to be distributed across a cluster of networked machines. We will expand on this theme in Section 5.

2.4 System architecture

The JiST system consists of four distinct components: a compiler, language runtime or virtual machine, rewriter and simulation time kernel. Figure 2 presents the JiST architecture pictorially. A simulation is first compiled, then dynamically rewritten at application load time, and finally executed by the virtual machine with support from the simulation time kernel.

A primary goal of JiST is to execute simulations efficiently and transparently, using only a standard language and runtime. Consequently, the compiler and runtime components of the JiST system can be any standard Java compiler and Java virtual machine, respectively. We use the Java 2 v1.4 implementation, both the `javac` and `jikes` compilers, and the Sun HotSpot Java Virtual Machine v1.4.1 on Linux and Windows. Simulation time execution semantics are introduced by the two remaining system components.

The *rewriter* component of JiST is a dynamic class loader. It intercepts all class load requests, and subsequently verifies and modifies the requested classes. These modified, rewritten classes now incorporate the embedded simulation time operations, but otherwise completely preserve the existing program logic. The program transformations occur once, at load time, and do not incur rewriting overhead during execution.

At runtime, the modified classes interact closely with the *simulation time kernel* through the various injected or modi-

class	Object		Timeless		Entity		total	
base size	28947		2940		2177		34064	
total increase	6908	(23.9%)	816	(27.8%)	4025	(184.9%)	11749	(34.5%)
constant pool	4908	(17.0%)	436	(14.8%)	3006	(138.1%)	8350	(24.5%)
code, etc.	2000	(6.9%)	380	(12.9%)	1019	(46.8%)	3399	(10.0%)

Figure 4: Rewriter processing increases class sizes. The figures shown above are the increases in bytes (and as a percentage), from the processing of the complete SWANS code-base. The data is split into the three JiST class categories, showing that the majority of the increase occurs among entity classes, and that much of the increase is due to new constant pool entries.

fied operations. The kernel is responsible for all the runtime aspects of the simulation time abstraction. For example, it maintains the simulation time of each entity, ensures proper synchronization, implements efficient checkpointing or rollback and balances load on behalf of the running simulation application. It can be implemented with a variety of concurrency and synchronization models. The kernel can also integrate with external software components designed for high-performance parallel distributed applications (such as MPI [29] or PVM [61]) and leverage specialized hardware when available.

3. IMPLEMENTATION

The notable pieces of the JiST system are the byte-code rewriter and the simulation time kernel, since these components introduce and support the simulation time execution semantics, respectively. We describe their implementation in detail. We then describe the JiST API, which exposes the execution semantics at the language level, and provide a simple "Hello world!" for illustration. Finally, we describe the manner in which simulations written to this API can be transparently executed both in parallel and optimistically.

3.1 Rewriter

The purpose of the rewriting step is to transform the JiST instructions embedded within the compiled simulation program into code with the appropriate simulation time semantics, respectively. The result is a partitioned application, as depicted in Figure 3, in which entities encapsulate private state, reference other entities only through separator stubs, and communicate with one another only via the simulation time kernel. The basic design of the rewriter is that of a multi-pass visitor over the class file structure, traversing and possibly modifying the class, its fields and methods, and their instructions, based on the set of rules summarized below.

The rewriter first verifies an application by performing byte-code checks, in addition to the standard Java verifier, that are specific to simulations. Specifically, it ensures that all classes that are tagged as entities conform to entity restrictions: the fields of an entity must be non-public and non-static; all public methods should be concrete and should return `void`; and some other minor restrictions. These ensure that the state of an entity is completely restricted to its instance, and also allow entity methods to be invoked without continuation, as per simulation time semantics.

Conforming to the earlier-stated goal of partitioning the application state, entities are never referenced directly by other entities. This isolation is achieved by the insertion of stub objects, called *separators*. The rewriter also adds a self-referencing separator field to each entity and code to initialize it using a unique reference provided by the simulation time kernel upon creation.

For uniformity, all entity field accesses are converted into method invocations. Then, all method invocations on entities are subsequently replaced with invocations to the simulation time kernel. This invocation requires the caller entity time, the method invoked, the target instance and the invocation parameters: the simulation time comes from the kernel; the method invoked is identified using an automatically created and pre-initialized method reflection stub; the target instance is identified using its separator, which is found on the stack in place of the regular Java object reference, along with the invocation parameters, which must be packed into an object array to conform with Java calling conventions. The rewriter injects all the necessary code to do this inline, and also deals with the natural complications of handling primitive types, the `this` keyword, constructor invocation restrictions, static initializers, and other Java-related details.

The rewriter then modifies all entity creations in all classes to place a separator on the stack in place of the object reference. All entity types in all entities are also converted to separators, namely in: field types, method parameter types and method return types, as well as typed instructions, including field accesses, array accesses and creation, and type casting instructions. Finally, all static calls to the `JistAPI` object are converted into equivalent implementations that invoke functionality of the simulation time kernel.

In addition to the entity-related program modifications, the rewriter also performs various static analyses that help drive runtime optimizations. For instance, the rewriter identifies open-world immutable objects [12, 52] and labels them as *timeless*, which means that they may be passed by reference across entities, not by copy. In some cases, the analysis can be overly conservative due to Java static type restrictions. The programmer can then explicitly define an object to be timeless, implying that the object will not be modified in the future even though it technically could be. The rewriter performs a number of similar analyses and generates code to assist in efficient object checkpointing and remote entity invocation as in the KaRMI system [54].

For ease-of-use, the JiST rewriter is implemented as a dynamic class loader. It uses the Byte-Code Engineering Library [20] to automatically modify the simulation program byte-code as it is loaded by the JiST bootstrapper into the Java virtual machine. Since the rewriting is performed only once, it could, if necessary, also be implemented as an offline process. Thus, rewriting speed is not a critical metric. Nevertheless, JiST loads, verifies and rewrites all of the SWANS classes in under 2 seconds. Even the addition of a rewriter cache does not reduce this time significantly, indicating that

JistAPI.java

```
1  package jist.runtime;
2  public class JistAPI {
3    public static interface Entity { }
4    public static interface Timeless { }
5    public static long getTime() {..}
6    public static void sleep(long n) {..}
7    public static void end() {..}
8    public static void endAt(long t) {..}
9    public static JistAPI.Entity THIS;
10   public static EntityRef ref(Entity e) {..}
11 }
```

Figure 5: 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.

the majority of the startup time is due to the Java class loader.

The rewriter processing naturally increases the size of the simulation class files. As shown in Figure 4, this overall increase is not considerable. The greatest relative increase is concentrated among entity classes, because they are usually just small wrappers and have accessor methods, stub fields, self-referencing separators and various runtime helper methods added to them during rewriting. Furthermore, the majority of the increase in all classes is concentrated in the constant pool. The rewriter, as currently implemented, simply adds constants that it requires and leaves any unused entries in the constant pool. While these could be removed to further reduce the class size, they do not affect performance and amount to only a few kilobytes of memory. Most importantly, the increase to the code segment of non-entity classes, which is the bulk of the simulation model, is less than 7%. The majority of this code is for packing invocation parameters and dealing with the Java primitive types.

3.2 Simulation time kernel

After rewriting, the simulation classes may be executed over a regular Java virtual machine. During their execution, these rewritten applications interact with the simulation time kernel, which supports the simulation time semantics.

The simulation time kernel serves a number of functions. The kernel is responsible for scheduling and transmitting time-stamped messages among the entities. It provides unique identifiers for each entity created in the system, which are used, for example, by the entity separator stubs during method invocation. The kernel maintains a time-stamp and a message queue structure for every entity in the system, and is thus able to respond to application `getTime` requests and time-stamp outgoing invocation messages. The kernel queues messages on behalf of each entity, and automatically advances the entities though simulation time, delivering messages for application processing as appropriate. And, finally, the kernel supports various system maintenance functions, such as entity garbage collection, load balancing and application monitoring.

Simulation processing begins via an anonymous bootstrap entity with a single scheduled message: to invoke the `main()` method of the given entry point class at time t_0. The system then processes events in simulation temporal order until there are no more events to process, or until a predetermined time is reached, whichever comes first. This general approach supports the sequential execution of any discrete event simulation. JiST may transparently exploit parallelism or process messages optimistically, as discussed shortly.

In general, the design of the JiST simulation time kernel is similar to that of the TimeWarp Operating System [33] kernel and that of the Parsec runtime, however it is considerably more lightweight and efficient. The language-based implementation allows efficient message delivery to local entities, without any serialization. Furthermore, since entities are merely objects rather than threads or processes, they utilize fewer system resources: JiST entities require less memory and neither require a stack nor encumber the system scheduler. Finally, JiST entities can transparently support efficient checkpointing and rollback using language-based serialization and reflection.

3.3 API and semantics

The result of the application modifications combined with the necessary runtime support is the introduction of simulation time execution semantics. These semantics are driven through a small API, which is exposed at the language level through various interfaces, methods and fields. The entire JiST simulation time interface is contained within the `JistAPI` class listed in Figure 5 and explained below:

- **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`.
- **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 transfered among entities. Immutable objects are implicitly tagged as timeless. e.g. `jist.swans.node.Message`.
- **getTime()**: returns the calling entity simulation time. The current time is the time of the current message being processed plus any additional `sleep` time.
- **sleep(long)**: advance calling entity simulation time.
- **end()**: end simulation after the current time-step.
- **THIS**: entity self-referencing separator, analogous to Java `this` object self-reference.
- **ref(Entity)**: returns a separator stub 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.

3.4 Hello world!

These basic primitives allow us to write simulation programs, including full wireless network simulators. The simplest such program, that still uses simulation time semantics, is a counterpart of the obligatory "hello world" program. It is a simulation with only a single entity that emits one message at every simulation time-step, as listed in Figure 6.

This simplest of simulations highlights some important points. To begin, the `hello` class is an entity, since it implements the `JistAPI.Entity` interface (line 2). Entities can be created (line 5) and their methods invoked (lines 6 and 10)

hello.java

```
1  import jist.runtime.JistAPI;
2  class hello implements JistAPI.Entity {
3   public static void main(String[] args) {
4    System.out.println("simulation start");
5    hello h = new hello();
6    h.myEvent();
7   }
8   public void myEvent() {
9    JistAPI.sleep(1);
10   myEvent();
11   System.out.println("hello world, t="
12     +JistAPI.getTime());
13  }
14 }
```

Figure 6: The simplest of simulations consists of a single entity that emits a message at each time step.

just as any regular Java object. The entity method invocation, however, happens in simulation time. This is most apparent on line 10, which is a seemingly infinite recursive call. In fact, if this program is run under a regular Java virtual machine (i.e. without the JiST rewriting machinery) then we would encounter a stack overflow at this point. However, under JiST, the semantics is to schedule the invocation via the simulation time kernel, and thus the call becomes non-blocking. Therefore, the `myEvent` method, when run under JiST semantics, will advance simulation time by one time step (line 9), then schedule a new event at that future time, and finally print a hello message (line 11) with the entity simulation time (line 12). As expected, the output is:

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

3.5 Parallel simulation execution

The system, as described thus far, is capable of executing simulations sequentially, and it does so with performance that either matches or exceeds existing, highly optimized simulation engines. JiST, however, was explicitly designed with concurrent, distributed and speculative execution in mind. By modifying the simulation time kernel, the same, unmodified simulations can be executed over a more powerful computing base. This section describes the mechanisms that allow for applications to be transparently executed in this manner.

Parallel execution in JiST is achieved by dividing the simulation time kernel into multiple threads of execution, called *controllers*. Each controller owns and processes the events of a subset of the entities in the system, as shown in Figure 7. Naturally, controllers synchronize with one another in order to bound their otherwise independent forward progress.

However, JiST automatically supports rollback, so the simulation time synchronization protocols among the various controllers need not be conservative. State checkpoints can be automatically taken through object cloning. Alternatively, efficient undo operators can be statically generated through code inspection in some cases, or possibly provided by the developer in other cases, for performance. In any event, entity state changes can always be dynamically intercepted and logged either at the entity, object or field level,

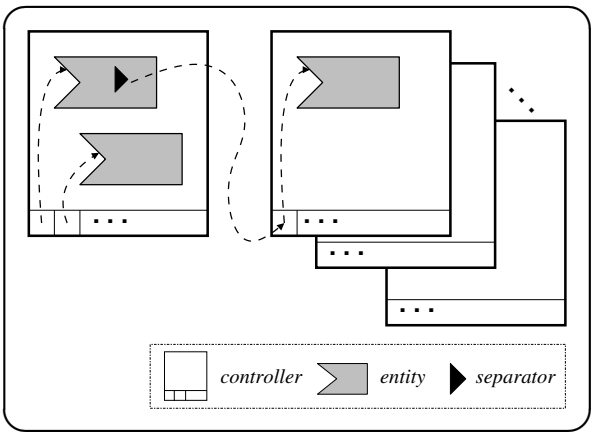

Figure 7: The JiST framework allows parallelism to be transparently introduced by partitioning the system entities among different, possibly distributed threads of control. Separators function to create a single system image abstraction among the distributed controllers.

allowing the JiST system to transparently perform speculative execution.

Controllers are distributed in order to run simulations across a cluster of machines. Conveniently, Java support for remote method invocation combined with automatic object serialization provides location transparency among the distributed controllers. Even beyond this, separators allow entities to dynamically be moved among controllers in the system, for balancing load or for minimizing invocation latency and network bandwidth. The automatic insertion of separators between entities at the static rewriting phase provides the simulation developer with a convenient single system image abstraction.

4. EVALUATION

Based on conventional wisdom [6], one would argue against implementing the JiST system in Java, an interpreted language, for performance reasons. In fact, the vast majority of existing simulation systems have been written in C and C++, or their derivatives. Our choice in favor of Java was made, despite this knowledge, for software engineering reasons. It was decided to sacrifice some sequential event-processing throughput so that we could:

- transparently run our simulations in parallel and optimistically, as discussed in Section 3.5, hoping to regain simulation performance and scalability through an analysis of time warp protocols in the context of wireless simulation; and
- gain prototyping speed from automatic garbage collection, type safety and other similar language benefits, as discussed in Section 5.2.

We were thus pleasantly surprised to discover that JiST actually out-performs the popular ns2 [45] and also the scalable GloMoSim [65] simulators, even on sequential benchmarks. We chose to investigate this matter further, and

discovered that 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 [8] runtime! In this section, we present 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 presented represents an average of at least two runs, with deviations of less then 1% in all cases. All tests were also performed on a second machine – a more powerful and memory rich dual-processor – giving identical absolute or relative performance results.

4.1 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 do nothing but schedule another identical event in 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.[1] 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`, 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 8. 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 outperforms 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.

[1]Simplified listings are included in the appendix.

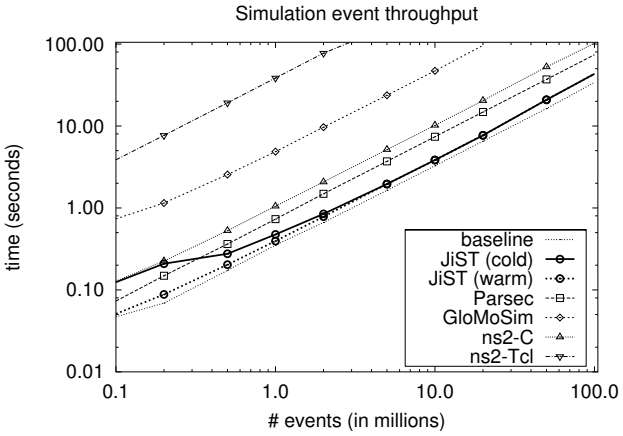

Figure 8: JiST has higher event throughput, and comes within 20% of the baseline lower bound. The kink in the JiST curve in the first fraction of a second of simulation is evidence of JIT compilation at work.

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 9: Time to perform 5 million events, and normalized against both the baseline and JiST performance.

The table in Figure 9 shows the time taken to perform 5 million events in each of the measured simulation systems, and also those figures normalized against both the baseline and JiST performance. JiST is twice as fast as both Parsec and ns2-C. GloMoSim and ns2-Tcl are one and two orders of magnitude slower, respectively.

4.2 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

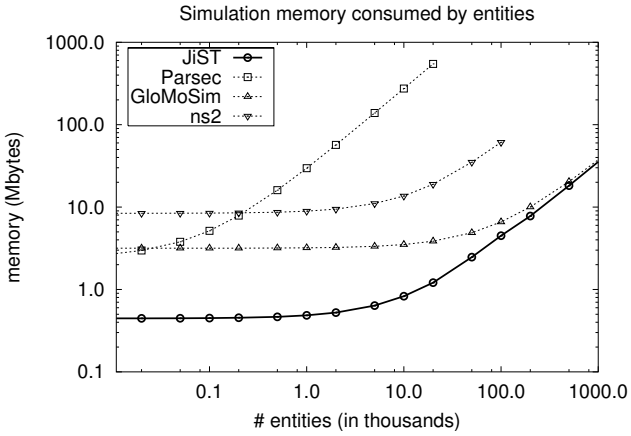

Figure 10: JiST allocates entities efficiently: comparable to GloMoSim at 36 bytes per entity, and over an order of magnitude less that Parsec or ns2.

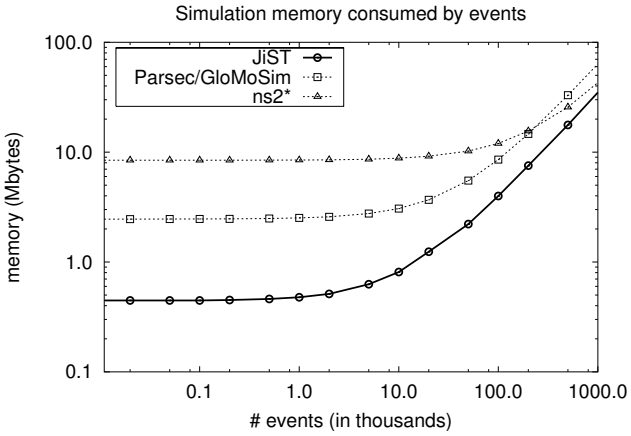

Figure 11: JiST allocates events efficiently: comparable to ns2 (in C) at 36 bytes per queued event, and half the size of events in Parsec and GloMoSim. (*) Events scheduled in ns2 via Tcl will allocate a split object and thereby incur the same memory overhead as above.

time scheduler. The overheads of both Parsec and JiST were empirically measured. They are both negligible, and do not represent a scalability constraint.

4.3 Memory utilization

Another important resource that limits scalability is memory. We thus 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 Figures 10 and 11, respectively. 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.

Figure 10 – 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[2]. 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.

Figure 11 – 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. On the other hand, Tcl-based ns2 events 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. Presumably, Parsec uses some additional space in the event structure for scheduling algorithm accounting.

[2]Minimum stack size allowed by Parsec is 20 KB.

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 12: 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.

The memory requirements per entity, mem_{entity}, and per event, mem_{event}, in each of the systems are tabulated in Figure 12. 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 [58, 48].

4.4 Parallelism

Thus far, we have shown that our technique for embedding simulation time produces an efficient sequential simulation engine. There are numerous remaining design options and trade-offs for the efficient implementation of a concurrent, distributed and optimistic simulation runtime, including areas such as entity cloning and serialization, remote invocation, checkpointing and undo, dynamic entity migration, load balancing, latency reduction, speculative execution bounds and message cancellation. We have not discussed these aspects of JiST, and therefore leave their evaluation outside the scope of this paper.

5. DISCUSSION

Having evaluated the computational and memory performance of JiST against ns2, GloMoSim and Parsec, we found that JiST either matches or out-performs all of these systems in both time and space. This section summarizes the important design decisions in each of the systems that bear most significantly on these performance results. We then discuss the various language and runtime alternatives that we considered, and also some uses for JiST as a research platform in both the simulation and applied simulation spaces.

5.1 Performance summary

Parsec runs very quickly, and there are a number of reasons for this. It is compiled and uses the `gcc` compiler to produce highly optimized binaries. It also uses non-preemptive logical processes to avoid system switching overhead[3]. The process-oriented model, however, exacts a very high memory cost per entity, since each entity must store a program counter and its stack.

GloMoSim remedies the per entity overhead by inserting an level of indirection in the message dispatch path and aggregating multiple nodes into a single entity. While this reduces the number of entities in the system, the indirection comes with a performance penalty.

ns2 is a sequential engine, so message queuing and dispatch is efficient. However, ns2 employs a split object model across C and Tcl to facilitate dynamic simulation configuration. This not only forces a specific coding pattern, it also comes at a performance cost of replicating data between the two memory spaces. More importantly, it exacts a high memory overhead. The ns2 code written in C still runs quickly; the Tcl-based functionality is almost two orders of magnitude slower.

JiST uses a concurrent object model of execution, and thus does not require node aggregation. Since entities are objects, as opposed to processes, the memory footprint is small. There is also no context switching overhead on a per event basis, and dynamic Java byte-code compilation and optimization result in high computational throughput. Since Java is a dynamic language, JiST does not require a split object model. JiST has a built-in scripting language that operates by reflection directly on the same objects used to run the simulation. This both eliminates the performance gap and the additional memory requirements.

[3]A Parsec context switch is implemented efficiently using only a `setjmp` and a stack switch.

5.2 Language alternatives

However, given that JiST is a Java-based system, it is natural to question this decision and to ask whether similar benefits could not be attained using other languages. Java has a number of advantages. It is a *standard*, widely deployed language, not specific to writing simulations. Consequently, the language boasts a large number of compilers as well as *portable*, *optimized* language runtimes. Java is *object-oriented*[4] and supports object reflection, serialization and cloning, which facilitates reasoning about the simulation state at runtime. The intermediate *byte-code* representation conveniently facilitates code instrumentation to support the simulation time semantics. *Type-safety* and *garbage collection* greatly simplify writing simulations by addressing common sources of error. Type-safety allows, for example, the event queue to be statically type-checked, rather than forcing programmers to strictly adhere to event constants when casting pointers to their appropriate function types and to ensure stack discipline. Garbage collection prevents memory leaks associated with simulation objects, such as network packets, that can have variable lifetimes and may traverse many different code paths in the system. This is particularly important for large, long-running simulations, where memory leaks are especially hazardous. These latter two properties also considerably shorten and simplify the simulation code.

Some of these properties exist in other simulation languages. JiST inherits them all from Java for "free". Also, these properties are not unique to Java. A system like JiST could likely be implemented atop a number of other languages. Based on our knowledge of existing languages and our experience implementing JiST, the most suitable candidates include Smalltalk, C#, Ruby and Python.

5.3 Runtime alternatives

Alongside the language decision there were various, related design alternatives for the system runtime. Among the more important design considerations, were the criteria discussed below and summarized in Figure 13. For the purposes of illustration, we compare JiST to the popular ns2 and scalable GloMoSim simulators, which run over Tcl+C and Parsec runtimes, respectively.

5.3.1 Compilation

Certain simulation systems, such as ns2 which relies on embedded Tcl, can be scripted. As noted by the ns2 designers, their *split object* model allows for convenient simulation composition and configuration. One can also dynamically insert runtime inspection points and triggers to facilitate tracing and debugging using reflection. On the other hand, GloMoSim is written exclusively in Parsec, a compiled simulation language, providing significant performance advantages as well as important static processing and checks.

JiST lies between these two points: it is compiled, and thus statically type-checked, but only to an intermediate byte-code representation. Since the byte-code is interpreted, one still retains the ability to dynamically perform simulation configuration, tracing, and debugging. Specifically, JiST supports Java-based scripting by integrating with the

[4]Except Java primitive types, which are not objects. This language design decision imposes special processing requirements and unfortunately introduces unavoidable overhead in some of the most performance-critical sections of JiST.

simulator	runtime	compilation	sharing	reuse
ns2/ PDNS	Tcl, C	split objects	shared everything	none
GloMoSim	Parsec	compiled	shared nothing	language
SWANS	JiST	unified objects, dynamic	shared nothing, aggregation	language and runtime

Figure 13: Summary of runtime design alternatives – SWANS/JiST is compared with GloMoSim/Parsec and ns2/Tcl+C along the criteria of compilation, sharing and reuse, as discussed in Section 5.3

BeanShell interpreter [51]. Other candidate Java-based interpreter engines include Jython (Python), Rhino (JavaScript) and Jacl (Tcl). The JiST machinery also allows actions to be executed before and after each event on a per-entity basis, which allows for orthogonal simulation inspection and logging, modifying the simulation state while it is in progress and general-purpose debugging. It is also useful for application-level functionality, such as efficiently implementing node mobility. Thus, JiST inherits the positive attributes of both the interpreted and the compiled. However, it does rely heavily on JIT optimizations for performance.

5.3.2 Sharing

When modeling the state of a simulation, it is often possible to partition the state into entities, which serve to restrict random state access. ns2 does not partition its simulation state; it is built in a *shared everything* style. The advantages are a simple programming model and fast local memory access. However, attempts to parallelize ns2, such as PDNS [58], are forced to construct distributed shared memory, which can result in non-uniform memory access as well as hidden serialization and synchronization costs.

GloMoSim is built over Parsec, which constructs a message passing or *shared nothing* system. In Parsec, each entity becomes an individually schedulable thread of control with independent state (a logical process), context-switched either at the system or user level. The advantage is that state is explicitly partitioned and distributed under programmer control. However, entity memory overheads and inter-entity communication costs can become prohibitive. To reduce this cost, GloMoSim implements *state aggregation*, both vertically and horizontally: the states of an entire simulated communication stack of components and the states of many physically adjacent simulated nodes are combined within a single Parsec entity. Though more efficient, this both breaks the clean entity separation that Parsec supports and, more importantly, inserts so much state into a single entity that it becomes inefficient to perform speculative execution, increasing synchronization costs.

JiST again lies between these two points: at the language level entities, as in Parsec, share no state and communicate only via messages, but at runtime all local entities co-exist within a single execution context in one virtual machine and efficiently pass messages as object references via the simulation time kernel. Furthermore, the aggregation is automatic, and does not require the extensive GloMoSim machinery; it can occur non-uniformly and dynamically at entity granularity, and without any programmer-defined vertical or horizontal partitioning schemes. Finally, the JiST model transparently permits the use of specialized hardware capable of supporting multiple processors and non-uniform memory access (NUMA) architectures by using Java's threading model over suitable virtual machines.

5.3.3 Reuse

The entire ns2 system is encoded in Tcl and C. However, these languages are not *reused*, in the sense that they are independent of the simulator implemented. In contrast, the GloMoSim system is written in Parsec, a language that is based on C. The Parsec compiler is actually implemented as a front-end to `gcc`. A notable advantage [65] of this approach is that various protocol implementations in C can be ported to Parsec with relative ease. While this may be of little use for prototype implementations, the reuse of large portions of the `gcc` compiler also has some other advantages. It permits the specialized Parsec compiler and the Parsec community to benefit from sophisticated optimizations implemented in the widely used `gcc` engine. It also provides other software engineering benefits such as portability and credibility.

The JiST design goes one step further by reusing both the language and its runtime. First, JiST embeds the simulation time primitives within plain Java in order to reuse the standard Java language and existing compilers. The primitives were designed to be preserved through the compilation process, thus allowing the rewriter to operate at the byte-code level without source-code access. The transformation also preserves various important Java properties such as type safety. Secondly, JiST reuses the Java runtime. Since the rewriter output is a set of class files and the simulation time kernel is written entirely in Java, JiST can operate over existing, highly-optimized virtual machines. Moreover, this homogeneity actually permits important JIT optimizations to occur across the application-kernel boundary, as shown by the Jalapeño project [2]. Finally, the runtime provides important functionality transparently, including reflection and serialization.

5.4 Research platform

One of the design goals of the JiST system is to create a platform upon which current PDES research may be transparently exposed to the broader simulation community. In fact, one of our goals is to apply JiST in the domain of wireless network simulation. We also contend that the JiST system is a convenient platform for the exploration of new simulation techniques.

5.4.1 SWANS - Wireless network simulation

Both as a proof of the JiST concept and as a research tool, we are building SWANS, a **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator. The SWANS software is organized as independent software components that can be composed to form complete wireless simulations, as shown in Figure 14. 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.

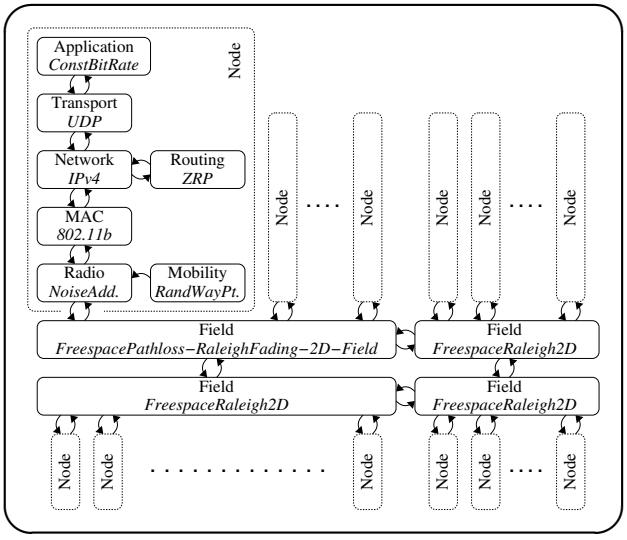

Figure 14: The SWANS simulator consists of event-driven components that can be configured and composed to form a meaningful wireless network simulation. Different classes of components are shown in a typical arrangement together with specific instances of component implementations in italics.

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. 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. Finally, it also restricts the simulation communication pattern. For example, application or routing 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 contrast, different *Radio*s do contend (in simulation time) over the shared *Field* entity, and raise the synchronization cost of a concurrent simulation execution. To reduce this contention, as observed by the JiST simulation time kernel, the simulated field is partitioned into non-overlapping, cooperating *Field* entities along a grid.

It is important to note that communication among entities is very efficient. The JiST 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 their reference. The messages themselves are a chain of nested objects that mimic the data encapsulation of the network stack. An additional convenience of this approach is that it allows for the use polymorphism in message processing.

Notably, the development of SWANS has been relatively painless. Since JiST inter-entity message creation and delivery is implicit, as well as message garbage collection and typing, the code is very compact and intuitive. Components in JiST consume less than half of the code (in uncommented line counts) of comparable components in GloMoSim, which are already smaller than their implementations in ns2.

The SWANS project is still underway; the implementation of more protocols and components is in progress. Currently, SWANS supports *all* of the functionality of GloMoSim below the MAC layer, and also selected protocols higher up the stack. This progress validates JiST from a software engineering perspective.

5.4.2 Simulation research

In addition to being a practical and efficient simulation tool for building simulators, we believe that the JiST abstraction can provide a base for ongoing simulation research. Primarily, this is due to the flexibility afforded by the intermediate Java byte-code representation of simulations.

We have described, for example, how to convert *simulation time* programs at the byte-code level to run in parallel using either conservative synchronization algorithms [17, 18, 34] or speculative global virtual time [32, 25] algorithms. The byte-code is also used to introduce a language-based single system image abstraction that permits simulations to be transparently distributed across a cluster of machines and to be dynamically load balanced.

Other, more recent, research candidates exist. Static code analysis techniques can be used to produce efficient entity checkpointing as in [28]. Alternatively, reverse computations can be produced for entity rollback as in [16]. The JiST design can simultaneously support different synchronization protocols for different entities by tagging them with different interfaces as in [43]. Finally, a continuation-passing style transformation can be applied to process-oriented simulations in order to eliminate the need for an entity stack as in [13]. Research into each of these ideas would benefit from the availability of pre-existing simulations.

6. RELATED WORK

The JiST work spans three domains of research. The following sections describe the related research focused on simulation technology, on network simulation tools and on improving the Java platform.

6.1 Simulation languages, libraries, systems

Simulation research has a rich history dating back to the late 60s, when it prompted the development of Simula [19]. Many other simulation languages, libraries and systems have since been designed, focusing on performance, concurrency and distribution, speculative execution, and new simulation domains.

One approach has been to create new simulation languages that are closely related to popular existing languages, with extensions for message dispatch, synchronization and other simulation time primitives. Csim [59], Yaddes (Parsimony) [56], Maisie [7], and Parsec [8], for example, are derivatives of C and C++, and include support for a process-oriented simulation execution model. Others, such as Apostle [14] and TeD [53] have taken a more domain-specific language approach. Some work in the early 90s also focused on object-oriented simulation languages. These projects, including Moose [64], Sim++ [5], Pool [3], ModSim II [15], Act++ [37] and Rosette [63] investigated various object-oriented possibilities for checkpointing, inheritance, concur-

rency and synchronization in the context of simulation, including a interesting mixture of the process oriented and concurrent object execution models.

Another approach has been to create simulation libraries that support parallel and speculative simulation execution, including projects such as OLPS [1], Speedes [60], Yansl [36], SimKit [27] and Compose [43]. A primary benefit of this approach is that these libraries are usable within existing languages.

Finally, researchers have investigated simulation "operating" systems, that support simulations as a set of communicating processes. The system kernel then schedules message delivery and can transparently checkpoint process state. For performance, these are often implemented as logical processes. The landmark paper in this space is the TimeWarp OS [33]. Projects such as GTW [21], Warped [42], Parasol [44], DaSSF and others, including some of the language projects mentioned above, have investigated important dynamic optimizations within this model.

6.2 Wireless network simulators

The networking community depends heavily on simulation to validate its research. The ns2 [45] network simulator has had a long history with the community and is widely trusted. It was therefore extended to support mobility and some wireless protocols [35]. Though it is primarily used sequentially in the community, researchers have extended ns2 to PDNS [58], allowing for conservative parallel execution.

GloMoSim [65] is a newer simulator written in Parsec [8] that has recently gained popularity within the wireless ad hoc networking community. The sequential version of GloMoSim is freely available. The conservatively parallel version has been commercialized as QualNet. Another notable, commercially-supported network simulator is OPNet. Recently, the Dartmouth Scalable Simulation Framework (DaSSF) has also been extended to support wireless ad hoc network simulations [40]. Some of the simulation language projects mentioned above, such as TeD and Speedes, have also written network simulation software.

6.3 Java-related

Java, because of its popularity and favorable properties, has become the focus of much recent research. The runtime has not only undergone extensive performance work, it has also become the compilation target for many other languages [62]. Projects such as JKernel [30] have investigated the advantages of bringing traditional systems ideas of process isolation and resource accounting into the context of a safe language runtime. Others, including cJVM [4] and Jessica [41], have used Java byte-code rewriting techniques to provide an abstraction of a single-system image abstraction over a cluster of machines. The MagnetOS project [9] has extended this idea to support transparent code migration in the context of an ad hoc network operating system [10]. The JavaParty [55], and xDU [26] projects have looked at mechanisms to facilitate Java application partitioning and distribution.

There has also been some research interest in using Java for simulation [38]. Silk [31] is a new process-oriented simulation language that has a Java-based compiler and targets the Java runtime. IDES [50] and Ptolemy [11] are a Java-based simulation libraries. Finally, DaSSF includes hooks to allow for Java-based event-handlers. To the best of our knowledge, JiST is the first system to integrate simulation execution semantics directly into the Java execution model.

7. CONCLUSION

In this paper, we have introduced JiST, a new Java-based simulation framework that executes discrete event simulations both efficiently and transparently, and does so by embedding simulation time semantics directly into the Java execution model. We outlined our rationale behind designing the system to function within the standard Java language and runtime, and contrasted it with other approaches that require either specialized languages, custom libraries or new runtimes. We then describe the details of the JiST implementation, including the rewriter, simulation time kernel and simulation programming interface, and continue with an explanation of how this transparently allows for concurrent and optimistic simulation execution. We present an evaluation of JiST, showing that it performs surprisingly well, either matching or exceeding the serial performance of existing wireless simulators in both time and memory consumption. Finally, we discuss the construction of the SWANS prototype, a wireless ad hoc network simulator, and the applicability of JiST for ongoing simulation research. Our future plans include development of more components for the SWANS simulator and conducting an analysis of optimistic execution in the context of wireless network simulation. We hope that the performance of JiST, combined with its pleasing software engineering attributes and the popularity of the Java language, will facilitate the broad adoption of parallel simulation research.

8. REFERENCES

[1] M. Abrams. Object library for parallel simulation (OLPS). In *Proceedings of the Winter Simulation Conference*, pages 210–219, Dec. 1988.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, A. Cocchi, S. F. Hummel, D. Lieber, T. Ngo, M. F. Mergen, J. C. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 314–324, Nov. 1999.

[3] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 161–168, Oct. 1990.

[4] Y. Aridor, M. Factor, and A. Teperman. cJVM: A single system image of a JVM on a cluster. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, Sept. 1999.

[5] D. Baezner, G. Lomow, and B. W. Unger. Sim++: The transition to distributed simulation. In *Proceedings of 1990 SCS Multiconference on Distributed Simulation*, pages 211–218, Jan. 1990.

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

[7] R. L. Bagrodia and W.-T. Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, Apr. 1994.

[8] 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.

[9] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *ACM SIGOPS Operating Systems Review*, 36(2):1–5, Apr. 2002.

[10] R. Barr, T. D. Kim, I. Y. Y. Fung, and E. G. Sirer. Automatic code placement alternatives for ad hoc and sensor networks. Technical Report 2001-1853, Cornell University, Computer Science, Nov. 2001.

[11] S. Bhattacharyya, E. Cheong, J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie, N. Smyth, J. Tsay, B. Vogel, W. Williams, Y. Xiong, and H. Zheng. Heterogeneous concurrent modeling and design in Java. Technical Report UCB/ERL M02/23, UC Berkeley, EECS, Aug. 2002.

[12] M. Biberstein, J. Gil, and S. Porat. Sealing, encapsulation, and mutability. In *Proceedings of 15th European Conference on Object-Oriented Programming*, pages 28–52, June 2001.

[13] C. J. M. Booth and D. I. Bruce. Stack-free process-oriented simulation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 182–185, June 1997.

[14] D. Bruce. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation. In *Proceedings of the Workshop on Domain-specific Languages*, Jan. 1997.

[15] O. Bryan, Jr. Modsim II - an object-oriented simulation language for sequential and parallel processors. In *Proceedings of the Winter Simulation Conference*, pages 122–127, Dec. 1989.

[16] C. D. Carothers, K. S. Perumalla, and R. M. Fujimoto. Efficient optimistic parallel simulations using reverse computation. In *Proceedings of the Workshop on Parallel and Distributed Simulation*, pages 126–135, May 1999.

[17] K. M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5:440–452, 1979.

[18] K. M. Chandy and R. Sherman. The conditional event approach to distributed simulation. In *Proceedings of the Distributed Simulation Conference*, 1989.

[19] O.-J. Dahl and K. Nygaard. Simula, an Algol-based simulation language. *Communication of the ACM*, pages 671–678, 1966.

[20] M. Dahm. Byte code engineering with the BCEL API. Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, Apr. 2001.

[21] S. R. Das, R. M. Fujimoto, K. S. Panesar, D. Allison, and M. Hybinette. GTW: A time warp system for shared memory multiprocessors. In *Proceedings of the Winter Simulation Conference*, pages 1332–1339, Dec. 1994.

[22] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[23] R. M. Fujimoto. Parallel discrete event simulation: Will the field survive? *ORSA Journal on Computing*, 5(3):213–230, Summer 1993.

[24] R. M. Fujimoto. Parallel and distributed simulation. In *Proceedings of the Winter Simulation Conference*, pages 118–125, Dec. 1995.

[25] R. M. Fujimoto and M. Hybinette. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modelling and Computer Simulation*, 7(4):425–446, Aug. 1997.

[26] S. Gehani and G. Benson. xDU: A Java-based framework for distributed programming and application interoperability. In *Proceedings of the Parallel and Distributed Computing and Systems Conference*, 2000.

[27] F. Gomes, J. Cleary, A. Covington, S. Franks, B. Unger, and Z. Ziao. SimKit: a high performance logical process simulation class library in C++. In *Proceedings of the 27th Winter Simulation Conference*, pages 706–713, Dec. 1995.

[28] F. Gomes, B. Unger, and J. Cleary. Language-based state-saving extensions for optimistic parallel simulation. In *Proceedings of the Winter Simulation Conference*, pages 794–800, Dec. 1996.

[29] W. Gropp and E. Lusk. The MPI communication library: its design and a portable implementation. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 160–165, Oct. 1993.

[30] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proceedings of the USENIX Annual Technical Conference*, pages 259–270, June 1998.

[31] K. J. Healy and R. A. Kilgore. Silk : A Java-based process simulation language. In *Proceedings of Winter Simulation Conference*, pages 475–482, Dec. 1997.

[32] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[33] 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.

[34] V. Jha and R. L. Bagrodia. Transparent implementation of conservative algorithms in parallel simulation languages. In *Proceedings of the Winter Simulation Conference*, Dec. 1993.

[35] D. B. Johnson. Validation of wireless and mobile network models and simulation. In *Proceedings of the DARPA/NIST Workshop on Validation of Large-Scale Network Models and Simulation*, May 1999.

[36] J. A. Joines and S. D. Roberts. Design of object-oriented simulations in C++. In *Proceedings of the Winter Simulation Conference*, pages 157–165, Dec. 1994.

[37] D. G. Kafura and K. H. Lee. Inheritance in Actor-based concurrent object-oriented languages. *IEEE Computer*, 32(4):297–304, 1989.

[38] R. A. Kilgore, K. J. Healy, and G. B. Kleindorfer. The future of Java-based simulation. In *Proceedings of the Winter Simulation Conference*, pages 1707–1712, Dec. 1998.

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

[40] J. Liu, L. F. Perrone, D. M. Nicol, M. Liljenstam, C. Elliott, and D. Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Proceedings of the European Simulation Interoperability Workshop*, 2001.

[41] M. J. M. Ma, C.-L. Wang, F. C. M. Lau, and Z. Xu. JESSICA: Java-enabled single system image computing architecture. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 2781–2787, June 1999.

[42] D. E. Martin, T. J. McBrayer, and P. A. Wilsey. Warped: A time warp simulation kernel for analysis and application development. In *Proceedings of Hawaii International Conference on System Sciences*, pages 383–386, Jan. 1996.

[43] J. M. Martin and R. L. Bagrodia. Compose: An object-oriented environment for parallel discrete-event simulations. In *Proceedings of the Winter Simulation Conference*, pages 763–767, Dec. 1995.

[44] E. Mascarenhas, F. Knop, and V. Rego. Parasol: A multithreaded system for parallel simulation based on mobile threads. In *Winter Simulation Conference*, pages 690–697, Dec. 1995.

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

[46] J. Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, Mar. 1986.

[47] D. M. Nicol. Parallel discrete event simulation: So who cares? In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, June 1997.

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

[49] D. M. Nicol and R. M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, pages 249–285, Dec. 1994.

[50] D. M. Nicol, M. M. Johnson, and A. S. Yoshimura. The IDES framework: a case study in development of a parallel discrete-event simulation system. In *Proceedings of the Winter Simulation Conference*, pages 93–99, Dec. 1997.

[51] P. Niemeyer. BeanShell: lightweight scripting for Java, at `http://www.beanshell.org/`, 1997.

[52] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. In *Proceedings of Java Grande*, Nov. 2002.

[53] K. S. Perumalla, R. M. Fujimoto, and A. Ogielski. TeD - a language for modeling telecommunication networks. *SIGMETRICS Performance Evaluation Review*, 25(4):4–11, 1998.

[54] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, 2000.

[55] M. Philippsen and M. Zenger. JavaParty — Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, 1997.

[56] B. R. Preiss. The Yaddes distributed discrete event simulation specification language and execution environment. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 139–144, 1989.

[57] G. Riley and M. Ammar. Simulating large networks: How big is big enough? In *Proceedings of the First International Conference on Grand Challenges for Modeling and Simulation*, Jan. 2002.

[58] 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.

[59] H. Schwetman. Csim18 - the simulation engine. In *Proceedings of the 28th Winter Simulation Conference*, pages 517–521, Dec. 1996.

[60] J. S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Proceedings of the SCS Multiconference on Advances in Parallel and Distributed Simulation*, pages 95–101, Jan. 1991.

[61] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, Dec. 1990.

[62] R. Tolksdorf. Programming languages for the Java virtual machine at `http://www.robert-tolksdorf.de/vmlanguages`, 1996-.

[63] C. Tomlinson and V. Singh. Inheritance and synchronization in enabled-sets. In *Proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 103–112, Oct. 1989.

[64] J. Waldorf and R. L. Bagrodia. MOOSE: A concurrent object-oriented language for simulation. *International Journal of Computer Simulation*, 4(2):235–257, 1994.

[65] 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.

APPENDIX

We have included simplified versions of the benchmark programs used to measure the event throughput of the JiST, Parsec, GloMoSim, and ns2 systems in Section 4.1. To the best of our knowledge, these are the fastest possible implementations. These listings do not include command-line parsing, integration code scattered in various common files (in case of GloMoSim and ns2), error handling, or the code for simulation timing. However, they closely reflect the style of code that a simulation developer would need to generate.

A. JiST

Jist.java

```
import jist.runtime.JistAPI;
class Jist implements JistAPI.Entity {
 public static void main(String args[]) {
  JistAPI.endAt(1000000);
  (new Jist()).event();
 }
 public void event() {
  JistAPI.sleep(1);
  event();
 }
}
```

B. Parsec

parsec.pc

```
message null { };
entity driver(int argc, char **argv) {
 int i;
 for(i=0; i<1000000; i++) {
  send null { }
   to self
   after 1;
  receive(null p) { }
 }
}
```

C. GloMoSim

glomo.h

```
#define MODE_NULL 0
typedef struct {
 int n; // number of events processed
 int size; // total number of events
} app_t;
void benchInit(GlomoNode *nodePtr);
void benchFinalize(GlomoNode *nodePtr, app_t *clientPtr);
void benchProcess(GlomoNode *nodePtr, Message *msg);
```

glomo.pc

```
#include "api.h"
#include "message.h"
#include "application.h"
#include "glomo.h"

static app_t *allocApp(GlomoNode *nodePtr) {
 // allocate application object in node structure
}
static app_t *getApp(GlomoNode *nodePtr) {
 // retrieve application object from node structure
}
void benchInit(GlomoNode *nodePtr) {
 app_t *clientPtr = allocApp(nodePtr);
 clientPtr->n = 0;
 clientPtr->size = 1000000;
 clientPtr->type = MODE_NULL;
 Message *timerMsg = GLOMO_MsgAlloc(nodePtr, GLOMO_APP_LAYER,
    APP_JISTBENCH, MSG_APP_TimerExpired);
 GLOMO_MsgSend(nodePtr, timerMsg, 0);
}
void benchFinalize(GlomoNode *nodePtr, app_t *clientPtr) {
 // finalization code
}
void benchProcess(GlomoNode *nodePtr, Message *msg) {
 switch(msg->eventType) {
  case MSG_APP_TimerExpired: {
   app_t *clientPtr = getApp(nodePtr);
   clientPtr->n++;
   if(clientPtr->n < clientPtr->size) {
    Message *timerMsg;
    timerMsg = GLOMO_MsgAlloc(nodePtr, GLOMO_APP_LAYER,
      APP_JISTBENCH, MSG_APP_TimerExpired);
    switch(clientPtr->type) {
     case MODE_NULL:
      GLOMO_MsgSend(nodePtr, timerMsg, 1);
      break;
     default: // error
    }
   }
   break;
  }
  default: // error
 }
 if(msg->info) free(msg->info);
 GLOMO_MsgFree(nodePtr, msg);
}
```

D. ns2-C

ns2.h

```
#include <tclcl.h>
#include "object.h"

class JistBenchEvents : public NsObject {
 protected:
  double events_;
 public:
  JistBenchEvents();
  void handle(Event* e);
  void schedulefirst();
  double events() { return events_; }
 protected:
  int command(int argc, const char*const* argv);
};
```

ns2.cc

```
#include "jist.h"
#include "scheduler.h"
#include "ns2.h"

static class JistBenchEventsClass : public TclClass {
 public:
  JistBenchEventsClass() : TclClass("JistBenchEvents") { }
  TclObject* create(int argc, char** argv) {
   return (new JistBenchEvents);
  }
} class_jist_bench_events;

JistBenchEvents::JistBenchEvents() {
 bind("events_", &events_);
}
int JistBenchEvents::command(int argc, char** argv) {
 if (argc==2) {
  if(strcmp(argv[1], "schedulefirst")==0) {
   schedulefirst();
   return TCL_OK;
  }
 }
 return TclObject::command(argc, argv);
}
void JistBenchEvents::schedulefirst() {
 Scheduler& s = Scheduler::instance();
 Event *ev = new Event;
 s.schedule(this, ev, 0);
}
void JistBenchEvents::handle(Event* ev) {
 delete ev;
 if(events_) {
  Scheduler& s = Scheduler::instance();
  ev = new Event;
  s.schedule(this, ev, 1);
  events_--;
 }
}
```

ns2.tcl

```
JistBenchEvents set events_ 0
JistBenchEvents set debug_ 0
set s [new Simulator]
set foo [new JistBenchEvents]
$foo set events_ 1000000
$foo schedulefirst
$s run
```