*Research Project Proposal*

# JIST – Java In Simulation Time

Rimon Barr, Cornell University
⟨barr@cs.cornell.edu⟩

*December 9, 2002*

**Abstract**

*Work in progress:* JIST is a Java-based simulation framework that can transparently modify a given discrete event simulation written in plain Java and execute it in parallel and optimistically.

## Introduction

Discrete-event simulators are event-driven programs that process time-stamped events in their temporal order, where event-processing code can both modify the simulation state and potentially generate future events. These programs are important scientific tools and are the focus of a vast body of computer science research that is directed at their efficient design and execution. From a systems perspective, researchers have considered many forms of parallel execution of such simulations spanning the gamut from conservative to aggressively optimistic concurrent execution, from shared-everything (shared memory) to shared-nothing (message passing) paradigms. From a modeling perspective, researchers have designed domain-specific languages that codify different forms of causality constraints and permit various static and dynamic optimizations. And, from a purely scientific perspective, research has grown increasingly dependent on and demanding of simulation.

The goal of the JIST project, which stands for **J**ava **I**n **S**imulation **T**ime, is to create the first simulation system that can execute discrete event simulations both *efficiently* and *transparently*. Efficiently denotes the ability to execute a given simulation implementation in parallel and optimistically, and that the system dynamically optimizes the simulation runtime configuration to improve throughput. Transparently refers to the automatic transformation of compiled simulation programs written in *plain* Java to run in "simulation time", allow for the necessary concurrency protocols and permit dynamic reconfiguration and inspection; all, while retaining the original simulation semantics. The term plain highlights an important distinction between JIST and previous simulation systems, in that the simulation code that runs on JIST need not be written in a task-specific language invented for simulation, nor need it be littered with special-purpose system calls and callbacks. Finally, the JIST system will be validated with the construction of SWANS, a prototype **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator. The proposed design of JIST is outlined below, followed by a discussion of the prior work and the project objectives.

## Motivation

The primary motivation for the JIST project originates from the current state of simulation in the ad hoc wireless networking community. Research activity relating to wireless devices, wireless networking and wireless applications has increased dramatically of late [survey papers]. Wireless device hardware has improved in processing capacity [ ], power efficiency [ ] and functionality [ ]. Wireless networking research has produced many new link [802.11, Bluetooth], transport [AODV, DSR, ZRP, (from a-exam and other papers)] and application level protocols [gossip work, ATP] suited to varied wireless conditions. And, finally, researchers have looked at new kinds of applications suited to wireless networks [7DS, Directed diffusion, UNSEEN, Cougar, Distributed and peer-to-peer query processing].

The proper measurement and validation of these projects, particularly scalable wireless projects (i.e. targeted at thousands of devices and up), has proven to be difficult for a number of reasons. Primarily, this difficulty stems from the fact that it is expensive to recruit the thousands of wireless devices. Such a large number of devices is also difficult to practically manage. Simply moving these devices in repeatable patterns and isolating experiments from surrounding interference is all but impossible. Charging devices and configuring software on potentially heterogeneous hardware platforms is also challenging.

A consequence of this practical difficulty is that the majority of research published in this area is evaluated in simulation. Some research groups have chosen to write custom simulators [ ]. Others have used existing simulators, such as ns2 [ ] with Monarch wireless extensions [ ], Glomosim [ ], or OPNet [ ], usually with some degree of modification and customization.

Moreover, the research ideas themselves, which are claimed to scale to at least thousands of nodes, have not been even simulated with either sufficient scalability or detail in published results. Published physical or MAC-layer simulations are only in the order of hundreds of nodes. Published packet-level simulations have reached thousands of nodes, by sacrificing some or all of the detail at the link and physical communication layers.

The JIST project aims to remedy this situation by providing a scalable simulator for the evaluation of large-scale wireless network protocols and applications. The kinds of questions that this research hopes to answer center around how to model a simulation of an ad hoc network in an manner that is efficient and utilizes the inherent structure of this problem. In addition, we hope to improve the state-of-the-art in general-purpose simulation by investigating how to build such simulations in standard systems languages (as opposed to domain-specific simulation languages) and transparently transform them into equivalent programs that run in parallel and optimistically.

## Proposed Design

As introduced and defined above, the purpose of the JIST system is to run discrete event simulations written in plain Java both efficiently and transparently. This section further elaborates on what that means and on the operational details of the JIST system. The entire process begins at the time when a compiled simulation is loaded into the JIST system. The application byte-code is verified, inspected and modified to run in simulation time. In the following section, the semantics of simulation time are described and how the transformation and a serial execution of the resulting application are to be performed. Following that, the parallel execution of simulations is explained, along with the associated optimisation problem of maximizing simulation event throughput through effective concurrent utilization of available CPU and network resources. Next, the optimistic execution of simulations is described, with a discussion regarding the need for balancing parallel forward progress in the system. Lastly, a brief design of the SWANS prototype is provided, followed by some concluding remarks regarding the design of the system.

## Running in Simulation Time

Regular Java Virtual Machines run their applications in *actual time*, and sometimes in *real time*. In both cases, the passing of time is independent of the application. The system clock advances while the program executes instructions, but time passes regardless of how many instructions are executed. In a real-time execution, the system attempts to guarantee that instructions or sets of instructions will meet given time deadlines. In essence, the rate of application progress is made dependent on the progress of time.

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

Individual primitive application byte-code instructions are processed sequentially, following the standard Java control flow semantics, but the system clock remains unchanged. Application code can advance its simulation time only via the `sleep(n)` system call[1]. In essence, every instruction takes zero time to process except `sleep`, which takes exactly $n$ time quanta of processing time. The JIST runtime processes its application in its simulation-time temporal order until the application ends or until a pre-determined simulation-ending time is reached, whichever comes first.

It is important to clarify that JIST is not intended to simulate the running of arbitrary Java programs. Rather, it is a Java-based simulation framework that can transparently and efficiently execute Java simulations programs. The structure of such an application is now described.

The simulation application consists of objects, and the state of the simulation is the combined state stored within each of those objects. At a coarser level of granularity, JIST also partitions the simulation state into *entities*. Entities are themselves objects. They are object that are either tagged in the source code by implementing the `Entity` interface, or defined as entities in an simulation description file. The state of an entity is defined to be the set of objects and primitive fields, if any, which it recursively contains. An important system restriction on the state of the application is that a regular object may not be referenced by more than one entity[2]. Entities perform a number of important functions within JIST.

Entities, as just defined, partition the state of a simulation application. They also partition the "application time". Namely, since entities do not share any application state, different entities may be at different points in simulation time during the execution of the simulation. There are natural

[1]Additional exceptions arise due to concurrency, synchronization and other blocking operations, but are outside the purpose and scope of this document.

[2]This application property is impossible to statically verify, in generality. Moreover, it can be trivially enforced over any application using pervasive copying at runtime. However, this is both expensive and may change the semantics of an invalid application. It is therefore simply assumed to be a property of the simulation and no copying is performed.

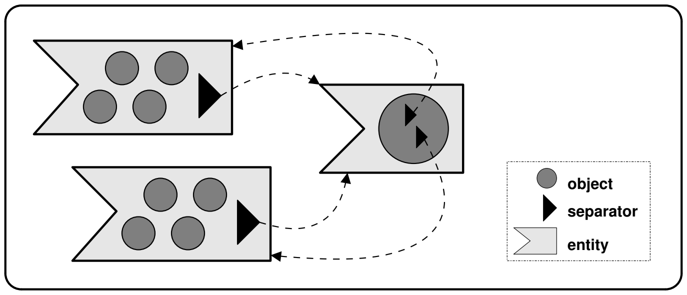

Figure 1: Runtime objects

restrictions on the separation of application time. Specifically, if one entity is to communicate with another, and possibly transfer some application state, then the two must be synchronized with respect to their simulation time.

Entities communicate via method invocations, just like regular objects in the system. There are a number of possible method invocation combinations to consider. When one object invokes another object within the same entity, including a recursive invocation, there is no concern, since both the caller and callee will be at the same simulation time. Another case is that of an object in one entity invoking an object within another entity. This case is just not possible, due to the above-mentioned restriction that no object may be a member of more than one entity. Lastly, there is the interesting case when some object in the system (possibly an entity object) invokes another entity, since the caller and callee may be at different points in simulation time. JIST solves this problem by introducing intermediaries know as *separators*.

Separators are special objects that interact closely with the JIST runtime. They do not exist in the original program. However, they are automatically generated and inserted into the application binary as part of the application modification process that occurs when it is first loaded. A separator is created for every entity, mimicking its public interface[3], and every entity reference in the program is replaced with a reference to its corresponding separator. The result is that every call to an entity instance will be intercepted by a separator instance. Rather than invoking the entity immediately, the separator will queue the invocation until the appropriate simulation time is reached. Note that the callee simulation time will always be equal to or precede the caller simulation time, since an entity can only move forward in time by calling `sleep`, and invocations of all entities in the system are always processed in simulation time order. More importantly, this design allows the state of two entities of the simulation to safely co-exist at two different points of simulation time.

Figure 1 shows objects, entities and separators of an simulation application running on JIST. Notice that the entire state of the application is partitioned into entities, which may contain objects, and that all references to entities are replaced with separators. The separators not only partition the application state, but also its time. Figure 2 includes the system event queue, which holds application callbacks until the appropriate simulation time for invocation. When one entity invokes a method on another entity, the invocation is intercepted and queued. In this manner, the simulation

[3]In reality, the process is more complex, involving some generated interfaces and other objects that are required to handle static state and other Java peculiarities, but these details are beyond the scope of this document.

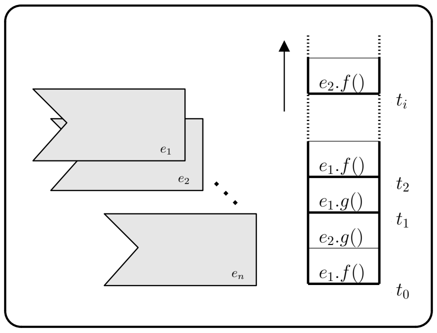

Figure 2: Simulation time execution

is executed in simulation time.

A novelty of the transformation just described is that discrete events may be expressed as regular method invocations among entities in plain Java. That is, a method invocation of an entity represents an event that will occur at the simulation time of the caller and is registered in the event queue for later invocation. However, since the separator method implementation merely registers the event, and does not invoke the real entity method nor does it wait for the invocation to occur, it can not return the actual invocation result. Thus, the signatures of entity methods are expected to return `void`, and this property is statically verified by JIST. This restriction ensures that it is not possible to transfer state into the simulation past and create an irreconcilable cyclical dependency in simulation time. However, since most discrete event simulators are already written in this message-passing form, it does not represent a significant restriction[4].

JIST begins the execution of a simulation by scheduling the `main()` method of one of its entities at time $t_0$. The system then processes events in simulation-time temporal order until there are no more events to process, or a pre-determined simulation time is reached, whichever comes first. This general approach will support the execution of any discrete event simulation. To execute the simulation more efficiently, JIST will exploit parallelism.

## Parallel Execution of Simulation Time

The previous section described how simulations that are written in plain Java are modified and executed. Implicitly, a single-threaded model was assumed, where events are processed strictly sequentially. However, if there are two or more events scheduled at the same simulation time (and this is often the case), it is possible to execute them concurrently.

This can be achieved with multiple, properly synchronized threads within a single JIST instance. The benefits include low context switching costs and a single shared memory space for the entire application state. This approach can maximize the throughput of a single host, however it will provide little speedup beyond the number of processors on that single host.

Further parallelism is possible by starting cooperating instances of JIST on different, networked hosts. These hosts operate in lock-step with respect to simulation time, and process all events of

[4]Also, it may be possible to efficiently and automatically transform an unrestricted application into continuation-passing style, though this is outside the scope of the research.

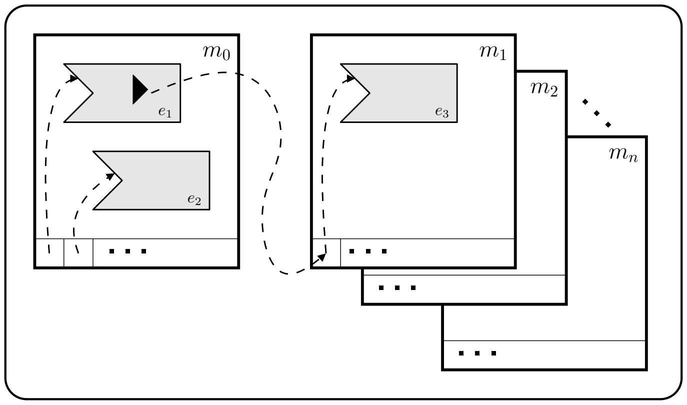

Figure 3: Location-independence of entities

that simulation time in parallel, with each host processing those events which are associated with locally residing entities. However, the degree of parallelism is heavily dependent on the event stream generated by the application. It depends both on the number of concurrent application events and the degree to which those events may be distributed. JIST dynamically migrates entities among servers in an effort to distribute and balance the computational load.

Providing transparent location independence for entities, which enables the orthogonal load balancing, is an important feature of the system. It is provided through separators and an additional level indirection at each node, as depicted in Figure 3. The separators of an entity are responsible for efficiently relaying remote events among entities as well as tracking the entity whenever it is migrated.

Since remote events are more expensive than local events, it is important to distribute entities throughout the system in a manner that is cognizant of the network overhead imposed. In other words, the system attempts to co-locate entities that communicate frequently in order to minimize the average event traffic overhead. Effective heuristics for load balancing, traffic minimization and entity tracking will be implemented.

## Optimistic Execution of Simulation Time

The model for running discrete event simulations, as previously described, assumes that any two events at the same simulation time may be processed in any serial order or concurrently, if they do not involve the same entity. The events that do not occur at the same simulation time have a strict processing order. This restriction implies that an event may not be invoked on an entity until all previous events on that entity have been processed, including all events that *may* be generated by entities that are still at a previous simulation time. Conservatively, an entity must wait until all other entities finish all the processing of the current simulation time before proceeding.

Optimistically, one could process an event without waiting for all the other entities to synchronize, as shown in Figure 4. This both reduces the synchronization overhead and can, in the optimistic case, greatly increase the opportunity for parallelism. However, it requires the ability to undo the effects of any optimistically processed events, should an event in "the past" be

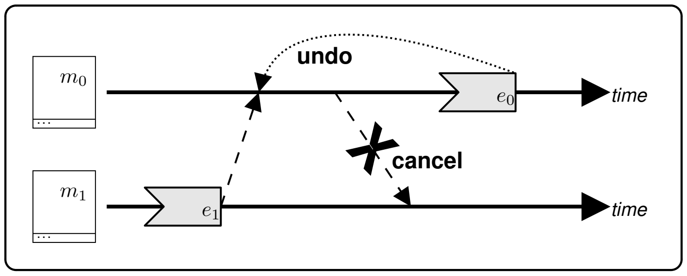

Figure 4: Optimistic execution

subsequently encountered.

JIST transparently supports this undo facility, by automatically check-pointing the state of an entity before processing an event on it. Moreover, to reduce the overhead of check-pointing, an optional *rollback* mechanism is provided, so that the simulation programmer may code explicit rollback methods. In some circumstances, (i.e. for simple methods, such as accessors), these rollback methods may be generated automatically. The system automatically handles the undo of cascading events, either through cancellation or undo propagation, as necessary.

Although optimistic execution provides much greater opportunity for parallelism, each optimistically executed event is only useful if that processing is not later undone. Since undos are more likely when there is a large imbalance in the times of various entities in the system, it is important to keep the forward progress of time on each of the entities roughly balanced, to improve performance. JIST performs this "time" balancing alongside the regular load balancing.

## Additional Comments

The injection of separators into the simulation code provides flexibility and extensibility. In addition to the functionality already described, the separator can carry a payload that is to be executed before and/or after each event on a per entity basis. This allows for a number of useful simulation functions including: orthogonal state inspection and logging, modifying the state of a simulation mid-flight and general-purpose debugging. It is also useful for application-level functionality, such as: updating the location of ad hoc network nodes in an efficient manner according to various mobility models.

The effect of the semantic difference in the programming model described above has far-reaching ramifications for the system. Issues that need to be addressed include: application-level synchronization and threading, pull-based cross-entity state access and blocking functions within event handlers. These issues will be resolved during implementation.

## System Validation Using SWANS

A prototype **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator, or SWANS, will be implemented to validate the JIST design and implementation. The SWANS prototype will contain a basic implementation of the physical (standard radio propagation), link (802.11b), routing (DSR or ZRP) and application (CBR) layers. SWANS will also contain a node entity, which encapsulates an entire simulated ad hoc node, and a few ad hoc mobility models. It is expected that the results of SWANS will be comparable to existing DSR and ZRP ns2 simulations.

## Related Work

- **TimeWarp**: TWOS by Jefferson, et. al., and other projects, such as GTW and U. Calgary
- **Network simulators**: NS2-Monarch, PDNS, OPNet, GlomoSim (see wns proposal)
- **SSI**: Emerald, etc., (see a-exam)
- **PDES**: UCLA and Georgia Tech projects.
- **Rewriting/Instrumentation**: BCEL, Kimera, JOrchestra, Coign, etc.
- **Parallel Java**: cJVM, Jessica, JavaParty

## Project Goals

The following are the goals for the JIST project, though not necessarily in time order:

- Implementation of JIST simulation time transformation
- Implementation of SWANS prototype
- Implementation of JIST parallelization
- Implementation of JIST optimistic execution
- Paper 1: Design of JIST – a paper about the simulation time transformation to a serial simulation execution with some examples from SWANS.
- Paper 2: Parallelization and optimistic concurrency in JIST – a paper about the parallelization and optimistic execution of JIST simulations.
- Thesis: Java In Simulation Time - A scalable, parallel, optimistic discrete event simulation framework.

## Work Outline

I intend (to the best that I can currently foresee) for these goals to be achieved as follows:

**Implementation of JIST simulation time transformation.** The first step of the project is to write a small dummy simulation program, and to successfully rewrite it to run in "simulation time". This involves checking the program statically to ensure various properties, including: no Java synchronization or threading is used, parameters of entity methods are serializable, etc. It also requires a rewrite of the entire program to allow for the insertion of separators between entities. This involves generating separator classes for each entity (as well as corresponding interfaces) with all the functionality described above, most notably the ability to intercept and serialize events in a type-safe manner. It also involves rewriting the rest of the application and modifying byte-code instructions for object creation, assignment, typecasting, field accesses, etc., as well as class definitions of fields and method parameters. This is step is quite technical, as it needs to deal with all the corner-cases of the JVM instruction set and the Java type system. Thereafter, a serial execution engine must be implemented. This involves defining the runtime and how a simulation is loaded and modified, ensuring that the entire implementation is thread-safe (for later parallelization), implementing an efficient event queue and event dispatching mechanism and keeping track of simulation time for each entity. The runtime must also implement the JIST API, most notably the `sleep` instruction to manage time and other simulation properties. Lastly, the runtime should support injection of orthogonal code into the application for purposes of inspection and other purposes, as described above. This section of the implementation is the most difficult and risky part of the project.

**Implementation of SWANS prototype.** The SWANS prototype will be implemented as a plain Java application and will serve as a validation of the JIST idea. In reality, the implementation of SWANS will overlap with and drive the implementation of JIST. SWANS will contain code for physical layer radio propagation, node movement and mobility models, 802.11b, both DSR and ZRP routing protocol, and a CBR traffic-generating application. It is expected to outperform ns2, even in a serial execution, and produce comparable simulation results. This phase of the project is labor intensive and could be split into some M.Eng. projects.

**Paper 1.** The paper will describe the design and benefits of JIST, as outlined above. It will also include DSR and ZRP results of greater scalability than previously published, which will be interesting in and of themselves, in addition to validating the JIST design and implementation.

**Implementation of JIST parallelization.** Extension of the JIST system to support parallel execution should not require any modification of existing application code. Separators need to be extended to support entity migration and tracking independently from the event processing machinery and be able to handle remote events using an efficient serialization mechanism. The runtime must also be extended to track local entities and maintain global virtual time with an appropriate synchronization protocol. Orthogonal to this there must be an inspection mechanism to monitor a simulation in progress and supply data to an independent load balancing and network overhead minimizing algorithm that must also be implemented. Lastly, the runtime will need some mechanism to aggregate and cooperate with other runtimes to form a cluster of simulation machines. A useful M.Eng. project would be to build a GUI monitor and controller for the cluster of JIST runtimes.

**Implementation of JIST optimistic execution.** Extension of the JIST system to support optimistic execution should also not require any modification of existing application, modulo the rollback methods that are discussed shortly. It will, however, require the implementation of the tricky machinery for logging and recovery within the runtime, including both undo propagation and message cancellation. It will also require the integration of separators with this machinery, and the revisiting of a number of topics. For example, it will require a definition of how inspection code is handled (for logging, debugging, etc.), likely using some form of commit semantics. Support for rollback functionality will be added in addition to the automatic check-pointing of state. It may be necessary to add rollback methods to improve performance of existing simulations. The load balancing algorithm will also be modified to incorporate the balancing of forward progress in simulation time to reduce rollbacks and memory requirements.

**Paper 2.** The paper will describe the transparent parallelization and optimistic execution of SWANS and investigate various performance properties of the system, such as overall speedup, the effectiveness of the time balancing in reducing memory consumption for logging, rollback versus check-pointing performance, etc.

**Thesis.** Proposed title is "Java In Simulation Time - A scalable, parallel, optimistic discrete event simulation framework". The thesis will contain the work that went into both papers, provide some additional context for the problem and a chapter to outline possible extensions to the work.