AN EFFICIENT, UNIFYING APPROACH TO SIMULATION USING VIRTUAL MACHINES

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Rimon Barr

May 2004

AN EFFICIENT, UNIFYING APPROACH TO SIMULATION USING VIRTUAL MACHINES

Rimon Barr, Ph.D.

Cornell University 2004

Due to their popularity and widespread utility, discrete event simulators have been the subject of much research. Systems researchers have built many types of simulation kernels and libraries, while the languages community has designed numerous languages specifically for simulation. In this dissertation, I propose a new approach for constructing simulators that leverages virtual machines and thus combines the advantages of both the traditional systems-based and language-based approaches to simulator construction.

I present JiST, a Java-based simulation engine that exemplifies virtual machine-based simulation. JiST executes discrete event simulations by embedding simulation time semantics directly into the Java execution model. The system provides all the standard benefits that the modern Java runtime affords. In addition, JiST is efficient, out-performing existing highly optimized simulation runtimes, and inherently flexible, capable of transparently performing cross-cutting program transformations and optimizations at the bytecode level. I illustrate the practicality of the JiST approach through the construction of SWANS, a scalable wireless ad hoc network simulator that can simulate million node wireless networks, which is more than an order of magnitude in scale over what existing simulators can achieve on equivalent hardware and at the same level of detail.

BIOGRAPHICAL SKETCH

Born on August 17, 1976, in Tel Aviv, Rimon Barr lived in Israel for almost five years, crawling around, eating well, and loving life. He grew up in Johannesburg, South Africa, enjoying soccer, swimming, chess, karate, and reading about anything and everything possible. He fondly remembers his high school, King David Linksfield, family trips to the wilderness, Sunday braais (barbecue in Afrikaans), and the family dogs: Chumi, Shooki, Cookie, Pitsi and Sheba. At the age of 16, his family moved to Toronto, Canada, where he promptly finished up his schooling. He then attended the University of Toronto to study Computer Science and Biology, specializing in Immunology. From the morning subways rides that were mostly uneventful to exciting research projects and competitions, from the litany of lectures to the teaching and other part-time work, and from the scholarly to the social and personal, his undergraduate days were engaging and filled to the brim. The summers were exploited to go traveling and to gain industry experience at places such as Microsoft and IBM. He graduated in 1998 with an Honors B.Sc., magna cum laude. Accepted to the doctoral program at Cornell, he came to Ithaca, NY, the smallest town he has ever lived in. Over the following six years, he became active in a number of community organizations, started to play the guitar, and also completed an MBA degree in 2002. In May of 2004, he will graduate with a Ph.D. in Computer Science, walking for the third time in as many years. He intends to move to New York City shortly thereafter and begin working at Google. The best is yet to come.

To my parents, Aiala and Zeev, and my sister, Iris.

To my friends. To the ongoing pursuit of progress.

ACKNOWLEDGEMENTS

My graduate work has been a humbling, fulfilling, and educational journey: educational, because I have accumulated much knowledge and experience along the way; fulfilling, because I have gathered so many cherished memories and happy moments that have enriched my life; and humbling, because I realize that this accomplishment is as much a product of those around me as of my own efforts. I have so many people to thank for believing in me, for expecting no less of me than the limits of my abilities, for caring about me, for trusting me, for lighting my path, for contributing perspective and wisdom, for giving me strength when my determination faltered, for sharing my load, and for helping me in ways that I may not yet completely comprehend, nor appreciate. Thank you; thank you, all, very much.

I have an incredible family. My mother and father, Aiala and Zeev, have given me only the very best in every respect. My sister, Iris, and I have grown together and are the closest of friends. I simply do not possess the words to express how immensely fortunate and blessed I feel for this. My gratitude and my love for them is beyond description.

I am thankful for my close friends, some of whom are scattered, unfortunately, across the United States, in Canada, in Israel, and elsewhere. They brighten my life and are forged into my life story, as am I into theirs. They all know who they are. Here, at Cornell, I have befriended some wonderful, kind people, who have been integral to my Ithaca experience: Tamara and Josh Goldfarb, Rina Kreitman, Ben Atkin, Adi Bozdog, Tash Katsnelson, Drorit Cohen, Siggi Cherem, Oren Kurland, Rivka Shoulson, Alin Dobra, Oren Harel, Boaz Nachshon, April Scheck, Dayana Habib, Eli and Regina Barzilay, Rabbis Eli Silberstein, Ed Rosenthal and Avi

Scharf, Leron Thumim, Josh Rosenthal, Nechama Poyurs, Paul Ampadu, Steve Fisher, and Ranveer Chandra. To those few that I have, no doubt, been unable to recall at this moment, I apologize. I sometimes catch myself searching for my eye-glasses when they are, in fact, sitting on my nose. The people who I have regretfully not mentioned would be those that, like my eye-glasses, are the most unobtrusively integral to my daily life. I thank you.

I am immensely grateful to my doctoral committee for shepherding me through the ups and downs of this degree. I wish to thank my adviser, Zygmunt Haas for his invaluable guidance and ongoing input into this work, and for his trust in me, his kindness, and his humor. I am greatly indebted to Alan McAdams for the wisdom and experience he shared with me and, most of all, for his friendship. I thank Robbert van Renesse for his kindness and sincerity towards me, and for inspiring me with his incredible creativity and talent. I thank Ken Birman for believing in me, for encouraging and helping me, and for inspiring me with his sharp insight and keen judgment. These individuals exemplify many of the ideals of academia, and I am honored to have had the opportunity to interact with them. Finally, I wish to thank all of my teachers, of which there have been so many, both during my university years and before, both in an official capacity and not, for the gifts that they have bestowed upon me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

From physics to biology, from weather forecasting to predicting the performance of a new processor design, and from estimating the expected value of an abstract financial instrument to the modeling of fracture propagation in concrete dams, people in many avenues of science and industry increasingly depend on software simulations. Simulators are used to model various realistic phenomena and also hypothetical scenarios that often cannot be satisfactorily expressed analytically nor easily reproduced and observed empirically. Instead, discretized simulation models are derived and then encoded as event-driven programs, wherein events are processed in their casual order, updating the simulation program state according to the given model and possibly scheduling more simulation events.

1.1 Existing approaches to simulation construction

Due to their popularity and widespread utility, discrete event simulators have been the subject of much research [71, 36, 77, 37] directed at their efficient design and execution. Systems researchers have built many types of simulation *kernels* and *libraries*, spanning the gamut from the conservatively parallel to the aggressively optimistic, and from the shared memory to the message passing paradigms. The languages community has designed numerous *languages* specifically for simulation, which codify event causality, execution semantics, and simulation state constraints, simplify parallel simulation development, and permit important static and dynamic optimizations.

Simulation ***kernels***, including systems such as the seminal TimeWarp OS [54], transparently create a convenient simulation time abstraction. By mimicking the system call interface of a conventional operating system, one can run simulations comprised of *standard*, unmodified programs. However, since the kernel controls process scheduling, inter-process communication and the system clock, the kernel can run its applications in simulation time. For example, an application `sleep` request can be performed without delay, provided that causal relationships between communicating processes are preserved. Moreover, the kernel can transparently support concurrent execution of simulation applications and even speculative and distributed execution. Thus, the process boundary provides much flexibility.

Unfortunately, the process boundary is also a source of inefficiency [16]. Simulation ***libraries***, such as Compose [68] and others, trade away the *transparency* afforded by process-level isolation in favor of increased *efficiency*. For example, by combining the individual processes into a single simulation process, one can eliminate the process context-switching and marshaling overheads required for event dispatch and thus increase simulation efficiency. However, various simulation functions that existed within the kernel, such as message passing and scheduling, must then be explicitly programmed in user-space. In essence, the simulation kernel and its applications are merged into a single monolithic process that contains both the simulation model as well as its own execution engine. This monolithic simulation program is more complex and littered with simulation library calls and callbacks. The library may also require certain coding practices and program structure that are not explicitly enforced by the compiler. This level of detail not only encumbers efforts to transparently parallelize or distribute the simulator, it also impedes possible high-level compiler optimizations and obscures simulation correctness.

Table 1.1: Trade-offs of different simulator construction approaches.

	kernel	library	language	**JiST**
transparent	++		++	++
efficient		+	+	++
standard	++	++		++

Simulation ***languages***, such as Simula [30], Parsec [11] and many others, are designed to simplify simulation development and to explicitly enforce the correctness of monolithic simulation programs. Simulation languages often introduce execution semantics that transparently allow for parallel and speculative execution, without any program modification. Such languages often also introduce handy constructs, such as messages and entities, that can be used to partition the application state. Constraints on simulation state and on event causality are statically enforced by the compiler, and they also permit important static and dynamic optimizations. An interesting recent example of a language-based simulation optimization is that of reducing the overhead of speculative simulation execution through the use of reverse computations [26]. However, despite these advantages, simulation languages are domain-specific by definition and therefore suffer from specialization. They usually lack modern features, such as type safety, reflection and garbage collection, as well as portability. They also lag in terms of general-purpose optimizations and implementation efficiency. These deficiencies only serve to perpetuate the small user-base problem, but perhaps the most significant barrier to adoption by the broader community is that programs need to be rewritten in order to be simulated.

In summary, each of these three fundamental approaches to simulation construction trades off a different desirable property, as shown in Table 1.1, where:

- **standard** means writing simulations in a conventional, popular programming language, as opposed to a domain-specific language designed explicitly for simulation;
- **efficient** denotes optimizing the simulation program statically and dynamically by considering simulation state and event causality constraints in addition to general-purpose program optimizations; creating a simulation engine that compares favorably with existing, highly optimized systems both in terms of simulation throughput and memory consumption, and; possibly distributing the simulation and executing it in parallel or speculatively across the available computational resources to improve performance;
- and **transparent** implies the separation of efficiency from *correctness*; that correct simulation programs can be automatically transformed to run efficiently without the insertion of simulation-specific library calls or other manual program alterations. **Correctness** is an assumed pre-condition that simulation programs must compute valid and useful results, regardless of how they are constructed.

1.2 Virtual machine-based simulation

In this dissertation, I propose a new, unifying approach to building simulators, which does not suffer this trade-off: to bring simulation semantics to a modern and popular virtual machine-based language. Unlike prior simulator designs, virtual machine-based simulation requires:

- *neither* a new simulation *language* – new languages, and especially domain-specific ones, are rarely adopted by the broader community;
- *nor* a new simulation *library* – libraries frequently require developers to clutter their code with simulation-specific library calls and impose unnatural program structure to achieve performance;
- *nor* a new simulation system *kernel* or language runtime – custom kernels and language runtimes are rarely as optimized, reliable, featured, or portable as their generic counterparts.

Instead, JiST, which stands for **J**ava **i**n **S**imulation **T**ime, embeds simulation semantics into the *standard* Java language and its runtime, and *transparently* performs important high-level simulation optimizations, resulting in an *efficient* discrete event simulation platform. The three attributes of virtual machine-based simulation – the first one in particular – highlight an important distinction between JiST and previous simulation system designs in that the simulation code that runs atop 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 runtime simulation functionality. Instead, JiST transparently introduces simulation time execution semantics to simulation programs written in plain Java and they are executed over an unmodified Java virtual machine. In other words, JiST converts a virtual machine into a simulation system that is flexible and efficient.

1.3 Thesis statement

Using the terminology just defined, I propose the following thesis:

> *A virtual machine-based simulator benefits from the advantages of both the traditional systems and language-based designs by leveraging standard compilers and language runtimes as well as ensuring efficient simulation execution through transparent cross-cutting program transformations and optimizations.*

1.4 Contributions

Thus, the primary contribution of this dissertation is virtual machine-based simulation, a new, unifying approach to building simulators. I outline the rationale for this new design and discuss its many benefits. I present the JiST prototype, a general-purpose Java-based simulation platform that embodies the virtual machine-based simulator design. JiST embeds simulation execution semantics directly into the Java virtual machine. The system provides all the standard benefits that the modern Java runtime affords. In addition, JiST is efficient, outperforming existing highly optimized simulation runtimes, and inherently flexible, capable of transparently performing cross-cutting program transformations and optimizations. I leverage this flexibility to introduce additional concepts into the JiST model, including process-oriented simulation and simulation time concurrency primitives. In this dissertation, I also present SWANS, a wireless ad hoc network simulator, built atop JiST, as a validation of the approach, and demonstrate that SWANS can scale to very large wireless network simulations even on

commodity machines. Finally, I utilize SWANS to perform a scalable analysis of the bordercast query propagation protocol.

1.5 Outline

The remainder of this dissertation is organized as follows:

Chapter 2 - introduces the fundamental ideas of virtual machine-based simulation.

Chapter 3 - continues the discussion with important extensions to the basic model that either simplify development or improve performance.

Chapter 4 - presents SWANS, a wireless network simulator that leverages the JiST design to achieve performance and scalability.

Chapter 5 - provides an evaluation of JiST and SWANS using various performance and software engineering benchmarks.

Chapter 6 - is a SWANS-based study of the scalability properties of the bordercast query propagation protocol.

Chapter 7 - discusses the body of prior and related work.

Chapter 8 - concludes with a short summary and possible future work.

1.6 Software and documentation

As of the time of this writing, the JiST and SWANS implementations (with complete source code), as well as user guides and related presentations, are currently available, for non-commercial academic use, on the World Wide Web at:

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

While this dissertation represents the current state of the project, it is expected (and hoped) that the project will continue to progress beyond its current functionality. Possible future directions are discussed in section 8.2.

Chapter 2

System Overview and Design

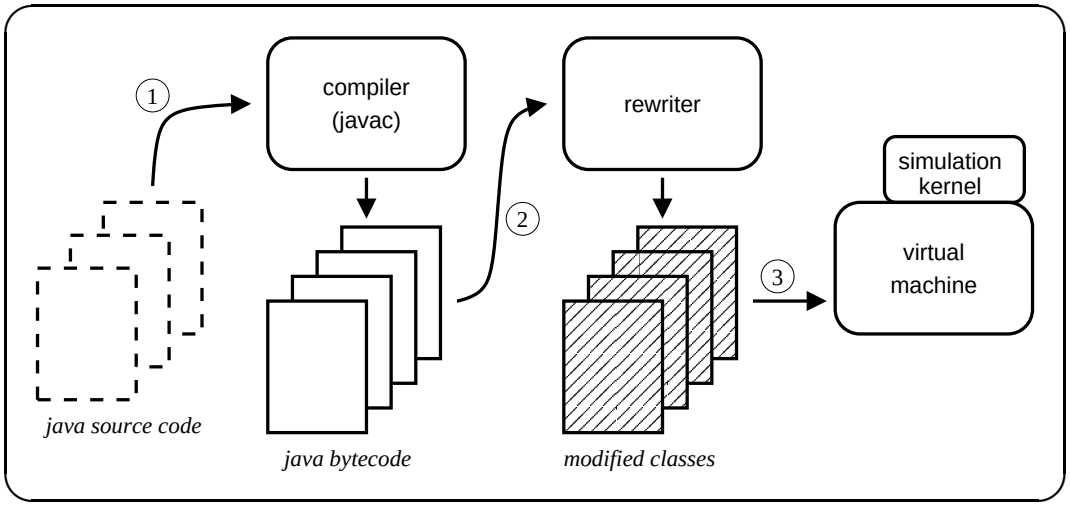
As introduced in the previous chapter, JiST converts a virtual machine into a simulation platform. This chapter outlines the basic system architecture, explains what it means to execute a program in *simulation time*, and describes how JiST supports the simulation time abstraction by extending the basic Java object model and its execution semantics.

2.1 Architecture

The JiST system consists of four distinct components: a compiler, a language runtime or virtual machine, a rewriter and a language-based simulation time kernel. Figure 2.1 presents the JiST architecture: (1) a simulation is first compiled, then (2) dynamically rewritten as it is loaded, and finally (3) executed by the virtual machine with support from the language-based simulation time kernel.

A primary goal of JiST is to execute simulations using only a standard language and runtime. Consequently, the compiler and runtime components of the JiST system can be any standard Java compiler and virtual machine[1], respectively. 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 they otherwise completely preserve the existing program

[1]JiST was built and tested with the Java 2 v1.4 JDK, the Sun `javac` and IBM `jikes` compilers, and the Sun HotSpot and IBM JVMs on both the Linux and Windows platforms.



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 are supported at runtime by the simulation time kernel.

Figure 2.1: The JiST system architecture.

logic. The program transformations occur once, at load time, and do not incur rewriting overhead during execution. The rewriter also does not require source-code access, since this is a bytecode to bytecode transformation. At runtime, the modified simulation classes interact with the JiST simulation time *kernel* through the various injected or modified operations. The JiST kernel is written entirely in Java, and it is responsible for all the runtime aspects of the simulation time abstraction. For example, it keeps track of simulation time, performs scheduling, and ensures proper synchronization.

2.2 Simulation time execution

The JiST rewriter modifies Java-based applications and runs them in *simulation time*, a deviation from the standard Java virtual machine (JVM) bytecode execution semantics [63]. Under the standard Java execution model, which I refer to as *actual time* execution, the passing of time is not explicitly linked to the progress of the application. In other words, the system clock advances regardless of how many bytecode 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 does not make strong guarantees regarding timely program progress. It may decide, for example, to perform garbage collection at any point.

Under *simulation time* execution, the progress of time is made dependent 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 simulation time has been completed. One could contrast simulation time execution with *real time* execution, wherein the runtime guarantees that in-

Table 2.1: The relationship between program progress and time under different execution models

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

structions or sets of instructions will meet given deadlines. In this case, the rate of application progress is made dependent on the passing of time. The different execution models are summarized in Table 2.1.

The notion of simulation time itself is not new: simulation program writers have long been accustomed to explicitly tracking the simulation time and explicitly scheduling simulation events in time-ordered queues [70]. The simulation time concept is also integral to a number of simulation languages and simulation class libraries. The novelty of the JiST system is that it embeds simulation time semantics into the standard Java language, which allows the system to transparently run the resulting simulations efficiently. Under simulation time execution, individual application bytecode instructions are processed sequentially, following the standard Java control flow semantics. However, the simulation time will remain unchanged. Application code can only advance simulation time via the `sleep(`n`)` system call. In essence, every instruction takes zero simulation time to process except for `sleep`, which advances the simulation clock forward by exactly n simulated time quanta, or ticks. In other words, the `sleep` function advances time under simulation time execution, just as it does under actual time execution. The primary difference is

that, under simulation time execution, all the *other* program instructions do not have the side-effect of allowing time to pass as they are processed.

Thus, JiST is not intended to simulate the execution of arbitrary Java programs. In other words, JiST is not a virtual machine simulator. Rather, it is a virtual machine-based simulation platform atop which discrete event simulation programs can be built and executed. JiST processes application events in their simulation-temporal order, until all queued events are exhausted or until a pre-determined ending time is reached, whichever comes first. This simulation program could be modeling anything from a wireless network to a peer-to-peer application to a new processor design to the execution of a Java program inside a simulated virtual machine. The structure of such simulation programs is described next.

2.3 Object model and execution semantics

JiST simulation programs are written in Java [42], an object-oriented language. Thus, the entire simulation program comprises numerous classes that collectively implement its logic and the state of the program is contained within individual objects during its execution. Interactions among object are represented syntactically as method invocations.

JiST extends this traditional programming model with the notion of simulation *entities*, defined syntactically as instances of classes that implement the empty `Entity` interface. Every simulation object must be logically contained within an entity, where object containment within an entity is defined in terms of its reachability: the state of an entity is the combined state of all objects reachable from it. Thus, although entities are regular objects within the virtual machine at runtime,

they serve to logically encapsulate application objects, as shown in Figure 2.2(a). Entities are components of a simulation and represent the granularity at which the JiST kernel manages a running simulation.

Each entity has its own simulation time and may progress through simulation time independently. Thus, an entity cannot share its state with any other entity, otherwise there could be an inconsistency in the state of the simulation. In other words, each (mutable) object of the simulation must be contained within exactly one entity. Since Java is a safe language, this constraint is sufficient to partition the simulation into a set of non-overlapping entities and also prevents unmediated communication across entity boundaries.

All instructions and operations *within* an entity follow the regular Java control flow and semantics. They are entirely opaque to the JiST infrastructure. Specifically, object method invocations remain unchanged. The vast majority of the entity code is involved with encoding the logic of the simulation model and is entirely unrelated 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 JiST system calls.

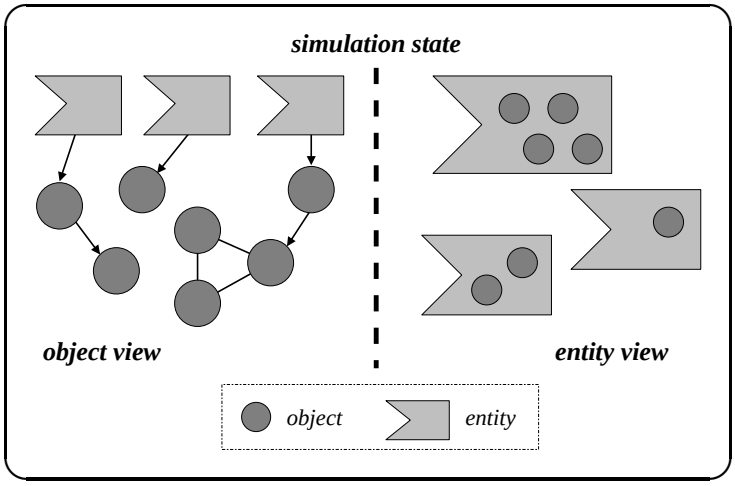
In contrast, invocations on entities correspond to simulation events. The execution semantics are that method invocations on entities are non-blocking. They are merely queued at their point of invocation. The invocation is actually performed on the callee (or target) entity only when it reaches the same simulation time as the calling (or source) entity. In other words, cross-entity method invocations act as synchronization points in simulation time. Or, from a language-oriented perspective, an entity method is like a coroutine, albeit scheduled in simulation time. This is a convenient abstraction in that it eliminates the need for an explicit

simulation event queue. It is the JiST kernel that actually runs the event loop, which processes the simulation events. The kernel invokes the appropriate method for each event dequeued in its simulation time order and executes the event to completion without continuation.

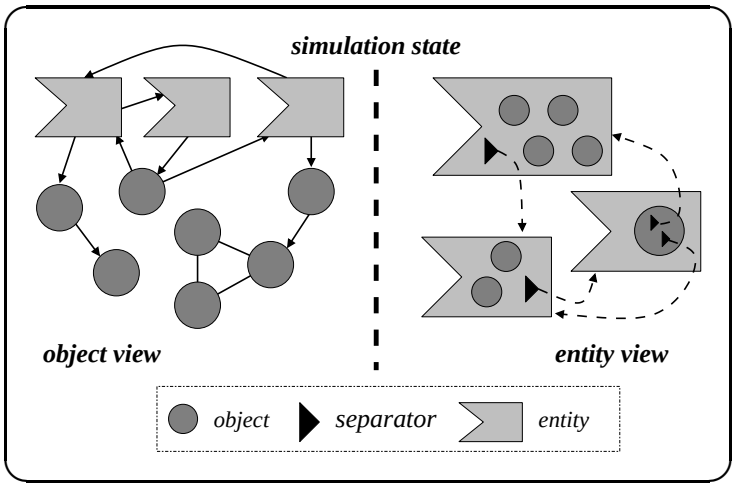
However, in order to invoke a method on another entity – to send it an event – the calling entity must hold some kind of reference to the target entity, as depicted in Figure 2.2(b). One must, therefore, distinguish between object references and entity references. All references to a given (mutable) object must originate from within the same entity. References to entities are free to originate from any entity, including from objects within any entity. The rationale is that object references imply inclusion within the state of an entity, whereas entity references represent channels along which simulation events are transmitted. As a consequence, entities do not nest, just as regular Java objects do not.

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

Due to this imposed separation, JiST guarantees that interactions among entities can only occur via the simulation kernel. This is similar in spirit to the JKernel design [49] in that it provides language-based protection and zero-copy inter-entity communication. However, JKernel is designed to provide traditional



(a) 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.



(b) At runtime, entity references are transparently replaced with separators, which both preserves the separation of entity state and serves as a convenient point to insert functionality.

Figure 2.2: Extending the Java object model.

system services, such as process-level protection, within a safe-language environment, whereas JiST is designed explicitly for simulation. For example, whereas JKernel utilizes native Java threads for concurrency, JiST introduces entities. Entities provide thread-free event-based simulation time concurrency, which facilitates scalable simulation.

The separators, in effect, represent an application state-time boundary around each entity, similar to a TimeWarp [54] process, but at a finer granularity. They are a convenient point to insert additional simulation functionality. By tracking the simulation time of each individual entity, these separators allow for concurrent execution. By adding the ability to checkpoint entities, the system may support speculative execution as well. Finally, separators also provide a convenient point for the distribution of entities across multiple machines. In a distributed simulation, the separators function as remote stubs and transparently maintain the abstraction of a single system image, by storing and tracking the location of entities as they migrate among machines in response to fluctuating processor, memory, and network loads.

The role of the simulation developer, then, is to write the simulation model in regular Java and to partition the program into multiple entities along reasonable application boundaries. This is akin to, and no more difficult than partitioning the application into separate classes. The JiST infrastructure will efficiently execute this program, comprised of entities of objects, while retaining its simulation time semantics.

The JiST model of execution, known as the *concurrent object model*, is similar to, for example, the Compose [68] 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 the Parsec language [11], wherein explicit blocking send and blocking 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 simulation library. Unlike Parsec, JiST embeds itself within the Java language and does not require new language constructs. And, with the introduction of continuations in chapter 3, these two simulation models will even be able to co-exist.

2.4 Simulation kernel interface

JiST simulations run atop the simulation kernel and interact with it via a short API. The entire JiST API is exposed at the language level via the `JistAPI` class listed partially in Figure 2.3.

The `Entity` interface tags a simulation object as an entity, which means that invocations on this object follow simulation time semantics: method invocations become events that are queued for delivery at the simulation time of the caller. The `getTime` call returns the current simulation time of the calling entity, which is the time of the current event being processed plus any additional sleep time. The `sleep` call advances the simulation time of the calling entity. The `endAt` call specifies when the simulation should end. The `THIS` self-referencing entity reference is analogous to the Java `this` object self-reference. It refers to the entity for which an event is currently being processed and is rarely needed. The `ref` call returns a separator stub of a given entity. All statically detectable entity references are automatically converted into separator stubs by the rewriter. This function

JistAPI.java

```
package jist.runtime;

class JistAPI
{
    interface Entity { }
    long getTime();
    void sleep(long ticks);
    void end();
    void endAt(long time);
    void run(int type, String name, String[] args, Object props);
    void runAt(Runnable r, long time);
    void setSimUnits(long ticks, String name);
    interface Timeless { }
    interface Proxiable { }
    Object proxy(Object proxyTarget, Class proxyInterface);
    class Continuation extends Error { }
    Channel createChannel();
    interface CustomRewriter {
     JavaClass process(JavaClass jcl);
    }
    void installRewrite(CustomRewriter rewrite);
    interface Logger
    {
     void log(String s);
    }
    void setLog(JistAPI.Logger logger);
    void log(String s);
    JistAPI.Entity THIS;
    EntityRef ref(Entity e);
}
```

The partial JiST system call interface shown above is exposed at the language level via the `JistAPI` class. The rewriter replaces these with their runtime implementations.

Figure 2.3: The JiST kernel interface

is included only to deal with rare instances, when entity types might be created dynamically, and for completeness.

The remaining elements of the API will be explained as the corresponding concepts are introduced. The complete listing can be found in Appendix A.

Note that, although it was possible to silently modify (i.e., rewrite) the meaning of some existing Java functions, such as `Thread.sleep` and `System.currentTimeMillis` instead of introducing new `JistAPI` functions, it was decided against this kind of syntactic sugar. First, not all simulation time primitives have Java counterparts and it is advantageous to keep all the simulation-oriented functions together. Second, this approach makes simulation-oriented primitives explicit, and preserves existing functionality.

2.5 Hello World!

The basic simulation primitives just introduced allow us to write 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 2.4. Note, first, that this is a valid Java program. You can compile it with a regular Java compiler, and run it as a regular Java program. But, to run `hello` with simulation semantics, it should be run atop JiST within a standard virtual machine.

This simplest of simulations highlights some important points. To begin, the `hello` class is an entity, because it implements the `Entity` interface (line 4). Entities can be created (line 7) and their methods invoked (lines 8 and 14) just as any regular Java object. The entity method invocation, however, happens in simulation time. This is most apparent on line 14, which is a seemingly infinite recursive

hello.java

```
1  import jist.runtime.JistAPI;
2
3  class hello implements JistAPI.Entity
4  {
5    public static void main(String[] args) {
6      System.out.print("start simulation");
7      hello h = new hello();
8      h.myEvent();
9    }
10
11   public void myEvent()
12   {
13     JistAPI.sleep(1);
14     myEvent();
15     System.out.print("hello world, t=" +
16       JistAPI.getTime());
17   }
18 }
```

The simplest of simulations, shown above, consists of a single entity that emits a message at each time step.

Figure 2.4: Hello World! simulation

call. In fact, if this program is run under a regular Java virtual machine (i.e., without JiST rewriting) then the program would abort with 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 13), then schedule a new event at that future time, and finally print a hello message with the entity simulation time (line 16). Instead of a stack overflow, the program runs in constant stack space and the output is:

```
> simulation start
> hello world, t=1
> hello world, t=2
> hello world, t=3
> etc.
```

The `JistAPI` class, used on lines 4, 13 and 16, represents the application interface exposed by the simulation kernel. If one executes the `hello` program without the JiST runtime, simply under a regular Java runtime, there will be no active simulation kernel. In this case, the entire `JistAPI` acts as a dummy class that merely facilitates type-safe execution: the `Entity` interface is empty, the `sleep` call does nothing and the `getTime` function returns zero. The correct way to think about the `JistAPI` is that it marks the program source in a manner that both respects Java syntax and is preserved through the compilation process. It allows type-safe compilation of JiST simulation programs using a conventional Java compiler.

When running the `hello` simulation within JiST, the simulation kernel will be loaded and running. Among other things, this kernel installs a class loader into the JVM, which dynamically rewrites the `hello` bytecode as it is loaded. The various `JistAPI` markings within the compiled simulation program serve to direct the code transformations that introduce the simulation time semantics into the

bytecode. The entire JiST functionality is exposed to the simulation developer in this manner. Thus, JiST extends the object model of the Java language and the execution semantics of its virtual machine. Nevertheless, the language, as well as its compiler and virtual machine, are reused.

2.6 Rewriter

The notable pieces of the JiST system are the bytecode rewriter and the simulation time kernel, since these components introduce and support the simulation time execution semantics, respectively. 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 2.2(b), 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 bytecode 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 reflection-based calling conventions. The bytecode 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` 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. These will be discussed in chapter 3.

For ease-of-use, the JiST rewriter is implemented as a dynamic class loader. It uses the Byte-Code Engineering Library [31] to automatically modify the simulation program bytecode 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 a separate offline process.

2.7 Simulation 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 events 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 an event queue structure for every entity in the system, and it is thus able to respond to application `getTime` requests and time-stamp outgoing events. The kernel queues events on behalf of each entity, and automatically advances the entities though simulation time, delivering events for application processing as appropriate. And, finally, the kernel supports various system maintenance functions, such as a entity garbage collection, load balancing and application monitoring.

Simulation processing begins via an anonymous bootstrap entity with a single scheduled event: 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 pre-determined 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 in chapter 3.

In general, the design of the JiST simulation time kernel is similar to that of the TimeWarp Operating System [54] 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 or memory copy. 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, the JiST kernel can transparently support efficient checkpointing and rollback of entities using language-based serialization and reflection.

2.8 Summary

JiST is a prototype virtual machine-based simulator that combines the traditional language-based and systems-based approaches to simulator design. It allows simulations to be written in plain Java and then executed over a standard Java virtual machine, yet introduces simulation time semantics and other simulation constructs and optimizations using bytecode-level program transformations. This chapter, I have described the fundamentals of simulation time execution, and the various components of the JiST architecture that support it.

Chapter 3

Flexibility of Virtual Machine-based Simulation

One of the key advantages of virtual machine-based simulation is its inherent flexibility. This chapter is devoted to a discussion of various extensions to the basic model described in chapter 2, introducing concepts such as timeless objects, proxy entities, blocking events, and simulation time concurrency. Many of these additions are completely orthogonal to the simulation program. In other cases, a few annotations within the simulation code drive high-level optimizations and cross-cutting transformations performed by the rewriter, akin to aspect-oriented programming [61]. Moreover, the ease with which these enhancements are integrated into the basic design underscores the flexibility of the JiST approach and suggests that it is a compelling vehicle for ongoing simulation research.

3.1 Zero-copy semantics

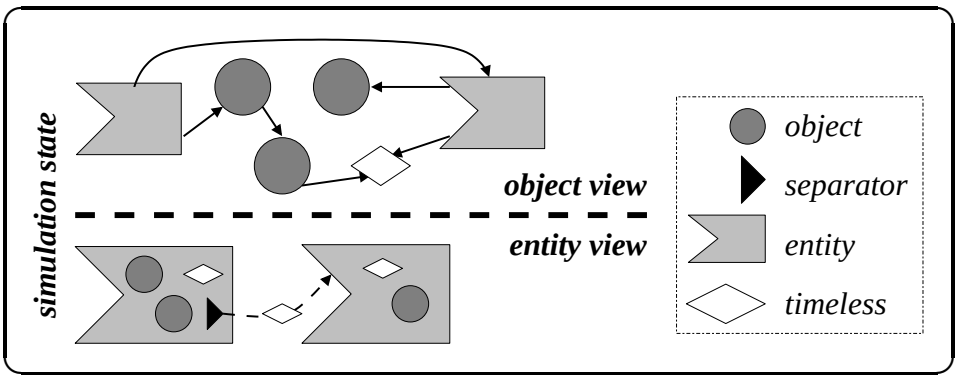
The first extension to the model is that of *timeless* objects. A timeless object is defined as one that will not change over time. Knowing that a value is temporally stable allows the system to safely pass it across entities by reference, rather than by copy, significantly improving event throughput.

The system may be able to statically infer that an object is transitively open-world immutable [18] and automatically infer that event parameters are timeless. However, any static analysis will be overly conservative at times. Thus, one can also explicitly request zero-copy semantics by using the `JistAPI.Timeless` interface to

tag an object. The annotation implies that the object will not be modified at any time after references to it escape an entity boundary. The addition of a single tag, or the automatic detection of the timeless property, conveniently affects *all* the events throughout the simulation that contain parameters of this type. Such cross-cutting transformations lie at the heart of the JiST design, providing transparency, as defined back in chapter 1: the separation of concerns related to correctness and execution efficiency.

In addition to the performance benefits of eliminating a memory copy, the timeless tag is also useful for sharing state among entities to reduce simulation memory consumption, as depicted in Figure 3.1. For example, network packets are defined to be timeless in SWANS, a JiST-based wireless network simulator (see chapter 4), in order to prevent unnecessary duplication: broadcasted network packets are therefore not copied for every recipient, nor are they copied into the various sender retransmit buffers. Similarly, one can safely share object replicas across different instances of a simulated peer-to-peer application. This sharing of immutable state among entities is key to the efficient utilization of available memory and to handling larger simulation models.

Note, however, that the explicit tagging facility should be exercised with care: shared objects *must* actually be treated as immutable once they escape an entity boundary. Otherwise, such shared objects can lead to temporal inconsistencies within the simulation state, because individual entities may progress through simulation time at different rates. Conversely, the simulation developer should be aware that non-timeless objects are passed across entities by copy, which differs from regular Java call-by-reference semantics for non-primitive parameters. As a direct consequence, it is impossible to transmit information back to the caller using



Objects passed across entities should be timeless – in other words, should hold temporally stable values – to prevent temporal inconsistencies in the simulation state. Sharing timeless objects among entities is an effective way to conserve simulation memory and using zero-copy semantics improves simulation throughput.

Figure 3.1: Timeless objects.

a (mutable) object parameter of an event. Two separate events should be used instead.

3.2 Reflection-based configuration

An important consideration in the design of simulators is configurability: the desire to reuse the simulator for many different experiments. However, this can adversely affect performance. Configuration is usually supported either at the *source-code* level, via *configuration files*, or with *scripting* languages.

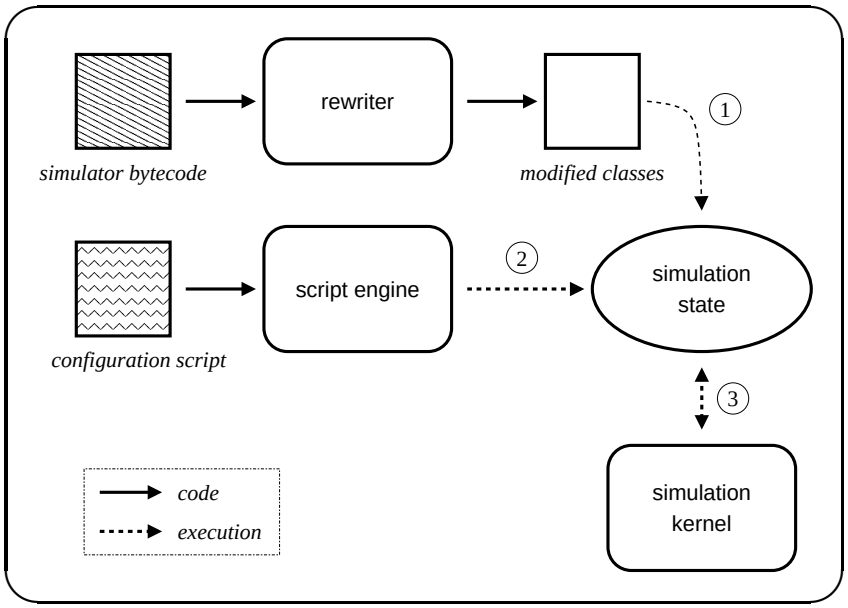
Source-level configuration entails the recompilation of the simulation program before each run with hard-coded simulation parameters and linkage with a small driver program for simulation initialization. This approach to configuration is flexible and runs efficiently, because the compiler can perform constant propagation and other important optimizations on the generic simulation code to produce a specialized and efficient executable. However, it requires recompilation on each run.

The use of ***configuration files*** eliminates the need for recompilation. The configuration is read and parsed by a generic driver program as it initializes the simulation. This option is not only brittle and limited to pre-defined configuration options, it eliminates opportunities for static compiler optimizations.

The ***script-based configuration*** approach is championed by ns2. A scripting language interpreter – Tcl, in the case of ns2 – is backed by the compiled simulation runtime, so that script variables are linked to simulation values, and a script can then be used to instantiate and initialize the various pre-defined simulation components. Unfortunately, the linkage between the compiled simulation components and the configuration scripts can be difficult to establish. In ns2, it is achieved

manually via a programming pattern called *split objects*, which requires a language binding that channels information in objects within the compiled space to and from objects in the interpreted space. This not only clutters the core simulation code, but it is also inefficient, because it duplicates information. Furthermore, script performance depends heavily on this binding. The choice of combining C with Tcl in ns2, for example, imposes excessive string manipulation and leads to long configuration times. More importantly, this approach eliminates static optimization opportunities, which hurts performance. It also results in the loss of both static and dynamic type information across the compiled-interpreted interface, thereby increasing the potential for error.

In contrast, JiST-based simulations enjoy both the flexibility of script-based configuration and the performance advantages of source-level configuration. The scripting functionality comes "for free" in JiST. It does not require any additional code in the simulation components, nor any additional memory. And, the script can configure a simulation just as quickly as a custom driver program. This is because Java is a dynamic language that supports reflection. As illustrated in Figure 3.2, the access that the script engine has to the simulation state is just as efficient and expressive as the compiled driver program. A script engine can query and update simulation values by reflection for purposes of tracing, logging, and debugging, and it can also dynamically pre-compile the driver script directly to bytecode for efficient execution. The simulation bytecode itself is compiled and optimized dynamically as the simulation executes. Thus, simulation configuration values are available to the Java optimizer and allow it to generate more efficient and specialized code. The information available to the optimizer at runtime is a super-set of what is available to a static simulation compiler. Finally, while the



JiST provides multiple scripting interfaces to configure its simulations without source modification, memory overhead, or loss of performance. (1) As before, simulation components are loaded and rewritten on demand. (2) The script engine configures the simulation via reflection and may dynamically compile scripts to bytecode for performance. (3) The simulation runs as before, interacting with the kernel as necessary.

Figure 3.2: Reflection-based configuration

hello.bsh

```
1 System.out.println("starting simulation from BeanShell!");
2 import jist.minisim.hello;
3 hello h = new hello();
4 h.myEvent();
```

hello.jpy

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

Figure 3.3: Example BeanShell and Jython script-based simulation drivers.

scripting language environment may support a more relaxed type system, the type-safety of the underlying simulation components is still guaranteed by the virtual machine, facilitating earlier detection of scripting errors.

The script functionality is exposed via the `JistAPI`, so that simulators may also embed domain-specific configuration languages. JiST supports the BeanShell engine, with its Java syntax, and also the Jython engine, which interprets Python scripts. Java-based engines for other languages, including Smalltalk (Bistro), Tcl (Jacl), Ruby (JRuby), Scheme (Kawa), JavaScript (Rhino), or a custom simulation definition language interpreter can easily be integrated as well and they can even co-exist. As is the case with compiled driver programs, the driver script is invoked within its host script engine by the JiST kernel. The script is the first, bootstrap simulation event. Example scripts are shown in Figure 3.3.

The scripting functionality within JiST is truly orthogonal to the simulation components. It requires no additional code within the component implementations, no memory overhead for simulation state, and no performance overhead over the equivalent compiled simulation driver programs.

3.3 Tight event coupling

Under JiST, simulation events are encoded as method invocations, which reduces the amount of simulation code required and improves its clarity without affecting runtime performance. The benefits of this encoding are summarized in Table 3.1. The first benefit is type-safety, which eliminates a common source of error: the source and target of an event are statically checked by the Java compiler. Secondly, the event type information is also managed automatically at runtime, which completely eliminates the many event type constants and associated event type-cast

code that are otherwise required. A third benefit is that marshaling of event parameters into the implicit event structures is performed automatically. In contrast, simulators written against event-driven libraries often require a large number of explicit event structures and code to simply pack and unpack parameters from these structures. Finally, debugging event-driven simulators can be onerous, because simulation events arrive at target entities from the simulation queue without any context. Thus, it can be difficult to determine the cause of a faulty or unexpected incoming event. In JiST, an event can automatically carry its context information: the point of dispatch (with line numbers, if source information is available), as well as the state of the source entity. These contexts can then be chained to form an event causality trace, which is the equivalent of a stack trace, but is far more useful in an event-driven application. For performance reasons, this information is collected only with the kernel in debug mode, but no changes to the application are required

The tight coupling of event dispatch and delivery in the form of a method invocation also has important performance implications. Tight event-loops, which can be determined only at runtime, can be dynamically optimized and inlined even across the kernel boundary between the JiST kernel and the running simulation, as first shown by the Jalapeño project [3]. For example, the dynamic Java compiler may decide to inline portions of the kernel event queuing code into hot spots within the simulation code that frequently enqueue events. Or, conversely, small and frequently executed simulation event handlers may be inlined into the kernel event loop. The tight coupling also abstracts the simulation event queue, which will, in the future, allow the JiST kernel to transparently execute the simulation more

Table 3.1: Benefits of encoding simulation events as entity method invocations.

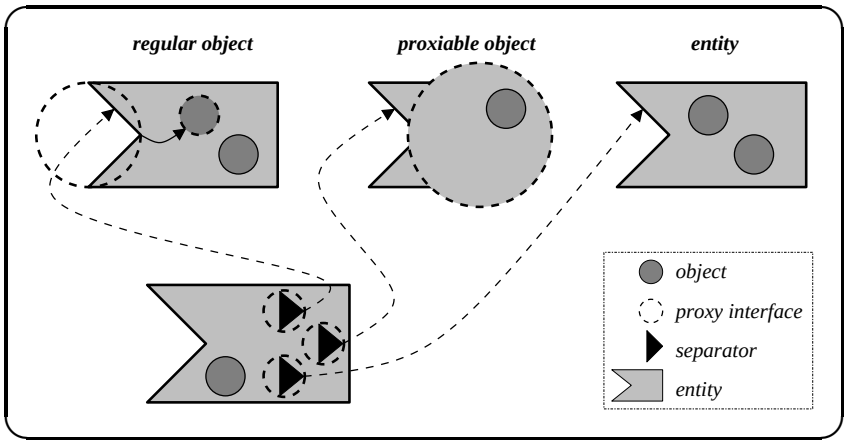
type safety	-	source and target of event statically checked by compiler
event typing	-	not required; events automatically type-cast as they are dequeued
event structures	-	not required; event parameters automatically marshaled
debugging	-	location of event dispatch and state of calling entity available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

efficiently – in parallel, distributed, and even optimistically – without requiring any modifications to the simulation code, as discussed in section 3.7.

3.4 Interface-based entities

Entities encapsulate simulation state and present an event-oriented interface using methods. However, methods also represent regular Java invocations on objects. Thus, method invocations have two meanings; there exists a clash of functionality between entity and object method invocation semantics at the syntactic level. This imposes restrictions on the developer and can lead to awkward coding practices within entity classes. One would prefer both kinds of method invocations to co-exist, and this is exactly the purpose of *proxy* entities. Proxy entities are interface-based entities that relay events onto internal target objects. They are created via the `proxy` system call, which accepts a proxy target and one or more proxy interfaces. The proxy target can be one of three types: a regular object, a *proxiable* object, or an entity, as illustrated in Figure 3.4 and described below. The proxy interface indicates which methods will be exposed and relayed. Clearly, the proxy target must implement the proxy interface and this is also verified by the system.

Regular objects do not contain the machinery to receive events from the kernel, so they are wrapped, along with any reachable objects, within a new entity, which relays methods to the given target object that it encloses. The effect of this entity wrapping is to insert an additional method invocation into the event delivery process. Note, also, that if one invokes the *proxy* call twice on the same object, the result is two new entities that share that object in their state, as well as all those reachable from it. This indirection can, however, be eliminated through the use of *proxiable* objects.



Proxy entities use an interface-based approach to identify entity methods, thus simplifying the development of entities. The proxy system call behavior depends on the type of the target.

Figure 3.4: Interface-based proxy entities

A ***proxiable object*** is an object that implements the `JistAPI.Proxiable` interface. This interface, like the `JistAPI.Entity` interface, is empty and serves only as a marker to the rewriter. The effect of this tag is to introduce the additional machinery for receiving events from the kernel. In this manner, the extra method call overhead of an intermediate relaying entity is eliminated, so this approach is always preferred over proxying regular objects, when simulator source code is available. It requires merely the addition of a proxiable tag to the target.

Finally, one can proxy an already existing ***entity***. An entity already includes all the machinery required to receive and to process events. Thus, the event delivery path is left unchanged. Nevertheless, the `proxy` call does serve an important function on the dispatch side. The result of the `proxy` call, in this and in all the previous cases, is a special separator object that relays only the events of the specified proxy interface. Thus, the system call serves to restrict events from a particular source, which is useful in larger simulations. The proxying generates a *capability* (in the systems sense), since it is unforgeable: the separator cannot be cast to other interfaces at runtime.

Proxy entities simplify development. They allow an object the flexibility of combining both event-driven and regular method invocations within the same class. They are interface-based, so they do not interfere with the object hierarchy. And, they allow for a capability-like isolation of functionality, which is useful in larger simulators. Finally, the internal mechanisms used for both event dispatch and delivery of events are different, but there is no visible difference from regular entities at the syntactic level nor any significant degradation in performance.

The code listing in Figure 3.5 shows how to use proxy entities with a basic example, similar to the earlier `hello` example. Note that `myEntity` is proxiable, because

proxy.java

```
1  import jist.runtime.JistAPI;
2
3  public class proxy
4  {
5    public static interface myInterface
6      extends JistAPI.Proxiable
7    {
8      void myEvent();
9    }
10
11   public static class myEntity implements myInterface
12   {
13     private myInterface proxy =
14       (myInterface)JistAPI.proxy(this, myInterface.class);
15     public myInterface getProxy() { return proxy; }
16
17     public void myEvent()
18     {
19       JistAPI.sleep(1);
20       proxy.myEvent();
21       System.out.println("myEvent at t="+JistAPI.getTime());
22     }
23   }
24
25   public static void main(String args[])
26   {
27     myInterface e = (new myEntity()).getProxy();
28     e.myEvent();
29   }
30 }
```

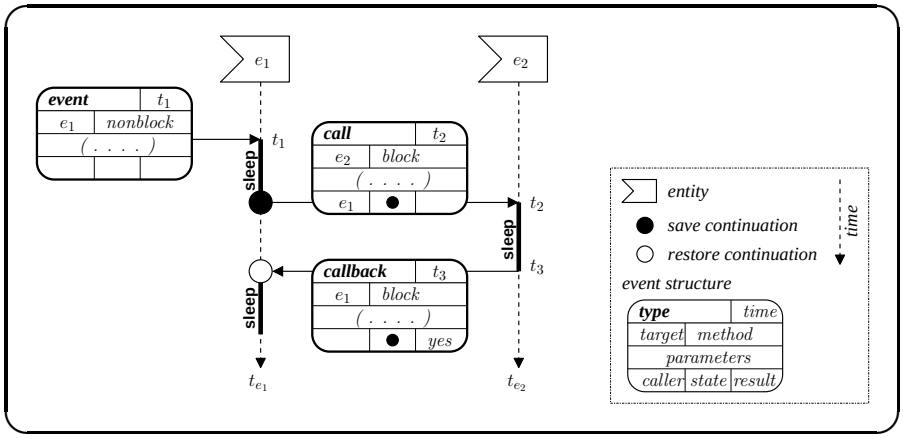
Figure 3.5: An example illustrating the use of proxy entities.

it implements the `myInterface` on line 11, which inherits `JistAPI.Proxiable` on line 6. The proxy separator is defined on line 14 using the `JistAPI.proxy` call with the target proxiable instance and appropriate interface class. The invocations on this proxy on lines 20 and 28 occur in simulation time.

3.5 Blocking invocation semantics

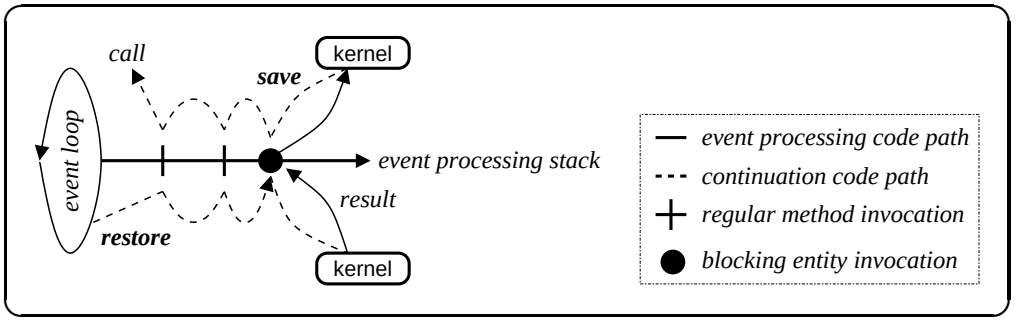
JiST conveniently models events as invocations on entities. This facility provides all the functionality of an explicit event queue, which is all that many existing simulators use. However, it remains cumbersome to model simulation *processes*, since they must be written as event-driven state machines. While many entities, such as network protocols or routing algorithms, naturally take this event-oriented form, other kinds of entities do not. For example, an entity that models a file-transfer is more readily encoded as a process than as a sequence of events. Specifically, one would rather use a tight loop around a `blocking_send` routine than dispatch `send_begin` events to some transport entity, which will eventually dispatch matching `send_complete` events in return. Many existing applications make use of system calls with blocking semantics. It is desirable to be able to run such applications within simulators. To that end, JiST supports blocking invocation semantics and *simulation time continuations*.

In order to invoke an entity method with continuation, the simulation developer merely declares that a given entity method is a *blocking* event. Blocking and non-blocking methods can co-exist within the same entity. Syntactically, an entity method is blocking, if and only if it declares that it throws a `JistAPI.Continuation` exception. This exception is not actually thrown and need not be explicitly handled by a caller. It acts merely as a tag to the rewriter,



Blocking events allow process-oriented development. When a blocking entity method is invoked, the continuation state of the current event is saved and attached to a call event. When this call event is complete, the kernel schedules a callback event to the caller. The continuation is restored and the caller continues its processing from where it left off, albeit at a later simulation time.

Figure 3.6: Blocking invocation semantics



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

Figure 3.7: Unrolling the Java stack.

preserved with the method signature through compilation to bytecode. Moreover, it need not be explicitly declared further up the call-chain, since it is a sub-class of `Error`, the root of an implicit exception hierarchy.

The semantics of a blocking entity method invocation, as shown in Figure 3.6, are a natural extension atop the existing event-based invocation. The kernel first saves the call-stack of the calling entity and attaches it to the outgoing event. When the call event is complete, the kernel notices that caller information was attached to it, and therefore dispatches a callback event to the caller, with its continuation information. Thus, when the callback event is eventually dequeued, the state is restored and the execution continues right after the point of the blocking entity method invocation. In the meantime, however, the local simulation time will have progressed to the simulation time at which the calling event was completed, and other events may have been processed against the entity.

This approach has a number of advantages. It allows blocking and non-blocking entity methods to co-exist, which allows a combination of event-oriented and process-oriented simulation. Methods can arbitrarily be tagged as blocking, and the basic event structures are extended to store the call and callback information. However, there is no notion of an explicit process, nor even a logical one. Unlike process-oriented simulation runtimes, which must pre-allocate fixed-size stacks for each real or logical process, the JiST stacks are variably-sized and allocated on demand. The stacks are allocated only at the point of the blocking entity invocation, and they exist on the heap along with the event structure that contains it. This dramatically reduces memory consumption. Moreover, the model is actually akin to threading, in that multiple continuations for processing within a single entity can exist concurrently. To the best of my knowledge, this is a first

for any simulation language. Finally, there is no system context-switch required. The concurrency occurs only in simulation time and the underlying events may be executed sequentially within a single thread of control.

The code listing in Figure 3.8 shows an entity with a single blocking method. Notice that `blocking` is a blocking method, since it declares that it may throw a `JistAPI.Continuation` exception on line 5. Otherwise, `blocking` would be a regular entity method invoked in simulation time. The effect of the blocking tag is best understood by comparing the output for the program both with and without the blocking semantics, i.e. both with and without the blocking tag on line 5. Note how the timestamps are affected, in addition to the ordering of the events.

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

Unfortunately, saving and restoring the Java call-stack for continuation is not a trivial task [96]. The fundamental difficulty arises from the fact that stack manipulations are not supported at either the language, library, or bytecode level. The JiST design draws and improves on the ideas in the JavaGoX [100] and the PicoThreads [14] projects, which also save the Java stack for entirely different reasons. The design eliminates the use of exceptions to carry state information. This is considerably more efficient for the purposes of simulation, since Java exceptions are expensive. The approach also eliminates the need to modify method signatures. This fact is significant, since it allows the JiST continuation capturing mechanism to function even across the standard Java libraries. In turn, this enables, for example, the execution of standard, unmodified Java network appli-

cont.java

```
1  import jist.runtime.JistAPI;
2
3  public class cont implements JistAPI.Entity
4  {
5    public void blocking() throws JistAPI.Continuation
6    {
7      System.out.println("called at t="+JistAPI.getTime());
8      JistAPI.sleep(1);
9    }
10
11   public static void main(String args[])
12   {
13     cont c = new cont();
14     for(int i=0; i<3; i++)
15     {
16       System.out.println("i="+i+" t="+JistAPI.getTime());
17       c.blocking();
18     }
19   }
20 }
```

Figure 3.8: An example illustrating the use of a blocking invocations.

cations within network simulators written atop JiST, as discussed in chapter 4. Briefly, a network socket operation is rewritten into a blocking method invocation, so that the application is "frozen" in simulation time until after the network operation processing is simulated.

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

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

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

Each method within the continuable set undergoes a basic CPS transformation, as shown in Figure 3.9. The rewriter scans the method for continuation points and assigns each one a *program location number*. It then performs a data-flow analysis of the method to determine the types of the local variables and stack slots at that point and uses this information to generate a custom class that will store the

Before CPS transform:

```
1  METHOD continuable:
2
3    instructions
4
5    invocation BLOCKING
6
7    more instructions
```

After CPS transform:

```
1  METHOD continuable:
2    if jist.isRestoreMode:
3      jist.popFrame f
4      switch f.pc:
5        case PC1:
6          restore f.locals
7          restore f.stack
8          goto PC1
9        ...
10
11   instructions
12
13   setPC f.pc, PC1
14   save f.locals
15   save f.stack
16 PC1:
17   invocation BLOCKING
18   if jist.isSaveMode:
19     jist.pushFrame f
20     return
21
22   more instructions
```

These pseudo-bytecode listings show the CPS transformation that occurs on all continuable methods. The transformation instruments the method to allow it to either (a) save and exit or (b) restore and start from any continuable invocation location.

Figure 3.9: CPS transformation on continuable program locations.

continuation frame of this program location. These classes, containing properly typed fields for each of the local variables and stack slots in the frame, will be linked together to form the preserved stack. Finally, the rewriter inserts both saving and restoration code for each continuation point. The saving code marshals the stack and locals into the custom frame object and pushes it onto the event continuation stack via the kernel. The restoration code does the opposite and then jumps right back to the point of the blocking invocation.

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

Finally, the kernel serves as the continuation trampoline, as shown in Figure 3.7. When the kernel receives a request to perform a call with continuation, it registers the call information, switches to save mode, and returns to the caller. The stack then unwinds, and eventually returns to the event loop, at which point the call event is dispatched with the continuation attached. When the call event is received, it is processed, and a callback event is dispatched in return with both the continuation and the result attached. Upon receiving this callback event, the kernel switches to restore mode and invokes the appropriate method. The stack then winds up to its prior state, and the kernel receives a request for a continuation call yet again. This time, the kernel simply returns the result (or re-throws the

exception) of the call event and allows the event processing to continue from where it left off.

The invocation time of a blocking event with this implementation is proportional to the length of the stack. The time to perform a blocking invocation with a short call stack is only around 2-3 times the dispatch time of a regular event. Thus, continuations present a viable, efficient way to reclaim process-oriented simulation functionality within an event-oriented simulation framework. Extensions to the Java libraries and virtual machine that expose the stack in a type-safe manner, as presented in [20], could completely eliminate the performance gap between non-blocking and blocking events.

3.6 Simulation time concurrency

Using only basic simulation events and continuations, a *cooperative* simulation time threading package has been constructed. It can be transparently used as non-preemptive replacement for the Java `Thread` class within existing Java applications to be simulated. *Pre-emptive* threading can also be supported, if necessary, by inserting simulation time context switch calls at appropriate code locations during the rewriting phase. However, since the primary objective is simulation throughput, cooperative concurrency is preferred.

Given simulation time concurrency, one may wish to recreate various simulation time synchronization primitives. As an example, I have constructed the channel primitive from Hoare's Communicating Sequential Processes (CSP) language [51]. It has been shown that other synchronization primitives, such as locks, semaphores, barriers, monitors, and FIFOs, can readily be built using such channels. Or, these primitives can be implemented directly within the kernel. As shown

in Figure 3.10, the CSP channel blocks on the first receive (or send) call and stores the continuation. When the matching send (or receive) arrives, then the data item is transferred across the channel and control returns to both callers. In other words, two simulation events are scheduled with the appropriate continuations. A JiST channel is created via the `createChannel` system call. It supports both CSP semantics as well as non-blocking sends and receives. JiST channels are used, for example, within the SWANS implementation of TCP sockets in order to block a Java application when it calls `receive` and to send information back to the application when a packet arrives. In this case, the channel mimics the traditional boundary between the user-level network application and the in-kernel network stack implementation.

3.7 Parallel, optimistic, and distributed execution

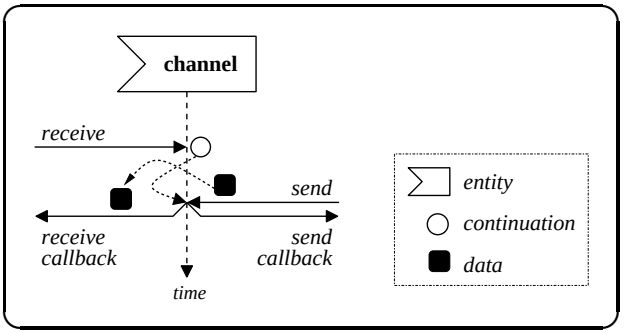
The JiST system, as described thus far, is capable of executing simulations sequentially and it does so with performance that exceeds existing, highly optimized simulation engine instances. JiST also supports *inter*-simulation concurrency. Any number of JiST engines can be started on separate machines, each capable of accepting simulation jobs from a central JiST job queue, where JiST clients post simulations. As each job is processed on the next available server, the corresponding client will service remote class loading requests and receive redirected output and simulation logs. This naïve approach to parallelism has proven sufficient at present, since JiST can already model very large networks on individual machines, and it provides perfect speed-up for batches of simulations. JiST, however, was explicitly designed to allow concurrent, distributed and speculative execution, or *intra*-simulation parallelism. By modifying the simulation time kernel, unmodified

simulations can be executed over a more powerful computing base. These kernels have not been implemented, but many hooks are already in place for such extensions, and they can be implemented transparently with respect to existing simulation programs.

Parallel execution in JiST may be achieved by dividing the simulation time kernel into multiple threads of execution, called `Controller`s, which already exist in the single-threaded implementation. Each controller owns and processes the events of a subset of the entities in the system, as shown in Figure 3.11. The controllers synchronize with one another in order to bound their, otherwise, independent forward progress.

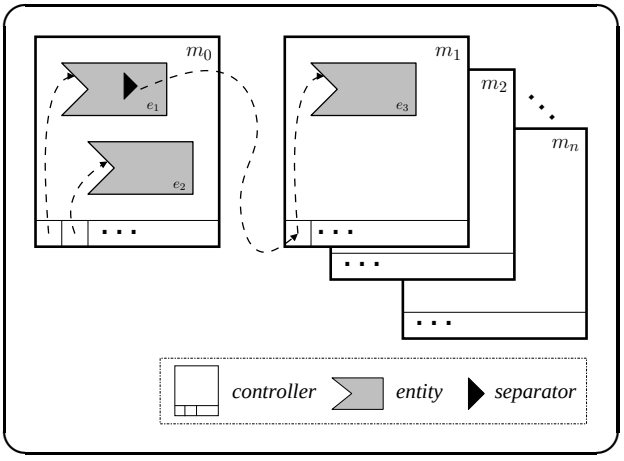
JiST can further be extended to transparently support entity rollback, so that simulation time synchronization protocols among the various controllers need not be conservative. State checkpoints can be automatically taken through object cloning. Alternatively, efficient undo operators can be statically generated through code inspection in some cases or possibly provided by the developer in other cases for added performance. In any case, entity state changes can always be dynamically intercepted either at the level of complete entities, individual objects, or even individual fields within an object. These state changes can be logged for possible undo, allowing the JiST kernel to transparently perform speculative execution of the simulation.

Controllers may also be distributed in order to run simulations across a cluster of machines. Conveniently, Java support for remote method invocation (or more efficient drop-in alternatives such as KaRMI [84]) combined with automatic object serialization provides location transparency among distributed controllers. Even beyond this, the existing separator objects (which replace entity references during



A blocking CSP channel is built using continuations. SWANS uses JiST channels to create simulated sockets with blocking semantics. Other simulation time synchronization primitives can similarly be constructed.

Figure 3.10: Simulation time CSP Channels.



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

Figure 3.11: Partitioning simulation entities among Controllers.

rewriting) allow entities to dynamically be moved among controllers in the system, for balancing load or for minimizing invocation latency and network bandwidth, while the simulation is running. The automatic insertion of separators between entities provides the simulation developer with a convenient single system image abstraction of the cluster.

3.8 Simulation research platform

In addition to being a practical and efficient simulation tool for building simulators, the JiST abstraction provides a convenient research platform for the exploration of new simulation techniques. Primarily, this is due to the flexibility afforded by the intermediate Java bytecode representation of simulations.

I have just described, for example, how to convert *simulation time* programs at the bytecode level to run in parallel using either conservative synchronization algorithms [27, 28, 55] or speculative global virtual time [53, 38] algorithms. The bytecode 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 candidates from the more recent simulation research exist. Static code analysis techniques can be used to produce efficient entity checkpointing as in [41]. Alternatively, reverse computations can be produced for entity rollback as in [26]. The JiST design can simultaneously support different synchronization protocols for different entities by tagging them with different interfaces as in [68]. 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 [19]. Research into each of these ideas would benefit from the availability of pre-existing simulations

and, as with JiST, from the ability to reuse a standard language compiler and its highly optimized, robust runtime.

3.9 Summary

In this chapter, a number of extensions to the basic model have been presented, highlighting the advantages of the JiST design and its inherent flexibility. I have discussed timeless objects, reflection-based scripting, tight event coupling, proxy entities, blocking events, simulation time concurrency, concurrent and distributed simulation kernels, and other ideas from the simulation research literature that could be incorporated into the JiST rewriting phase. The table below summarizes the benefits of the virtual machine-based approach to simulation.

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

language-oriented benefits

reflection	-	allows script-based simulation configuration, debugging and tracing in a manner that is transparent to the simulation implementation
safety	-	allows for an object-level isolation of state between entities; ensures that all simulation time calls pass through the kernel; allows sharing of immutable objects; provides flexibility in entity state aggregation
rewriting	-	operates at the bytecode level; does not require source-code access; allows cross-cutting program transformations, guided by program analysis and annotations
Java	-	JiST reuses the standard language, libraries, compiler, and runtime

system-oriented benefits

inter-process communication	-	since co-located entities share the same process space, messages are passed by reference and there is no serialization; there is also no context switch required
language-based kernel	-	permits cross-layer optimization between kernel and running simulation for faster system calls and event dispatch
robustness	-	strict Java verification ensures that simulations will not crash, but, at worst, fail by exception

garbage collection	-	memory for objects with long and variable lifetimes, such as network packets, is automatically managed; facilitates memory savings through sharing of immutable objects; avoids memory leaks for long-running simulations and facilitates sophisticated memory management protocols
concurrency	-	simulation object model and execution semantics support parallel and optimistic execution transparently with respect to the application
distribution	-	provides a single system image abstraction that allows for dynamic entity migration to balance processor, memory or network load

hardware-oriented benefits

portability	-	pure Java; "runs everywhere"
cost	-	runs on COTS clusters (NOW, grid, etc.), as well as more specialized architectures

Chapter 4

Building a Scalable Network Simulator

As a validation of the virtual machine-based approach, I have constructed SWANS, a **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator, atop the JiST platform. Note, however, that JiST is a general-purpose discrete event simulation engine, and that nothing within its design is specific to the simulation of wireless networks. This application domain was selected primarily because it would produce a useful research tool: existing wireless network simulation tools are not sufficient for current research needs. Other related and applicable areas of recent research interest include overlay networks [5] and peer-to-peer applications [102, 112, 94, 89].

4.1 Background

Wireless networking research is fundamentally dependent upon simulation. Analytically quantifying the performance and complex behavior of even simple protocols on a large scale 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 majority of publications in this area are based entirely upon simulation.

At a minimum, one would like to simulate networks of many thousands of nodes. However, even though a few parallel discrete event simulation environments have been shown to scale to networks of beyond 10^4 nodes, slow sequential network sim-

ulators remain the norm [92]. In particular, most published ad hoc network results are based on simulations of few nodes only (usually fewer than 500 nodes), for a short duration, and over a small geographical area. Larger simulations usually compromise on simulation detail. For example, some existing simulators simulate only at the packet level without considering the effects of signal interference. Others reduce the complexity of the simulation by curtailing the simulation duration, reducing the node density, or restricting mobility. My goal in building SWANS is to improve the state of the art in this regard.

The two most popular simulators in the wireless networking space are ns2 and GloMoSim. The ns2 network simulator [70] has a long history with the networking community, is widely trusted, and has been extended to support mobility and wireless networking protocols. It is built as a monolithic, sequential simulator, in the *library*-systems simulator design. 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 [93]. However, this technique has proved beneficial primarily 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. Recently, simulation researchers have shown ns2 to scale, with difficulty and substantial hardware resources, to simulations of a few thousand nodes [92].

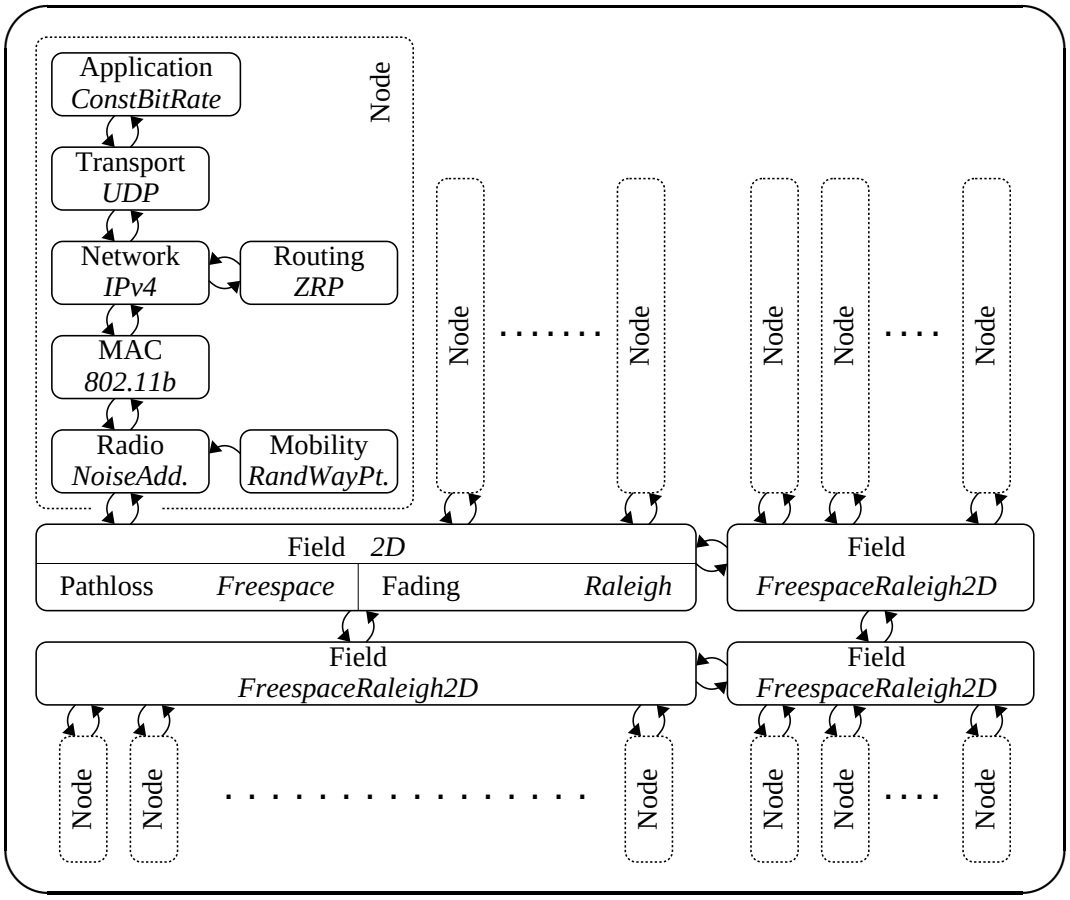
GloMoSim [111] is a newer simulator written in Parsec [11], a highly-optimized C-like simulation language. GloMoSim has recently gained popularity within the wireless ad hoc networking community. It was designed specifically for scalable

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 the state of multiple simulation nodes are multiplexed within a single Parsec entity. While this effectively reduces memory consumption, it incurs a performance overhead and also increases code complexity. Unfortunately, the aggregation of state also renders speculative execution techniques impractical, as has been noted by the authors. GloMoSim has been shown to scale to 10,000 nodes on large, specialized multi-processor machines.

4.2 Design highlights

The SWANS software is organized as independent software components that can be composed to form complete wireless simulations, as shown in Figure 4.1. Its capabilities are similar to ns2 [70] and GloMoSim [111], two popular wireless network simulators. 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 models. Instances of each component type are shown italicized in the figure. A more detailed description of the various SWANS components can be found in Appendix B.

Notably, the development of SWANS has been relatively painless. Since JiST inter-entity event creation and delivery is implicit, as well as event garbage collection and typing, the code is compact and intuitive. Components in JiST consume less than half of the code (in uncommented line counts) of comparable compo-



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.

Figure 4.1: The SWANS component architecture.

nents in GloMoSim, which are already significantly smaller than their counterpart implementations in ns2.

Every SWANS component is encapsulated as a JiST entity: it stores it own local state and interacts with other components via their exposed event interfaces. SWANS contains components for constructing a node stack, as well components for a variety of mobility models and field configurations. This pattern simplifies simulation development by reducing the problem to one of creating relatively small, event-driven components. And, unlike the design of ns2 and GloMoSim, it explicitly partitions the simulation state and the degree of inter-dependence between components, allows components to be readily interchanged with suitable alternate implementations of the common interfaces, and for each simulated node to be independently configured. Finally, it also confines the simulation communication pattern. For example, `Application` or `Routing` components of different nodes cannot 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 in a distributed simulation, the simulated field may be partitioned into non-overlapping, cooperating `Field` entities along a grid.

It is important to note that, in JiST, communication among entities is very efficient. The design incurs no serialization, copy, or context-switching cost among co-located entities, since the Java objects contained within events are passed along by reference via the simulation time kernel. Simulated network packets are actu-

ally a chain of nested objects that mimic the chain of packet headers added by the network stack. Moreover, since the packets are timeless by design, a single broadcasted packet can be safely shared among all the receiving nodes and the very same object sent by an `Application` entity on one node will be received at the `Application` entity of another node. Similarly, if one uses TCP in the node stack, then the very same object will also be referenced in the sending node's TCP retransmit buffer. This design conserves memory, which, in turn, allows for the simulation of larger network models.

Dynamically created objects such as packets can traverse many different control paths within the simulator and can have highly variable lifetimes. The accounting for when to free unused packets is handled entirely by the garbage collector. This not only greatly simplifies the memory management protocol for packets, but also eliminates a common source of memory leaks that can accumulate over long simulation runs.

The partitioning of node functionality into individual, fine-grained entities provides an additional degree of flexibility for distributed simulations. The entities can be *vertically* aggregated, as in GloMoSim, which allows communication along a network stack *within* a node to occur more efficiently. However, the entities can also be *horizontally* aggregated to allow communication *across* nodes to occur more efficiently. In JiST, this reconfiguration can happen without any change to the entities themselves. The distribution of entities across physical hosts running the simulation can be changed dynamically in response to simulation communication patterns and it does not need to be homogeneous.

4.3 Embedding Java-based network applications

SWANS has a unique and important advantage over existing network simulators. It can run regular, unmodified Java network applications, such as web servers, peer-to-peer applications, and application-level multicast protocols, over a simulated network. These applications do not merely send packets to the simulator from other processes. They operate in simulation time within the same JiST process space, allowing far greater scalability. As another example, one could perform a similar transformation on Java-based database engines or file-system applications to model their I/O performance within a disk simulator.

This integration is achieved via a special `AppJava` SWANS application entity designed to be a harness for Java networking applications, inserting an additional rewriting phase into the JiST kernel that substitutes SWANS socket implementations for any of the Java counterparts found within the application. These SWANS sockets have identical semantics, but send packets through the simulated network. Specifically, the input and output methods are still blocking events (see section 3.5). And, to support these blocking semantics, JiST automatically modifies the necessary application code into continuation-passing style, allowing the application to operate within the event-oriented simulation time environment.

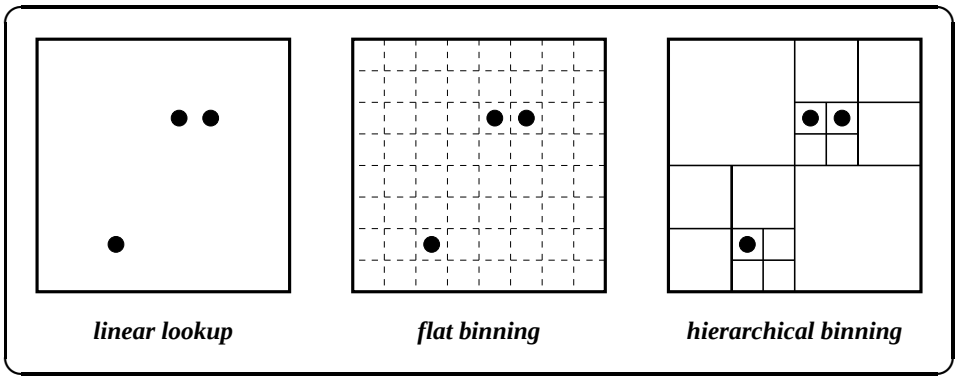
4.4 Efficient signal propagation using hierarchical binning

Finally, modeling signal propagation within the wireless network is strictly an application-level issue, unrelated to JiST performance. However, doing so efficiently is essential for scalable wireless simulation. When a simulated radio entity transmits a signal, the SWANS `Field` entity must deliver that signal to all radios

that could be affected, after considering fading, gain, and path loss. Some small subset of the radios on the field will be within reception range and a few more radios will be affected by the interference above some sensitivity threshold. The remaining majority of the radios will not be tangibly affected by the transmission.

ns2 and GloMoSim implement a naïve signal propagation algorithm, which uses a slow, $O(n)$, linear search through *all* the radios to determine the node set within the reception neighborhood of the transmitter. This clearly does not scale as the number of radios increases. ns2 has recently been improved with a grid-based algorithm [73]. I have implemented both of these in SWANS. In addition, SWANS has a new, more efficient algorithm that uses *hierarchical* binning. The spatial partitioning imposed by each of these data structures is depicted in Figure 4.2.

In the grid-based or flat binning approach, the field is sub-divided into a grid of node bins. A node location update requires constant time, since the bins divide the field in a regular manner. The neighborhood search is then performed by scanning all bins within a given distance from the signal source. While this operation is also of constant time, given a sufficiently fine grid, the constant is very sensitive to the chosen bin size: bin sizes that are too large will capture too many nodes and thus not serve their search-pruning purpose; bin sizes that are too small will require the scanning of many empty bins, especially at lower node densities. A reasonable bin size is one that captures a small number of nodes per bin. Thus, the bin size is a function of the local radio density and the signal propagation radius. However, these parameters may change in different parts of the field, from radio to radio, and even as a function of time as in the case of power-controlled transmissions, for example.



Efficient signal propagation is critical for wireless network simulation performance. Hierarchical binning of radios on the field allows location updates to be performed in expected amortized constant time and the set of receiving radios to be computed in time proportional to its size.

Figure 4.2: Alternative spatial data structures for radio signal propagation.

Hierarchical binning improves on the flat binning approach. Instead of a flat sub-division, the hierarchical binning implementation recursively divides the field along both the x and y-axes. The node bins are the leaves of this balanced, spatial decomposition tree, which is of height equal to the number of divisions, or $log_4(\frac{field\ size}{bin\ size})$. The structure is similar to a quad-tree, except that the division points are not the nodes themselves, but rather fixed coordinates. Note that the height of the tree changes only logarithmically with changes in the bin or field size. Furthermore, since nodes move only a short distance between updates, the expected amortized height of the common parent of the two affected node bins is $O(1)$. This, of course, is under the assumption of a reasonable node mobility that keeps the nodes uniformly distributed. Thus, the amortized cost of updating a node location is constant, including the maintenance of inner node counts. When scanning for node neighbors, empty bins can be pruned during the spatial descent. Thus, the set of receiving radios can be computed in time proportional to the number of receiving radios. Since, at a minimum, SWANS will need to simulate delivery of the signal at each simulated radio, the algorithm is asymptotically as efficient as scanning a cached result, as proposed in [21], even assuming perfect caching. But, the memory overhead of hierarchical binning is minimal. Asymptotically, it amounts to $\lim_{n\to\infty} \sum_{i=1}^{log_4 n} \frac{n}{4^i} = \frac{n}{3}$. The memory overhead for function caching is also $O(n)$, but with a much larger constant. Furthermore, unlike the cases of flat binning or function caching, the memory accesses for hierarchical binning are tree structured and thus exhibit better locality.

4.5 Summary

The SWANS simulator runs over JiST, combining the traditional systems-based (e.g., ns2) and languages-based (e.g., GloMoSim) approaches to simulation construction. SWANS is able to simulate much larger networks and has a number of other advantages over existing tools. The JiST design is leveraged within SWANS to:

- *achieve high simulation throughput*: Simulation events among the various entities, such as packet transmissions, are performed with no memory copy and no context switch. The system also continuously profiles running simulations and dynamically performs code inlining, constant propagation and other important optimizations, even across entity boundaries. This is important, because many stable simulation parameters are not known until the simulation is running. Greater than 10× speedups have been observed.

- *save memory*: Memory is critical for simulation scalability. Automatic garbage collection of events and entity state not only improves robustness of long-running simulations by preventing memory leaks, it also saves memory by facilitating more sophisticated memory protocols. For example, network packets are modeled as immutable objects, allowing a single copy to be shared across multiple nodes. This saves the memory (and time) of multiple packet copies on every transmission. A different example of memory savings in SWANS is the use of soft references for storing cached computations, such as routing tables. These routing tables can be automatically collected, as necessary, to free up memory.

- *run standard Java applications*: SWANS can run existing Java network applications over the simulated network without modification. The network application is automatically transformed to use simulated sockets and then into a continuation-passing style around blocking socket operations. The original network applications are run within the same process as SWANS and JiST, which increases scalability by eliminating the considerable overhead of process-based isolation.

In addition to the simulator design, it is also essential to model wireless signal propagation efficiently at the application level, because this computation is performed on every packet transmission. The hierarchical binning data structure is an improvement over the traditional flat binning approach that allows node location updates to be performed in expected amortized constant time and the receiver node set to be computed in time proportional to the number of receivers.

The combination of these attributes leads to a surprisingly efficient network simulator. A performance evaluation of JiST and SWANS follows in the next chapter.

Chapter 5

JiST and SWANS Performance

Conventional wisdom regarding language performance [9] argues against implementing a simulation platform in Java. In fact, the vast majority of existing simulation systems have been written in C and C++, or their derivatives. Nevertheless, the results in this chapter show that JiST and SWANS perform surprisingly well: aggressive profile-driven optimizations combined with the latest Java runtimes result in a high-performance simulation system. SWANS is compared with the two most popular ad hoc network simulators: ns2 and GloMoSim. These were selected because they are widely used, freely available sequential network simulators and designed in the systems-based and language-based approaches, respectively. Both macro-benchmark results, running full SWANS simulations, as well as micro-benchmark results, highlighting the throughput and memory advantages of JiST, are shown.

Unless otherwise noted, the following measurements were taken on a 2.0 GHz Intel Pentium 4 single-processor machine with 512 MB of RAM and 512 KB of L2 cache, running the version 2.4.20 stock Redhat 9 Linux kernel with glibc v2.3. I used the publicly available versions of Java 2 JDK (v1.4.2), Parsec (v1.1.1), GloMoSim (v2.03), and ns2 (v2.26). Each data point presented represents an average of at least five runs for the shorter time measurements. All tests were also performed on a second machine – a more powerful and memory rich dual-processor – giving identical absolute memory results and relative results for throughput (i.e., scaled with respect to processor speed).

5.1 Macro-benchmarks

In the first experiment, JiST is benchmarked running a full SWANS ad hoc wireless network simulation. SWANS was configured to simulate an ad hoc network of nodes running a UDP-based beaconing node discovery protocol (NDP) application. Node discovery protocols are an integral component of many ad hoc network protocols and applications [46, 57]. Also, this experiment is representative both in terms of both code coverage and network traffic: it utilizes the entire network stack and transmits over every link in the network every few seconds. However, the experiment is still simple enough to provide high confidence of simulating *exactly* the same operations across the different platforms (SWANS, GloMoSim and ns2), which permits comparison and is difficult to achieve with more complex protocols. Finally, the simulation results were validated using analytical estimates.

The following identical scenario was constructed in each of the simulation platforms. The application at each node maintains a local neighbor table and beacons every 2 to 5 seconds, chosen from a uniform random distribution. Each wireless node is placed randomly in the network coverage area and moves with random-waypoint mobility [57] at speeds of 2 to 10 meters per second selected at random and with pause times of 30 seconds. Mobility in ns2 was turned off, because the pre-computed trajectories resulted in excessively long configuration times and memory consumption. Each node is equipped with a standard radio configured with typical 802.11b signal strength parameters. The simulator accounts for free-space path loss with ground reflection and Rayleigh fading. I ran simulations with widely varying numbers of nodes, keeping the node density constant, such that each node transmission is received, on average, by 4 to 5 nodes and interferes with

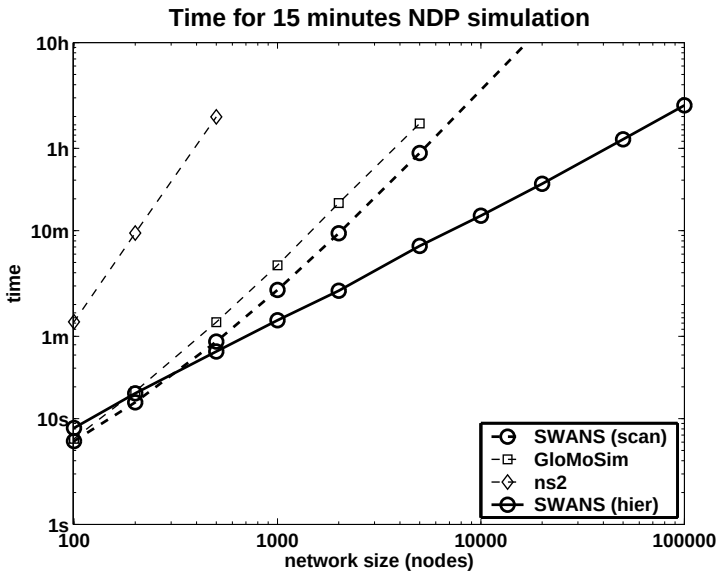
approximately 35 others. Above each radio, there is a stack of 802.11b MAC, IPv4 network, UDP transport, and NDP application entities.

This network model was run for 15 simulated minutes, measuring the overall memory and time required. The memory measurements include the base process memory, the memory overhead for simulation entities, and all the simulation data at the beginning of the simulation. The time measurements include the simulation setup time, the event processing overheads, and the application processing time.

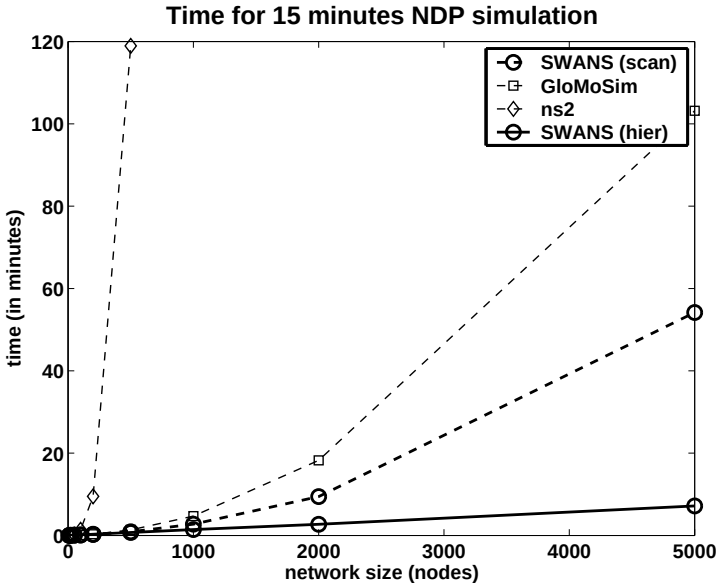
The throughput results are plotted both on log-log and on linear scales in Figure 5.1. As expected, the simulation times are quadratic functions of n, the number of nodes, when using the naïve signal propagation algorithm. Even without node mobility, ns2 is highly inefficient. SWANS outperforms GloMoSim by a factor of 2. SWANS-hier uses the improved hierarchical binning algorithm to perform signal propagation instead of scanning through all the radios. As expected, SWANS-hier scales linearly with the number of nodes.

The memory footprint results are plotted in Figure 5.2 on log-log scale. JiST is more efficient than GloMoSim and ns2 by almost an order and two orders of magnitude, respectively. This allows SWANS to simulate much larger networks. The memory overhead of hierarchical binning is asymptotically negligible. Selected data points are tabulated in Table 5.1.

Next, SWANS was tested with some very large networks simulations, using a dual-processor 2.2GHz Intel Xeon machine (though only one processor was used) with 2GB of RAM running Windows 2003. The results are plotted in Figure 5.3 on a log-log scale. Both the naïve propagation algorithm and hierarchical binning are shown. One can observe linear behavior for the latter in all simulations up to networks of one million nodes. The 10^6 node simulation consumed just less than



(a) log-log scale



(b) linear scale

Figure 5.1: SWANS significantly outperforms both ns2 and GloMoSim in simulations of the node discovery protocol.

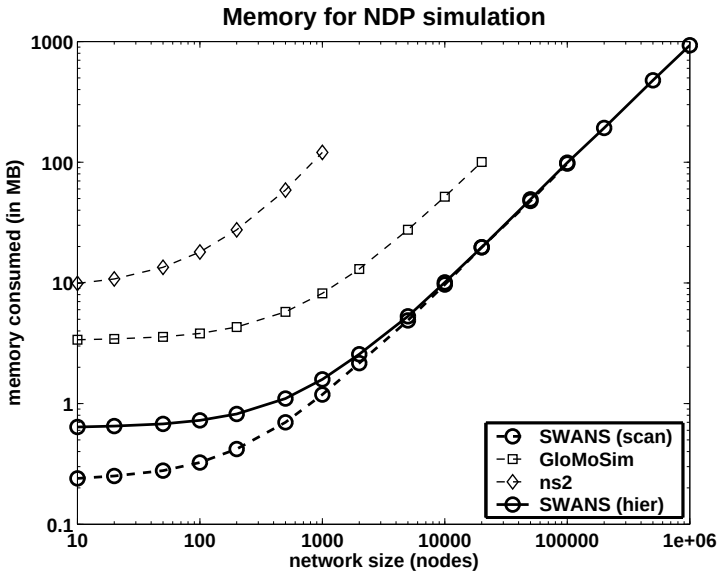


Figure 5.2: SWANS can simulate larger network models due to its more efficient use of memory.

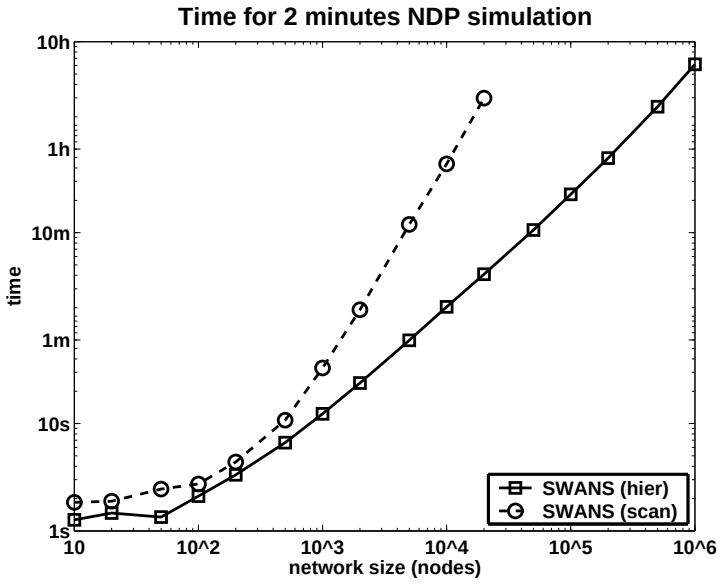


Figure 5.3: SWANS scales to networks of 10^6 wireless nodes. The figure shows the time for a sequential simulation of a node discovery protocol in a wireless ad hoc network running on a commodity machine.

Table 5.1: Select data points showing SWANS time and memory performance, relative to GloMoSim and ns2, running node discovery protocol simulations.

nodes	simulator	time	memory
500	**SWANS**	**54 s**	**700 KB**
	GloMoSim	82 s	5759 KB
	ns2	7136 s	58761 KB
	SWANS-hier	*43 s*	*1101 KB*
5,000	**SWANS**	**3250 s**	**4887 KB**
	GloMoSim	6191 s	27570 KB
	SWANS-hier	*430 s*	*5284 KB*
50,000	**SWANS**	**312019 s**	**47717 KB**
	SWANS-hier	*4377 s*	*49262 KB*

1GB of memory on initial configuration, ran with an average footprint of 1.2GB (fluctuating due to delayed garbage collection), and completed within $5\frac{1}{2}$ hours. This exceeds previous ns2 and GloMoSim results by two orders of magnitude, using the same commodity hardware.

Finally, SWANS was tested with some large network simulations of an actual routing protocol, ZRP. The ZRP entity in each node stack stores information about neighbors, zone-wide link state, and routes. This clearly increases the per-node memory requirements. Figure 5.4 shows the time and memory required to simulate different size networks at a fixed density of 10 neighbors per node. It is not clear that such a large flat ad hoc network is meaningful, except to exhibit SWANS scalability. However, one could certainly simulate many smaller, connected flat ad hoc networks with comparable workload. The memory requirements grow linearly with the size of the network. The time required grows only slightly faster than linearly due to garbage collection overhead (using default GC parameters).

During the bordercast evaluation performed for chapter 6, it certainly helped to have an average simulation running time well below 15 minutes across the 4000+ simulations that were needed. Using a pool of between 10 to 40 desktop machines, coordinated using a poor-man's prioritized batch queuing system, allowed for rapid and continuous collection of simulation results.

5.2 Event throughput

Having presented macro-benchmark results, the emphasis will now shift to micro-benchmarks: event processing throughput, memory consumption for both simulation entities and events, and the context-switching overhead. High event throughput is essential for scalable discrete event simulation. Thus, in the following micro-

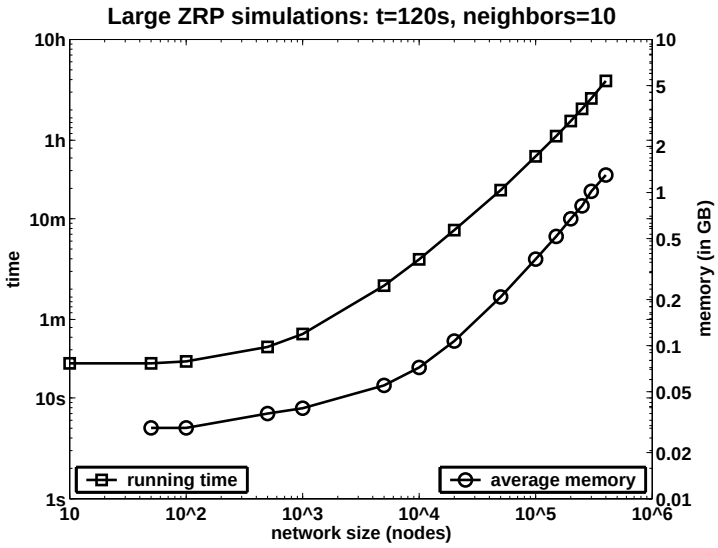


Figure 5.4: SWANS simulates up to 500,000 nodes at a density of 10 neighbors per node, each running ZRP, on a 2.8GHz machine with 4GB of memory.

Table 5.2: Time and memory to run ZRP simulations in SWANS.

nodes	**time**	**avg.memory**	**max.memory**
10,000	3m57s	72MB	94MB
100,000	41m36s	367MB	476MB
500,000	4h50m	1.63GB	1.9GB

benchmark, the performance of each of the simulation engines was measured in performing a tight simulation event loop. The simulations begin at time zero, with an event scheduled to do nothing but schedule another identical event in the subsequent simulation time step. Each simulation runs for n simulation time quanta, over a wide range of n, measuring the actual time elapsed. Note that, in performing these no-op events, any variations due to application-specific processing are eliminated, and one is able to observe just the *overhead* of the underlying event processing for each of the simulator designs.

Equivalent and efficient benchmark programs were written in each of the systems. The implementation of each is briefly described. The JiST program looks much like 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 program considers the overhead of the node aggregation mechanism built over Parsec. It is implemented as an application component that sends messages to itself. Both the Parsec and GloMoSim tests are compiled using `pcc -O3`, the most optimized Parsec compiler 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 node mobility, that depend on Tcl along the critical path. Consequently, I ran two tests. The ns2-C and ns2-Tcl tests correspond to events scheduled from either of the languages. ns2 simulation performance lies somewhere between these two widely divergent values, dependent on how frequently each language is employed during a given simulation. Finally, the reference test provides a computational lower bound. It is a program, written in C and com-

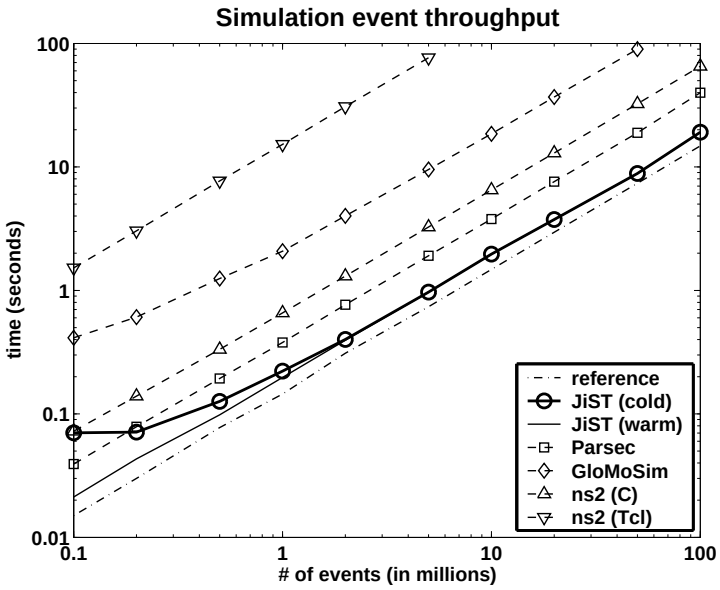
piled with `gcc -O3`, that merely inserts and removes elements from an efficient implementation of an array-based priority queue. Program listings are provided in Appendix C.

The results are plotted in Figure 5.5, as log-log and linear scale plots. As expected, all the simulations run in time linear with respect to the number of events, n. A counter-intuitive result is that JiST out-performs all the other systems, including the compiled ones. It also comes within 30% of the reference measure of the computational lower bound, even though it is written in Java. This achievement is clearly due to the impressive JIT dynamic compilation and optimization capabilities of the modern Java runtime. Furthermore, these optimizations can actually be seen as a kink in the JiST curve during the first fraction of a second of simulation. To confirm this, JiST was warmed with 10^6 events (or, for two tenths of a second) and the kink disappears. The linear-scale plot shows that the time spent on dynamic optimizations is negligible.

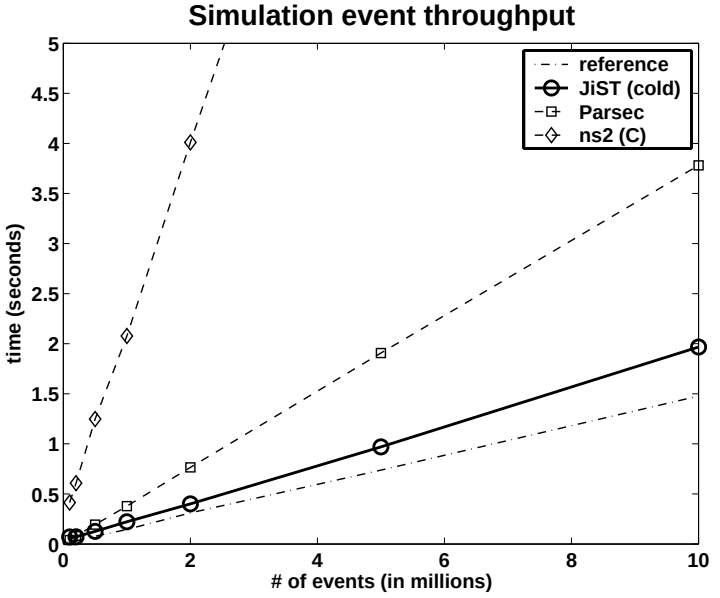
Table 5.3 shows the time taken to perform 5 million events in each of the measured simulation systems and also those figures normalized against both the reference program and JiST performance. JiST is twice as fast as both Parsec and ns2-C. GloMoSim and ns2-Tcl are one and two orders of magnitude slower, respectively.

5.3 Context switching

Alongside event throughput, it is important to ensure that inter-entity message passing and context switching 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.



(a) log-log scale



(b) linear scale

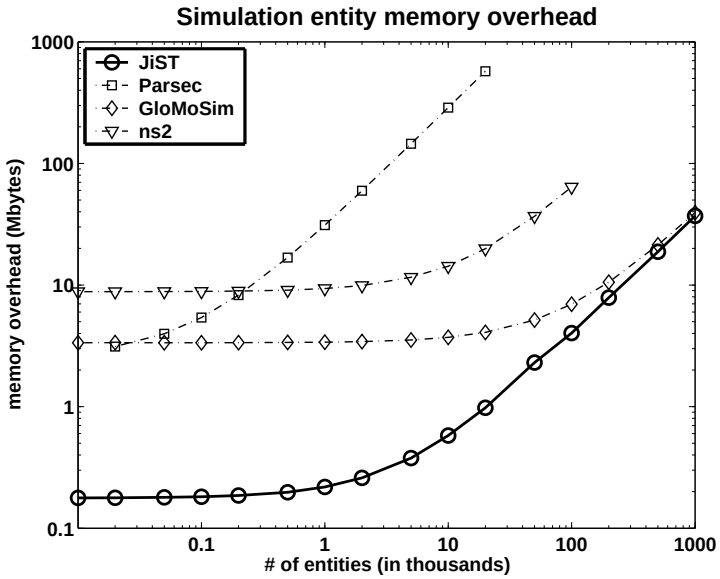
Figure 5.5: JiST has higher event throughput and comes within 30% of the reference lower bound program. The kink in the JiST curve in the first fraction of a second of simulation is evidence of JIT compilation and optimization at work.

The systems presented do not exhibit this problem. ns2 models events in single global event list. Parsec, and therefore also GloMoSim, models entities using logical processes implemented in user space and uses an efficient simulation time scheduler. JiST implements entities as concurrent objects and also uses an efficient simulation time scheduler. The context switching overheads of both Parsec and JiST were empirically measured. They are negligible and do not represent a scalability constraint on the number of entities in the simulation.

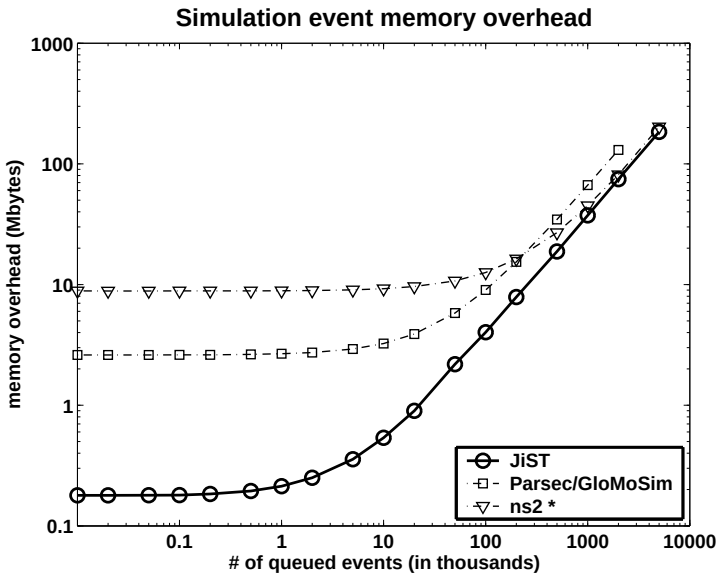
5.4 Memory utilization

Another important resource that may limit scalability is memory. In many simulations, memory is the critical scalability-limiting factor, since it establishes an upper bound on the size of the simulated model. The following experiments measure the memory consumed by simulation entities and by queued simulation events in each of the systems. Measuring the memory usage of entities involves the allocation of n empty entities and observing the size of the operating system process over a wide range of n. Similarly, a large number of events are queued to measure per event memory requirements. In the case of Java, a garbage collection sweep is performed before requesting an internal memory count. Note also that this benchmark, as before, measures the memory *overhead* imposed by the simulation system. The entities and events are empty. They do not carry any application data.

The entity and event memory results are plotted on log-log scales in Figure 5.6. 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. (a) – JiST performs well with respect to memory requirements for simulation entities. It performs comparably with GloMoSim, which uses node



(a) Memory per entity, log-log scale



(b) Memory per event, log-log scale

Figure 5.6: JiST allocates entities efficiently: comparable to GloMoSim at 36 bytes per entity, and over an order of magnitude less that Parsec or ns2. 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.

Table 5.3: Time to perform 5 million events, normalized against both the baseline and JiST.

5×10^6 events	time (sec)	vs. reference	vs. JiST
reference	0.738	1.00x	0.76x
JiST	**0.970**	**1.31x**	**1.00x**
Parsec	1.907	2.59x	1.97x
ns2-C	3.260	4.42x	3.36x
GloMoSim	9.539	12.93x	9.84x
ns2-Tcl	76.558	103.81x	78.97x

Table 5.4: Per entity and per event memory *overhead* along with the system memory overhead for a simulation scenario of 10,000 nodes, i.e. *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.

memory	entity	event	10K nodes sim.
JiST	**36 B**	**36 B**	**21 MB**
GloMoSim	36 B	64 B	35 MB
ns2	544 B	40 B*	74 MB*
Parsec	28536 B	64 B	2885 MB

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 priority queue. In contrast, each Parsec entity contains its own program counter and logical process stack[1]. In ns2, the benchmark program allocates the smallest split object possible, an instance of `TclObject`, responsible for binding values across the C and Tcl memory spaces. JiST provides the same dynamic configuration capability without requiring the memory overhead of split objects (see section 3.2). (b) – JiST also performs well with respect to event memory requirements. Though they store slightly different data, the C-based ns2 event objects are approximately 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 its event scheduling algorithm. This could not be validated, since source code was not available.

The memory requirements per entity, mem_{entity}, and per event, mem_{event}, in each of the systems are tabulated in Table 5.4. The table also shows the memory overhead 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, $mem_{sim} = 10^4 \times (10 \times mem_{entity} + 50 \times mem_{event})$.

Note that these figures do not include the fixed memory base for the process nor the actual simulation data, thus showing the *overhead* imposed by each approach. 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

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

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 1 GB process [93, 76].

5.5 Performance summary

JiST out-performs ns2, GloMoSim, and Parsec both in time and space. Each system is discussed, in turn, below. Table 5.5 summarizes the important design decisions in each of the systems that bear most significantly on these performance results.

Parsec runs very quickly and there are a number of reasons for this. It is compiled, not interpreted, and uses a modified `gcc` compiler to produce highly optimized binaries. It also uses non-preemptive logical processes to avoid system switching overhead: a Parsec logical context switch is implemented efficiently using only a `setjmp` and a stack pointer update, like a user-level thread switch. 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 Parsec per entity overhead by inserting a level of indirection into 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. It also eliminates the transparency and many of the advantages inherent to a language-based approach. For example, the aggregation of state renders speculative execution techniques impractical.

ns2 is a monolithic, sequential engine, so message queuing and dispatch are 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 executes quickly, but the Tcl-based functionality is almost two orders of magnitude slower. Additionally, both ns2 and GloMoSim suffer performance loss from their approaches to simulation configuration that eliminates opportunities for static optimizations, as discussed in section 3.2.

JiST uses a concurrent object model of execution and thus does not require node aggregation. Since entities are objects all within the same heap, as opposed to isolated processes, the memory footprint is small. There is also no context switching overhead on a per event basis and dynamic Java bytecode compilation and optimization result in high computational throughput. Dynamic optimizations can even inline simulation code into the language-based kernel. Since Java is a dynamic language, JiST does not require a split object model for configuration. Instead, one can use reflection to directly observe or modify the same objects used to run the simulation. This both eliminates the performance gap and the additional memory requirements.

5.6 Rewriting and annotation overhead

In addition to runtime performance, it is important to evaluate the rewriter itself, which transforms the various JiST tags and instructions embedded within the compiled simulation program into code with the appropriate simulation time semantics. For ease-of-use, the JiST rewriter is implemented as a dynamic class loader. It uses the Byte-Code Engineering Library [31] to automatically modify the simulation program bytecode as it is loaded by the JiST bootstrapper into

Table 5.5: Summary of design characteristics that bear most significantly on simulation performance.

simulator	runtime environment	compilation
ns2/PDNS	Tcl, C	split-language objects
GloMoSim	Parsec	static optimizations
SWANS	JiST	static and dynamic optimizations

simulator	environment reuse	entity memory model
ns2/PDNS	none	no isolation
GloMoSim	language	process-based isolation
SWANS	language and runtime	language-based isolation

Table 5.6: Code size metrics for the JiST and SWANS codebase.

	files	classes	lines	semi	bytecode
JiST	29	117	14251	3526	355 KB
minisim	16	37	2438	724	56 KB
SWANS	**85**	**221**	**29157**	**6586**	**514 KB**
driver	15	36	4549	1739	152 KB
	146	**414**	**50612**	**12637**	**1084 KB**

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 can load, verify, and rewrite all of the SWANS classes in approximately 30 seconds, even though only a fraction of the components are ever actually used in a given simulation. As a point of comparison, the rewrite time is less than the `javac` compilation time for SWANS. Moreover, the on-disk JiST rewriter cache entirely eliminates this overhead on subsequent runs. Some code size statistics are presented in Table 5.6. Note that SWANS is already far larger than JiST and its size will increase as more simulation components are implemented.

The rewriter processing increases the size of the simulation class files. As shown in Figure 5.7, 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, which does not affect performance. Most importantly, the increase to the code segment of non-entity classes, which is the bulk of the simulation model, is around 3%. The majority of this code is for marshaling event parameters and converting Java primitive types to and from their object counterparts for reflection-based invocation.

Finally, it is instructive to consider the impact of the various JiST API calls and annotations at the source code level. Table 5.8 shows how infrequently the various JiST calls and annotations appear within the SWANS source code. The most popular calls, as expected, are `getTime`, `sleep` and `proxy`. The *`Continuable`*

and *Continuation* annotations are mostly concentrated in the SWANS simulated socket and I/O stream classes. Even so, the total count of all JiST references throughout the code represents far less than 1% of the codebase, when measured against the total line count. The vast majority of the SWANS codebase is plain Java code completely without reference to JiST.

5.7 Language alternatives

Given that JiST is a Java-based system, it is natural to question whether Java is an appropriate language for this work and 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 Java platform boasts a large number of optimized virtual machine implementations across many hardware and software configurations, as well as a large number of compilers and languages [104] that target this execution platform. Java is an object-oriented language and it supports object reflection, serialization, and cloning, features which facilitate reasoning about the simulation state at runtime. The intermediate bytecode representation conveniently permits instrumentation of the code to support the simulation time semantics. Type-safety and garbage collection greatly simplify the writing of simulations by addressing common sources of error.

Some of these properties exist in other languages as well, and they would be suitable candidates for a JiST-like transformation. Based on the experience of implementing JiST, the most suitable candidates include Smalltalk, C#, Ruby, and Python. However, the latter two may not provide adequate performance.

Table 5.7: 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. Regular objects, which contain the majority of the code, are hardly affected.

	Object, Δ%		**Timeless,** Δ%		**Entity,** Δ%		**total,** Δ%	
base size (bytes)	**452K**		237K		14K		703K	
total increase	32K	7.3	25K	10.7	13K	98.1	72K	10.3
constant pool	18K	4.0	16K	7.0	10K	71.5	45K	6.4
code, etc.	14K	**3.2**	8K	3.7	3K	26.6	27K	3.9

Table 5.8: Counts of JiST API calls and annotations that appear within the SWANS codebase.

***annotation* / system call**	**count**
Continuable	25
Continuation	60
Proxiable	8
Timeless	8
createChannel()	3
getTime()	27
proxy()	22
proxyMany()	1
run()	3
sleep()	56
TOTAL	**213**

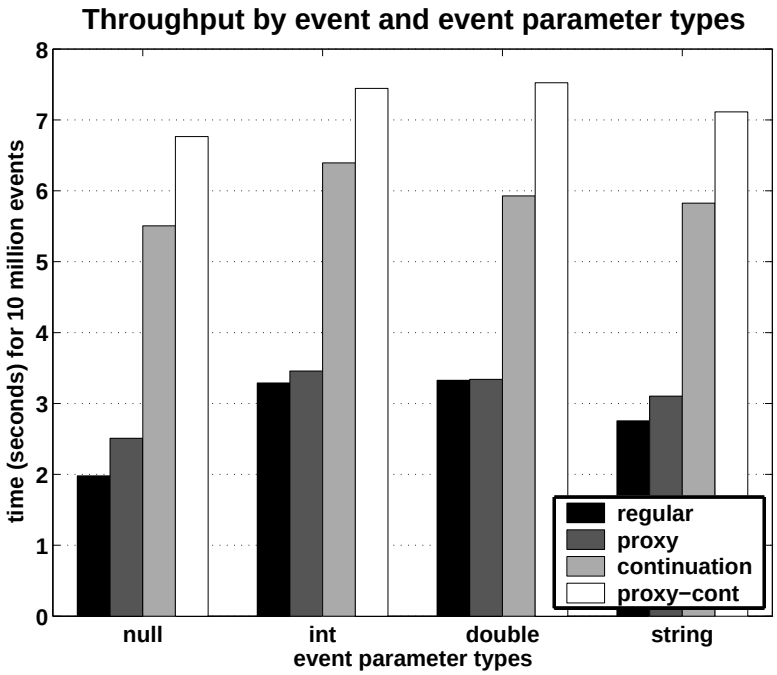


Figure 5.7: Manually boxing Java primitives for reflection-based invocation and unrolling the Java stack for blocking event continuations are Java-related overheads that could be eliminated from the performance-critical simulation event loop.

Java also has a number of disadvantages that are worth noting, because they adversely affect the performance of JiST. Java primitive types require special handling. They need to be manually "boxed" into their object counterparts for the reflection-based invocation that occurs in the performance-critical event-loop, thus incurring a relatively expensive object instantiation and requiring eventual garbage collection. Instead, the conversion from primitive to object and back should be performed internally by the JVM in an efficient manner, as in C#. Secondly, the Java virtual machine does not support tail-calls. These are a common occurrence within an event-based system and adding a bytecode instruction to assist the optimizer in detecting these could ensure that the proper stack discipline is used. Finally, Java completely hides the execution stack from the programmer. There are many applications that could benefit even from some restricted form of stack access, even as an immutable object. In JiST, it could eliminate the performance gap between regular and blocking simulation events.

Figure 5.7 shows the impact of primitive boxing and stack unrolling on various JiST event types. Null events are fastest, since they do not require the instantiation of an argument array. Events with object parameters (`string`) are only slightly faster than events with primitive parameters (`int` and `double`). Proxied events have equivalent performance to regular events, even though they add an additional method invocation. However, this additional stack frame hurts performance in the case of proxied-blocking events, which must disassemble the stack manually, frame by frame. Note also that the proxying method is simply a wrapper, so it is not touched by the CPS transformation. The JVM should certainly implement it using a tail-call, if not entirely inline the method.

5.8 Summary

In this chapter, I have highlighted the performance advantages of virtual machine-based simulation using macro and micro-benchmarks. JiST and SWANS outperform existing simulator alternatives both in time and space, due to fundamental design decisions. I have also shown that the rewriter overhead at the bytecode level, as well as the source code annotations required, are minimal.

Chapter 6

Density Independent Route Discovery

Ongoing research into dynamic, self-organizing, multi-hop wireless networks promises to improve the efficiency and coverage of wireless communication. However, the ability to scale ad hoc networks to large numbers of nodes remains an open problem. A number of ad hoc network routing protocols have been proposed [57, 82, 29, 15], but their evaluation has been hindered by the capabilities of existing simulators. The majority of the published simulation results are of networks with sizes on the order of a few hundred nodes.

In this chapter, I analyze the **B**ordercast **R**esolution **P**rotocol (BRP), the query propagation component used in the **Z**one **R**outing **P**rotocol (ZRP) framework [48]. I compute the cost of discovering a route within a flat ad hoc network and show that one can discover a route with cost proportional only to the area of the network, or independent of the number of network nodes. Furthermore, I show that this is optimal and that bordercast possesses these properties. These experiments, at the scales required to reach the appropriate conclusions, were possible due to the performance and scalability of JiST and SWANS. Thus, this chapter provides both a scalable networking research result as well as an example of the type and scale of research that these tools allow. In fact, such research was the motivation behind SWANS, which, in turn, motivated the JiST work.

6.1 Background

The recent widespread adoption of wireless communication technologies over the *last* hop of packet-switched networks has greatly enhanced connectivity and en-

abled numerous new applications. There are many available standards and protocols for different kinds of wireless links that can be used in wide-area, cellular, local area, or even "personal" area networks. Each of these is optimized for a specific point along the fundamental distance-capacity-power trade-off curves that meets the demands of a target set of applications (Table 6.1).

Ongoing research into dynamic, self-organizing, multi-hop wireless networks, called "ad hoc networks," hopes to facilitate yet another stride in network connectivity by improving the *efficiency* and *coverage* of wireless communication. The idea is well known: rather than tether wireless devices to the wired world via a *single* wireless hop to a base station, a device within an ad hoc network may be connected via *multiple*, shorter hops across other wireless devices that function as routers and repeaters. And, in general, the devices may also communicate with each other and even operate in isolation, without a base station. Regardless of the traffic pattern, since the strength of a signal dissipates faster than the square of the distance traveled, multiple shorter wireless links can support the same bandwidth capacity as a single long wireless hop using less power, a scarce resource in mobile settings. Furthermore, the overall effective capacity of the airspace may be increased, because these lower power, shorter range transmissions generate less interference. Alternatively, for the same power, can one achieve greater capacity or coverage.

Ad hoc networks have a variety of natural civil and military applications. They are particularly useful when networking infrastructure is impossible or too costly to install and when mobility is desired. As a benign, civil example, take a pre-existing, unwired office building that is to be filled with computers, printers, telephones, card readers, fire sensors, photocopiers, and many other devices that require or

can be enhanced with network connectivity. Assume, without loss of generality, the use of ubiquitous 802.11 networking equipment to network them, and radio power settings that provide an effective transmission radius of 50 meters. Using wireless links over the last network hop alone, one can provide network connectivity to an entire floor with (wired) base stations placed so that there exists at least one within 50m of every device.[1] The minimum number of base stations required is proportional to the area that is to be covered.

In contrast, by utilizing multiple wireless hops and assuming sufficient density (i.e., assuming that no device is more than 50m from at least one other device that can already be connected), one could achieve complete connectivity for the entire floor with just a single base station. More generally, a device can communicate with some destination, if and only if there exists at least one route of wireless hops, all shorter than 50m, between them. The devices may also change location over time, and they may join and fail (or voluntarily leave) the network. Such an architecture that does not contain long-range links and utilizes multi-hop routes across short wireless links in an unconstrained manner is called a *flat* ad hoc network.

Extending this example, is it feasible to connect the whole office building to a single point without any wired infrastructure? Is it possible, perhaps, to connect an entire university campus or a small town using just short wireless hops among wireless devices? The vision of extending the reach of wireless communication in this manner is enticing. However, routing and transmitting packets efficiently becomes increasingly difficult as the ad hoc network grows. In general, increasing

[1]Though wireless links are appropriate for a large class of applications, they may not adequately replace wires for certain demanding uses. Other issues need to be considered, such as network capacity, end-to-end latency, and security.

the scale of ad hoc networks to such magnitudes remains an important research challenge.

In the remainder of this chapter, I address an important component of this problem. Specifically, I analyze the cost of discovering a route within a flat ad hoc network in the absence of any information about the desired destination node except for its unique address. I show that one can discover a route with cost proportional only to the area of the network, and independent of the number of nodes in the network (i.e., independent of the network density). Furthermore, I show that this is optimal: this cost is a lower bound for any possible route discovery protocol that does not rely on additional information about the destination node. I analyze a protocol, called bordercast, which has these characteristics and evaluate it in simulation.

6.2 Scalability limits

The network model I shall use is that of a flat ad hoc wireless network. The network consists of n nodes that may move around within a given area A. These nodes are independent, unaware of their location, and randomly, but uniquely addressed. Nodes can communicate with other nodes that are located within their transmission radius, r. All nodes are equal participants in the network and there is no central, coordinator node or point of failure. There is also an assumption of a single, shared channel and, for simplicity, that all links are bi-directional.

Routing and transmitting packets efficiently over such a network becomes difficult as it grows in size. In fact, the scalability of flat ad hoc networks is fundamentally limited along a number of dimensions:

Capacity – The scalability of the effective capacity of a network depends on many parameters, including the traffic model (i.e., how it is used) and the node density. For a network with uniform node density, the effective capacity can grow proportionally to the network size, under the condition that the average end-to-end communication distance is constant. Consider, for example, the case of every node communicating only with its neighbors. At the other extreme, if all the nodes choose to communicate with one central node in the network, then the effective capacity of the entire network is capped by the capacity of that central node, regardless of the network size. The question of network capacity has been widely investigated under numerous assumptions [45, 43, 7] and is outside the scope of this work.

Latency – Flat wireless ad hoc networks are inherently limited in the area that they may cover, because, without long-range wireless or wired links, the network distance (in hops) between two nodes is proportional to the physical distance between them. So, too, is the network latency of the shortest route. For instance, traversing just 50km in 50m hops would require at least 1000 hops. This is far greater than the diameter of the Internet, where an initial TTL greater than 60 hops is considered safe[34]. Thus, one should restrict the network to a physical area, A, with diameter dr, or $A \leq \pi(dr)^2/4$, where d represents a network diameter of reasonable magnitude, which is likely (though not necessarily) smaller than that of the Internet. In practice, long-range network links may be used to inter-connect such areas in a hierarchical fashion, but such a network architecture is also outside the scope.

Discovery – Finally, a frequent operation in an ad hoc setting is to query it for information. The information may be at one or more individual nodes, or

it may be a function of some subset of the network. The ideas in this chapter are applicable to many kinds of queries, however, the focus will be on the cost of performing a route discovery, wherein a (source) node queries the network for a route to some destination.

6.3 Route discovery

A number of ad hoc network routing protocols have been proposed, including DSR [57], AODV [82], OLSR [29], and TBRPF [15] among many others. Each of these protocols is designed and optimized under different networking assumptions, such as the expected traffic pattern and the rate of topological change in the network. For example, the OLSR protocol is proactive in its route discovery, which is suitable for more static networks, while AODV is reactive, and thus is more efficient in highly mobile settings. The protocols also differ in their use of route caches, their mechanisms for detecting route failures, and their capabilities for maintaining routes.

However, these varied routing protocols also share certain commonalities. Most obviously, they provide the same functionality and interface to higher layers of the network stack, which is to furnish routes to any requested node in the network. Secondly, at one time or another, either due to limited cache sizes, changes in the network that invalidate existing information, or the arrival of a new query for an unknown destination, each of these protocols is forced to send a query into the network in search of some node or information about that desired destination available at some other, possibly nearer node.

Flooding is a naïve query propagation protocol, whereby each node, upon receiving a query for the *first* time, merely rebroadcasts it, possibly with some jitter

to reduce the probability of collision. The algorithm for flooding is shown in Figure 6.1. Ignoring failures, every node connected to the source, receives the query at least once and transmits it exactly once. Truncating the flood using an expanding TTL "ring", as in AODV and others, is merely a stop-gap measure that is useful only when the destination node or cached information happens to exist nearby. In general, as the number of network nodes increases, the cost of the flood increases in proportion as well.

Having established that query propagation is a necessary element of ad hoc network routing protocols and that flooding-based propagation is costly, one would like to replace this protocol with a more efficient and scalable query propagation protocol. The defining property of any propagation protocol is that it propagates a query to all nodes that need to hear it. Thus, in the absence of *any* information about the destination node, including cached information and even general location information, a route discovery query should be propagated to all nodes that are connected to the propagation source. But, it is certainly not necessary for every node to transmit the query, as is the case with flooding. This observation allows for significant improvements of many existing ad hoc network routing protocols. When operating in dense networks, the flooding-based propagation component of these protocols causes unnecessary transmissions and broadcast "storms."

6.4 Optimal propagation

Given some network, $G = \langle N, E \rangle$, where N denotes the set of nodes distributed uniformly across an area, A, and E denotes the links between them, determine the number of transmissions, t, required to propagate a route query from some source

Table 6.1: Capabilities of various wireless technology options (order of magnitude)

area	distance	bandwidth	power	protocols
Wide	10 km	0.1-2 Mbps	1 W	GPRS, CDMA, W-CDMA
Local	100 m	1-50 Mbps	300 mW	801.11b/a/g, Ricochet
Personal	10 m	1 Mbps	10 mW	Bluetooth, Infra-Red

```
state
  local: address
  processed: set of query

RECEIVE-FLOOD(q: query):
  PROCESS-QUERY(q)
  sleep JITTER-TIME()
  if q ∉ processed then
    processed ← processed ∪ q
    BROADCAST(q)

FLOOD(q: query):
  RECEIVE-FLOOD(q)
```

Figure 6.1: Flooding query propagation protocol

node, $n_0 \in N$, to all the other n_0-connected[2] nodes, $N^c_{n_0} \subseteq N$. Clearly, $t \leq |N^c_{n_0}|$, since this is the cost of flooding: each n_0-connected node transmits the query once.

However, this analysis fails to capture the spatial properties of a wireless broadcast. As defined above, network nodes are placed at random locations within the limits of the ad hoc network area, A. Thus, in the worst case, one would need to cover the entire area with query broadcasts. A single wireless broadcast can be received over an area $a = \pi r^2$ around the transmitter, where r is the transmission radius, as defined above. Thus, even if one could place transmitters at will, one still would need at least $t \geq (1+\epsilon)A/a$ transmitters, where ϵ accommodates for the packing inefficiency of the transmission coverage circles. For convenience, let $\rho = (1+\epsilon)A/a$.

As the size of the network is increased, by adding nodes at random locations within A, a number of things occur. The density of the network increases in proportion. The network becomes fully connected with high probability, so that $N^c = N$. And, the probability of having a node within δ distance of any chosen transmitter point, $P_\delta(t) = 1-(1-\pi\delta^2/A)^{|N|}$, approaches 1. To propagate a query over the entire area A, select transmitter points uniformly across A, such that the distance between any two is at most r. It is possible to cover the entire area with ρ circles of radius r. Place nodes at the centers of those circles and connect those nodes with an additional $\rho - 1$ intermediate nodes. Thus, even as the number of nodes in the network increases, one can still flood the entire network at a cost of $t \leq 2\rho - 1$ transmissions. Thus, $t = \Theta(A)$.

[2]Connected in the graph-theoretic sense, i.e., possibly through multiple hops.

The upper bound just derived may not be satisfying, since the selected transmission points are not actually nodes within the original network. Therefore, consider the *dominating set* over G, $\tilde{N}$. (A set of nodes, $\tilde{N}$, is dominating over G, if and only if every node in G is either an element of $\tilde{N}$ or is a neighbor of some node in $\tilde{N}$.) The *dominating number* of the network, or the size of the minimum dominating set, is $\gamma(G) \leq \rho$. If there are any more nodes in a dominating set than this upper bound, then at least one member node covers an area around it that is already completely covered by the remaining nodes in the set, and that node can be removed to form a smaller dominant set. Then, construct a minimum *connected dominating set*, $\tilde{N}^+$. A connected dominating set is a dominating set with nodes from G added to connect the existing dominating nodes, whenever those nodes are connected in G. Thus, by construction, the number of connected components in the connected dominating set will be equal to the number of connected network components in G. Note, also, that by the definition of the dominating set, the connected dominating set can be formed by adding at most 2 nodes for every dominating node: the neighbors of the two dominating nodes being connected. Thus, $|\tilde{N}^+| \leq 3|\tilde{N}|$, and the minimum connected dominating number of G, is $\gamma^+(G) \leq 3\rho$. Finally, since the minimum connected dominating set contains all the nodes that must transmit the query in order to completely propagate it through G, it remains that $t = \Theta(A)$.

To summarize, the optimal number of transmissions required to propagate a query is proportional to the area, A, of the network. It is independent of the number of network nodes, or equivalently of the network density. This is intuitively correct. Adding a node in an area that is already covered by an existing set of propagating nodes, should not increase the number of transmissions required. And,

if the area is finite, only a finite number of nodes are required to propagate the query across it, regardless of the number of nodes in the network that are to receive the query.

6.5 Zones and bordercasting

In reality, the minimum connected dominating set of the network is not computable. It is an NP-complete problem, even if a priori knowledge of the entire network topology were available. Instead, one can substitute an approximate algorithm that uses heuristics and only local information. The following is an overview of the **B**ordercast **R**esolution **P**rotocol (BRP), the query propagation component used in the **Z**one **R**outing **P**rotocol (ZRP) framework [48]. Bordercasting can replace the flooding-based query propagation used in many existing ad hoc network routing protocols.

The bordercasting functionality depends on information about the surrounding *zone* of a node. Each node has its own zone, which is defined to be the set of nodes that are within R network hops from it. R is called the zone radius. Each node knows about all the other nodes within its zone, as well as the links among those nodes. The mechanism by which this zone information is collected and maintained will be described in section 6.6. The zone radius, R, is a protocol parameter that may change over time and need not be homogeneous across the entire network [97]. However, for simplicity of exposition, assume that it as a network-wide constant. The *border* of a zone is defined as the set of nodes that are *exactly* R hops away.

Like flooding, the bordercast protocol propagates the query across the entire network. However, while flooding attempts to iteratively relay the query to any *neighbors* that have not heard it, the bordercast protocol seeks to iteratively relay

the query to any of its *border* nodes that have not heard it. Thus, while all the neighbor nodes will receive the query broadcast, not all of them will need to retransmit it on its way to the border nodes. If one considers the nodes within the zone to be a micro-ad hoc network with area, $A_z = \pi(Rr)^2$, it is clear that the cost of propagating the query across the zone is a function of its area and not of the number of nodes within it. Therefore, because of the broadcasting nature of wireless communications, the bordercast protocol broadcasts the query to all its neighbors, but selects only a few to re-bordercast the message. The other neighboring nodes are silent recipients.

It is important to understand that bordercasting does *not* actually attempt to deliver the query to every node within its zone. Rather, its objective is to relay the query only to any *border* nodes that have not yet received the query. The protocol still works correctly, because each node in the network maintains information about all the nodes within its zone and can answer queries about them or, at the very least, forward the query directly to the desired node. Thus, a node is *covered*, if and only if the query has been received by any node within its zone. The bordercasting protocol ensures that every node in the network is covered by the query. With larger zone radii and at higher network densities, a significant fraction of the network may be covered without actually receiving the propagating query.

Figure 6.2 contains the bordercast algorithm, which runs independently on every node. A bordercast query propagation is initiated with a call to BORDERCAST, and incoming messages are processed by RECEIVE-BORDERCAST. The ZONE and BORDER functions return the zone and border node sets, respectively, around a given node, based only on the local state.

```
type
  msg := {
    src: address,
    targets: set of address,
    query: query
  }
  info := { addr: address, ... }
state
  local: info
  zone: {
    nodes: set of info
    links: set of { src: address, dst: address }
  }
  covered: map query to set of address

RECEIVE-BORDERCAST(m: msg):
  ▷ accumulate query coverage information
  covered[m.query] ∪← ZONE(m.src)
  if local ∉ m.targets then
    covered[m.query] ∪← ZONE(local.addr)
  ▷ process query and wait to avoid collision
  ▷   and hear other broadcasts
  PROCESS-QUERY(m.query)
  sleep JITTER-TIME()
  ▷ relay query to any uncovered border nodes
  border ← BORDER(local.addr)−covered[m.query]
  if border ≠ ∅ then
    msg ← msg {
      src ← local.addr,
      targets ← SELECT-NEIGHBORS(border),
      query ← m.query
    }
    BROADCAST(msg)
    covered[m.query] ∪← ZONE(local.addr)

BORDERCAST(q: query):
  msg ← msg {
    src ← NULL,
    targets ← { local.addr }
    query ← q
  }
  RECEIVE-BORDERCAST(msg)
```

Figure 6.2: Bordercast query propagation protocol

The local state of each node consists of the zone-wide information, maintained by a separate protocol, as well as information regarding which nodes within the zone have already been covered by a query: `coverage`. This query coverage state maintained by each each node lies at the heart of the protocol and directs its behavior. As with flooding, a bordercasting node should transmit a query at most once. To ensure this property, the protocol marks[3] all the nodes of its zone, including the border nodes, as covered after a query has been relayed. When all the border nodes are covered, there is no need to relay the query. Similarly, when a node receives a bordercast message from some neighbor it marks all the locally known nodes in the zone of that neighbor as covered. In other words, the node updates its local query coverage state to indicate that all nodes within its zone that are within R hops of the sender node have been covered by the query. Thus, the coverage state directs the query outward, toward an expanding border.

Each bordercast message contains the query to be relayed, a source address, and a list of target addresses. The bordercast message is always broadcast and the list of target address are always selected from among the neighbors of the sender. If the receiving node is not one of the selected targets, it is implied that it is not required for the query to reach the border nodes of the sender. Furthermore, by the definition of a zone and the manner by which targets are selected, each of receiver's border nodes must be a border node of one of the targeted neighbors. Thus, when a node receives a bordercast message and is not in the target set, the protocol marks its entire zone as covered. The effect is, as above, to ensure that a non-targeted node will remain silent, since it is not required for query propagation.

[3]A local operation: updates `coverage` in the local node state.

In contrast, a targeted neighbor node that receives a query is responsible for re-bordercasting it to any of its uncovered border nodes. The protocol pauses for a short random interval before doing so. This wait time lowers the collision probability. It also allows the node to receive other bordercasts that may be occurring at neighbor nodes during this time. Learning that the query was processed at other nodes, may partially or completely cover the remaining uncovered border nodes and either reduce the number of targets required or perhaps eliminate the need to relay the query entirely.

Finally, before broadcasting the query, the protocol must select the target neighbors: SELECT-NEIGHBORS(). For each uncovered border node, there must be at least one neighbor chosen in the direction of that border node. In other words, the network distance between the selected target neighbor and the uncovered border node must be $R - 1$ hops. There may be many sets of nodes that meet this criterion, and one would like to find the smallest such set. Since this matching problem between closest neighbors and their uncovered border nodes is also NP-complete, a greedy approximation is implemented. The neighbor node that covers the greatest number of uncovered border nodes is chosen first. The border nodes closest to the chosen target are then covered, and then the algorithm iterates until all the border nodes have been covered. The query is then broadcast with this list of targets.

To summarize, Figure 6.3 depicts a bordercast in progress. The zone radius is 2. The query was initiated from node S, and arrived via G and B to node A. Note, that node B targeted only nodes A and C with its query broadcast, in order to reach its border node set $\{H, D, E\}$. Node F is not targeted since node A already covers it, in addition to covering D. Thus, when F receives the query from B, it

processes the query, notes that its entire zone is covered (since F is not a target of the query), and, therefore, remains silent. Similarly, G receives the query from B and remains silent, since G has already forwarded the query (to B, in this case) and marked all the nodes of its zone as covered. When A receives the query from B, it locally marks all the nodes of B as covered. Covered nodes are represented as black nodes in the figure. S is not marked, since A does not know about it. I, J, and K are also not marked, since they lie outside the zone of B. Thus, out of A's border set, which is $\{G, H, I, J, K\}$, only $\{I, J, K\}$ remain uncovered. Based on this information, and assuming that no other bordercasts are heard while A pauses, A will broadcast the query with $\{D, E\}$ as targets.

6.6 Zone maintenance

The bordercast operation requires information about the surrounding zone, which is collected and maintained by a separate protocol in the ZRP framework, called **I**ntr**A**-zone **R**outing **P**rotocol (IARP). The result of executing an IARP protocol is local knowledge of the zone, which includes all the nodes, N_z, that are within R hops as well as the state of the links, E_z, among them. For a sparse network, $|E_z| = O(|N_z|)$. However, as the network density increases the size of link state grows quadratically, – i.e., $|E_z| = O(|N_z|^2)$ – as does the cost of the zone maintenance protocol. This places a limit on the size of the zone, particularly when the membership of the zone and its link state change rapidly, such as in the case of high node mobility. Therefore, it is important for the zone maintenance protocol to be as efficient as possible. The following describes two possible zone maintenance protocols: IARP-node and IARP-zone.

The zone maintenance protocols receive information about their immediate neighbors and link state from an NDP (Node Discovery Protocol). The NDP can be either a simple heartbeat based node discovery protocol running at the network layer, or some other mechanism operating lower on the network stack. It provides information about the first hop with LINK-UP and LINK-DOWN notifications whenever a neighbor is discovered or is lost, respectively.

The IARP-node protocol broadcasts these *local* link state changes in a packet with the TTL set to R hops. Each link update is sequenced at the link's source. Every node that hears such a packet for the first time, simply decrements the TTL and rebroadcasts it, unless the remaining TTL has reached zero. In this manner, the entire zone learns of the change. The protocol also incorporates a jitter delay so as to lower the collision probability and possibly accumulate a few changes at the source node before sending a packet. For nodes that join the network, there is a mechanism to acquire zone state from a neighbor. Finally, there is also a periodic broadcast of the full link state at a larger time interval, which permits us to eventually expire links at nodes that have left the zones of either of the link endpoints or to nodes that have died.

When the zone is stable, the IARP-node protocol is silent, modulo the infrequent periodic broadcasts. However, in the worst case, for a zone of k nodes and l links that is changing all the time, each of k nodes will transmit a packet containing $O(l)$ link changes that will be rebroadcast by $k-1$ other nodes in the zone. In other words, the worst case is $O(k^2 l)$ information and $O(k^2)$ packets per zone, or $O(kl)$ information and $O(k)$ packets per node, where k and l are functions of the zone radius R, the network density n/A, and the transmission radius r.

The IARP-zone protocol takes a different approach. Every node broadcasts only a *single* packet at every period, if there has been any change to the link state. Each packet has a TTL of 1, but contains the known changes of the entire *zone-wide* link state since the last transmission. The same packet is received by neighbors in multiple directions, but each one prunes out the information relevant for its zone based on shortest network distance calculations. As in the IARP-node protocol, each link update is sequenced at the link source, there is a forced periodic update at a longer time interval that permits link expiration, and the protocol is otherwise silent if the zone is stable. In the all-changing, worst case, there is still $O(kl)$ information sent per node, but only in a single packet. This larger packet may be readily sub-divided into independent, smaller packets, if the information happens to exceed the link MTU. However, the advantage of transmitting zone-wide changes in batches is retained: multiple links updates contain common endpoints, allowing for a more efficient encoding. Within each packet, there is a table of endpoints, which contains IP addresses and link source sequence numbers. The packet then contains a list of source endpoint indices, each with a sub-list of destination endpoint indices, representing all of the links. Bi-directional links are encoded using a reversal bit.

6.7 Bordercast evaluation

In this section, I evaluate the performance of bordercast using SWANS. The first experiment measures the relative unit cost of each of the protocols discussed, for networks of different sizes, but at constant density. The network is generated by placing wireless nodes randomly within a square field, and increasing the field size in proportion to the number of nodes. Each network node is static in this

experiment, and is turned on at time $t = 0$ with no information other than its unique address. The protocol stack at each node comprises a wireless radio, the 802.11b MAC, IPv4 network, ZRP routing, UDP transport, and test application components that generate traffic. Note that since the various protocols perform link-level broadcasts, the 802.11 collision avoidance and retransmission mechanisms do not play a role in these simulations. However, each of the simulated protocol incorporates jitter to reduce the probability of collisions and is already resilient to point failures, either due to repetition (NDP) or due to a flooding-like behavior (IARP and BRP). The simulator accounts for signal interference, path loss, and fading.

The simulator measures the *unit* packet cost of a protocol, which is defined as the number of packets sent throughout the network to perform a single round or operation. The unit cost of the IARP protocols is the number of packets for the protocol to quiesce, such that every node has learned its complete zone state. Since the nodes begin with no information, this represents the worst case for the protocol, which is when *all* the zone links must be added. The unit cost of a bordercast operation is the number of packets transmitted to cover the entire network with a query. For any fixed density, both of these protocols grow linearly with the area of the network or, equivalently, linearly with the number of nodes in the network.

Presented next, is a similar experiment, but with the network area kept constant as the number of network nodes increases. Figure 6.4 compares the performance of query propagation over the fixed area using flooding versus using bordercast. Each point represents the average of at least 10 runs. The graph shows how a flooding-based propagation grows in proportion to the number of nodes, but that bordercast

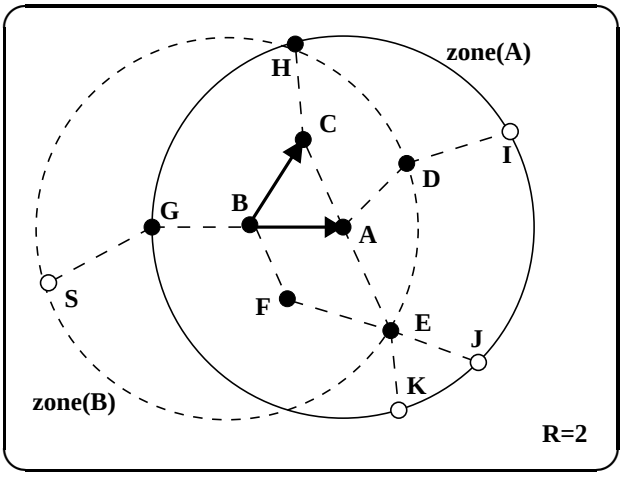


Figure 6.3: A bordercast in progress

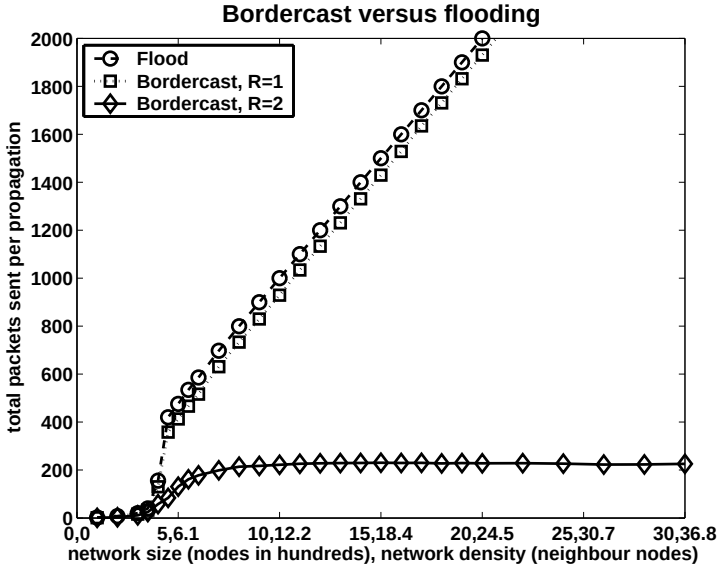


Figure 6.4: Unlike flooding, the bordercast cost of query propagation is independent of the network density.

is density independent. In other words, adding more nodes to the network does not increase the cost of bordercasting. Note that the left side of the curves represents a very sparse network that is poorly connected. In this case, both flooding and bordercast are simply not able to span the area for lack of intermediate nodes. The x-axis shows both the total number of nodes, as well as the network density in terms of the expected average number of neighbors per node. This number of neighbors is computed from the node density and the transmission radius, i.e., $E[l/k] = \pi r^2(n/A)$. It matches the values reported by NDP in simulation. Finally, one can observe that by setting the zone radius to 1, the bordercast performance degenerates to flooding. This is expected, since with $R = 1$ the border set becomes the neighbor set. The slight advantage of bordercast over flooding is merely an edge effect: edge nodes do not retransmit the query under the bordercast protocol, because all of their neighbors are already covered.

Increasing the zone radius improves the performance of bordercast somewhat, as shown in Figure 6.5. However, the majority of the improvement is due to an edge effect: the query need not be relayed to the last $R - 1$ nodes at the edge of the network. While these edges do not diminish as the number of nodes in the network increases, they should not be attributed to bordercasting. Instead, this benefit is reminiscent of proactive routing, and will therefore erode in cases of increased mobility. To quantify and eliminate the impact of the edges, Figure 6.6 shows the same experiment, but run on a field with opposite edges wrapped around to create a torus.

Some smaller improvements due to larger zone radii can be seen within the core of the network, where the protocol can sometimes avoid regions that are sufficiently sparse. These are, in some sense, "internal edges" of the network.

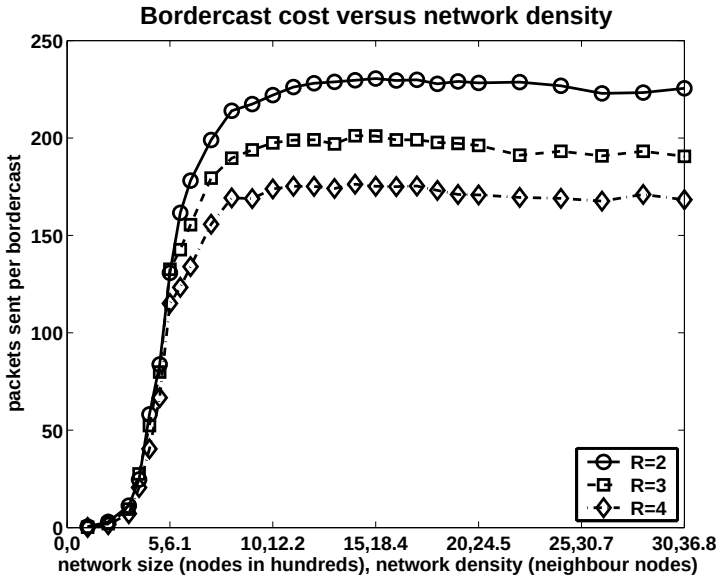


Figure 6.5: Increased zone radius improves bordercast performance, primarily due to edge effects.

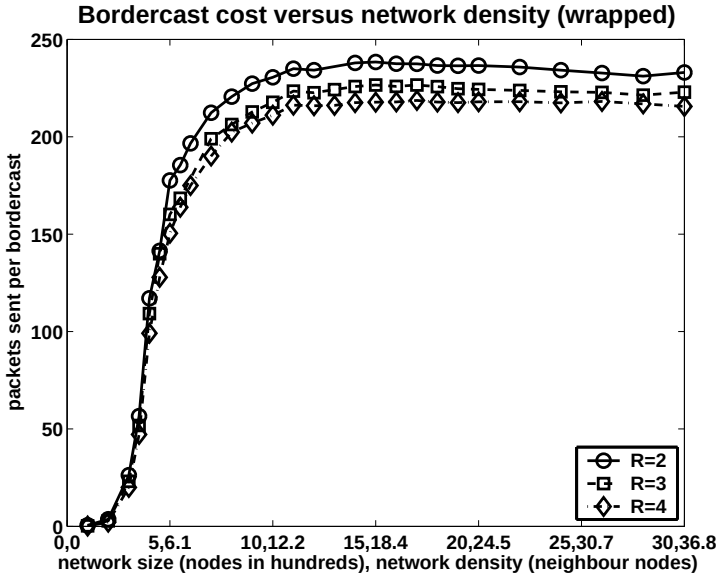


Figure 6.6: Discounting edge effects, bordercast cost is not significantly affected by increased zone radius.

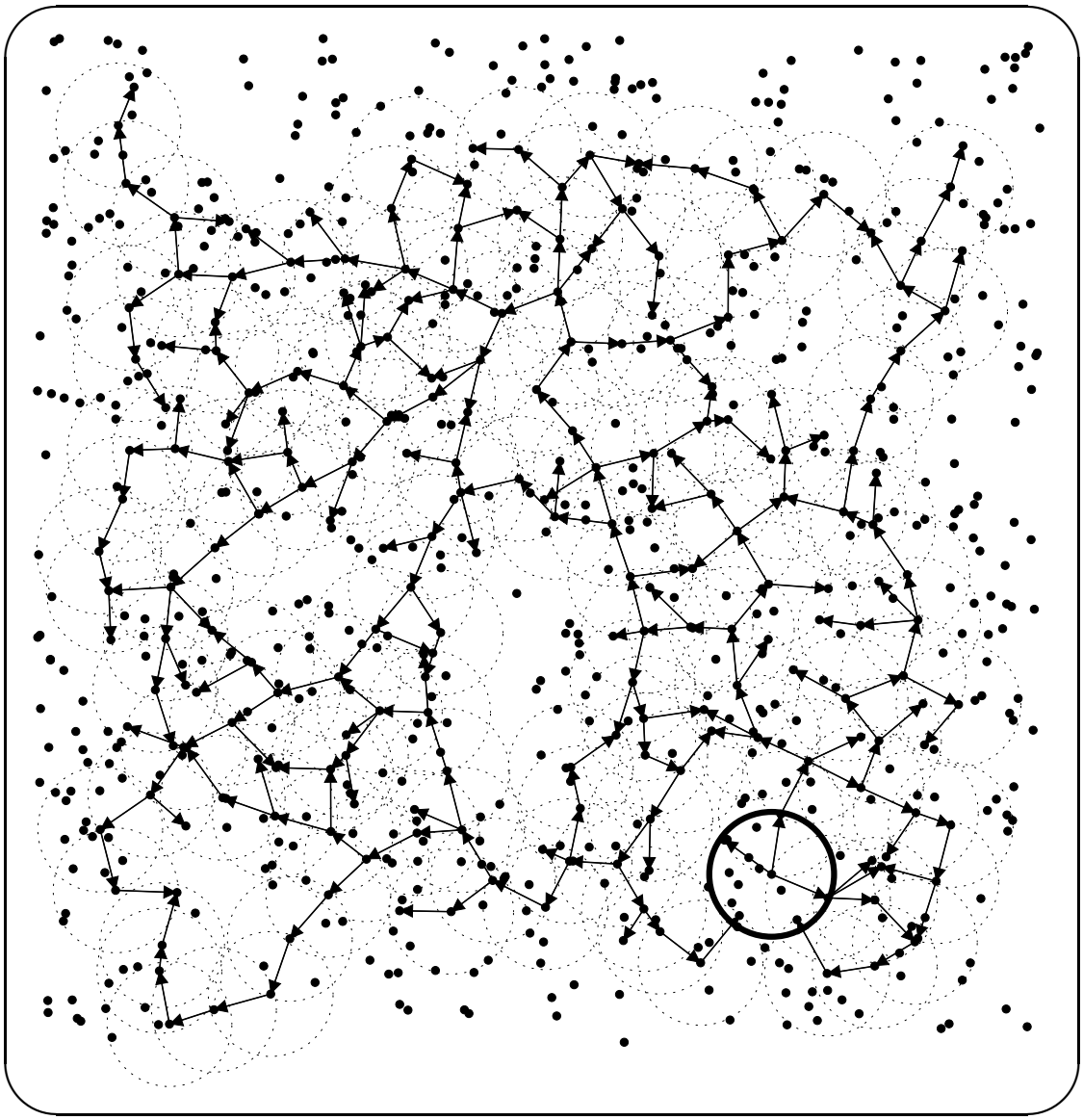


Figure 6.7: An example 800-node, $R = 4$ bordercast plot

Figure 6.7 highlights this phenomenon with a spatial plot of one of the smaller ($n = 800$, $R = 4$) simulations from Figure 6.5. The heavy circle at the bottom right highlights the query source. The circle's radius equals the transmission radius and lighter circles surround nodes that transmitted the query. Thus, all nodes within circles receive the propagating query. Arrows represent the targeted neighbors, which may relay the query, if necessary. Notice that many nodes are not within these circles, which means they never hear the query. However, *all* the nodes are contained within 4 hop-sized circles around nodes that actually receive the query, a close approximation of the actual zones, indicating that the protocol is covering the network. (Zone circles are omitted to improve the clarity of the image.) One can also see an example of internal nodes in the center that are covered, but without receiving the query.

Another interesting phenomenon shown by the spatial plot is that the targeted neighbors are usually found near the boundaries of the transmission circles, even though there is no location information available to the protocol. This occurs because the bordercast protocol tries to minimize the number of targeted neighbors by selecting those neighbor nodes that are closer to (i.e., $R - 1$ hops from) the *greatest* number of uncovered border nodes.

The following experiments analyze the cost of maintaining the required zone state at each node. Shown in Figure 6.8 is the effect of increased density on the number of packets sent by the two zone maintenance protocols. The graph shows the cold-start scenario, where all links in the zone must be discovered and added. As expected, the total number of the IARP-node packets (left axis) increase quadratically with the density, i.e., $O(k^2)$, where k is the number of nodes within the zone: each node sends $O(k)$ packets and there are more nodes in the network.

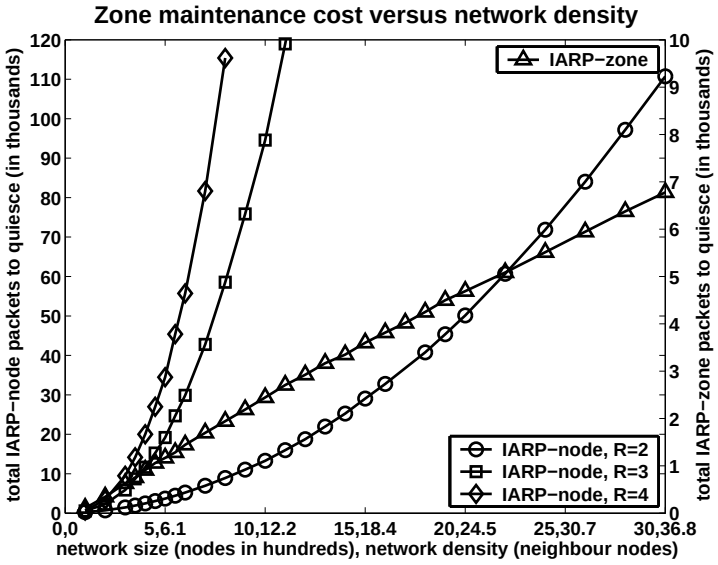


Figure 6.8: Cost of zone maintenance increases dramatically with increased density and zone radius.

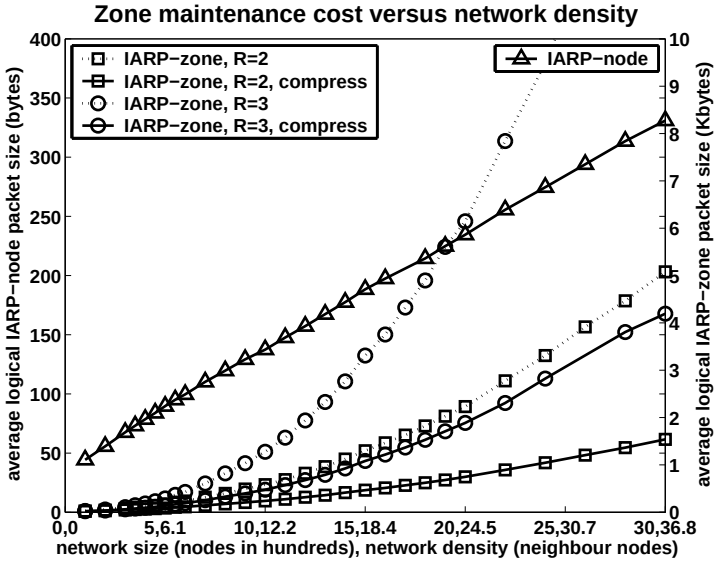


Figure 6.9: Aggregated link state can be encoded efficiently to reduce average size of update packets.

In turn, the number of nodes within a zone increases quadratically with the zone radius, and thus the total number of packets in the network increases with the fourth power of the zone radius! The $R = 4$ curve is actually significantly lower than expected due to the large number of collisions caused by the flurry of link state updates in the large zone. However, since IARP-node is a flooding protocol, lost packets are often retransmitted by other neighbors and, with high probability, any loss of information is local. Nevertheless, it is interesting that the missing link state does not appreciably affect the bordercast performance above. As expected, the number of IARP-zone packets (right axis) increases linearly with the density, since each node sends a constant number of packets, forwarding the "waves" of new information traveling in each direction. The number of IARP-zone packets is not affected by the zone radius, since it merely passes along more information within the same packet.

Figure 6.9 compares the average packet sizes of the two zone maintenance protocols. IARP-node packets (left axis) are very small, because they contain the link states of only a single node. In contrast, IARP-zone packets (right axis) contain information about changes to the entire zone. The size of these packets is proportional to the number of links, which grows quadratically with density and with the fourth power of the zone radius. However, the more efficient encoding of this information saves around 60% of the packet size at a density of 30 neighbors per node, independent of the zone radius. The compression increases with the density, since the proportion of common link endpoints increases with the density.

Figure 6.10 shows the total bandwidth consumed for each of the zone maintenance protocols to quiesce starting from no zone state. The two protocols transmit the same amount of information, but IARP-zone outperforms IARP-node through

efficient encoding of the update packets. The plotted data does *not* include packet headers; this would further benefit IARP-zone.

Finally, it is important to consider the effect of mobility on each of these protocols. By their design, both the node discovery and bordercasting protocols are unaffected by mobility. However, mobility can change the zone link state and its membership, which will necessitate zone maintenance. To measure this cost, I create a random network and allow the zone maintenance protocols to quiesce. I then move each node a fixed distance in a random direction and measure the number of packets required to update the zone information. Figure 6.11 shows the total number of IARP-node and IARP-zone packets for a network of 10,000 nodes. One can see that the number of packets grows in proportion to the number of changes in the zone, but has an upper-bound that corresponds to the total amount of zone information. The lower line below each curve is the unit cost of each respective zone maintenance protocol from a cold start. The upper line above each curve is twice this value. The additional transmissions above the lower line are due to link state *drop* notifications, which do not occur from a cold start. Notice, however, that the IARP-zone cost actually tends back down toward the lower line. This peculiar phenomenon results from IARP-zone pruning its link state after incorporating each update packet. This pruning is essential, because under IARP-zone, nodes will receive some link state that is not relevant for their zone. And, if this new link state is forwarded on, the zone information at each node will eventually include the entire network. An added bonus of this pruning is that link failure notifications are suppressed when a node has traveled so far out of its original zone as to be irrelevant.

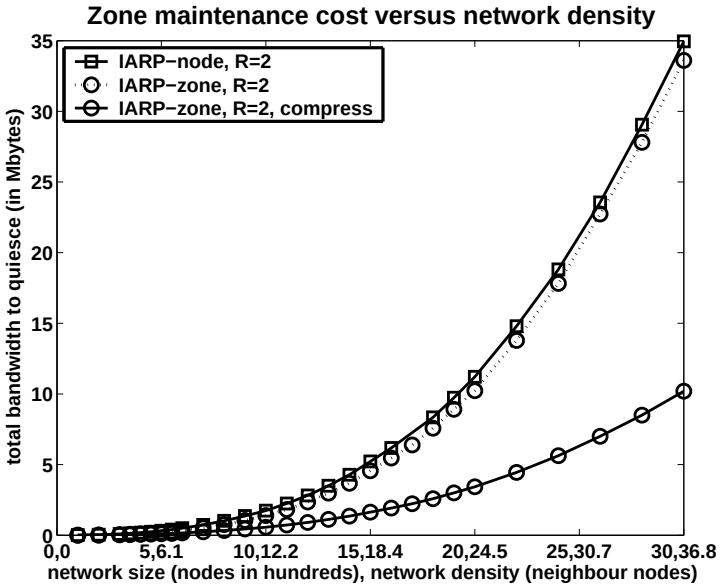


Figure 6.10: Comparing the two zone maintenance protocols shows that zone-wide link update aggregation and efficient encoding is beneficial.

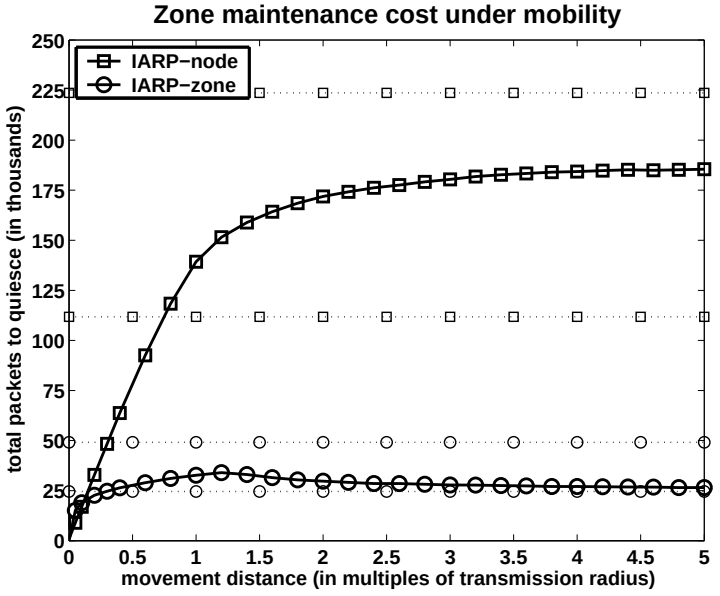


Figure 6.11: Mobility increases the cost of zone maintenance.

6.8 Conclusions

Given these findings, one can draw certain inferences and make recommendations regarding the use of bordercasting in ad hoc networks:

- *Bordercast should be used in place of flooding.* Bordercast performs query propagation with cost proportional to the the area of an ad hoc network (or equivalently, the network diameter), regardless of the density of nodes (or equivalently, the number of nodes). It can replace the flooding-based query propagation found in many ad hoc routing protocols, as well as in other network querying operations, such as resource discovery and some sensor network data queries.

- With respect to bordercast performance, *set the zone radius, R, to 2 hops.* Setting a higher zone radius results in little bordercast improvement and substantially increases the cost of zone maintenance, especially at higher network densities. Note that there may be other reasons to have a larger zone, including proactive route maintenance and a high rate of route requests relative to the rate of link changes (i.e., a mostly stationary network). However, one of the contributions of these experiments is that bordercast performance is not among those reasons.

- *Aggregate and compress link state updates.* IARP-zone outperforms IARP-node, because it aggregates link state updates, thus transmitting fewer packet headers and reducing the average packet size through efficient encoding of links.

- *Adjust the transmission power, if possible, to reduce the transmission radius, r,* while maintaining network connectivity. Shorter r settings reduce the zone membership and the cost of zone maintenance both in terms of packets and transmission power per packet. However, a shorter transmission radius implies a larger network diameter (in hops). And, in turn, this implies greater latencies and more bordercast packets to cover the same area.

This final point raises the question of whether bordercast is necessary at all. If one can set the transmission radius low enough, such that the network degenerates into a tree (i.e., an average of 2 neighbors per node), then bordercast is strictly equal to flooding. However, the average route length along such a tree may increase dramatically. Even at the magic number of 6 spatially separated neighbors per node on average (i.e., approximately 60 degrees from one another) the two protocols will match. In general, the benefits of bordercast stem from its ability to silence neighbors that are not required to propagate the query. If all the neighbors are required to relay the query, due to sparseness of the network, then bordercast will not exceed the performance of flooding. However, lowering the transmission radius is not always possible for a number of reasons. These include fixed hardware power settings, increased probability of link breakage and route failure, overhead of power adaptation protocol, increased number of hidden terminals, increased network diameter, increased average route length and packet latencies, and decreased route diversity. Thus, if the number of neighbors cannot be reduced at the link level, bordercast presents a viable alternative to do so at the network level, preventing unnecessary transmissions and "broadcast storms" during propagation.

6.9 Summary

The scalability of ad hoc networks – the ability to efficiently route and transmit packets across ad hoc networks as they grow in size – is a key research challenge. In this chapter, I have analyzed the cost of discovering a route to some desired destination node using only its unique address. I have evaluated bordercast, a query propagation protocol that is density-independent, and have proven that this is optimal. Bordercast can improve the performance of many existing routing protocols in dense networks by replacing their flooding-based query propagation. These results also show that: optimal bordercast performance does not require zone radii larger than 2 hops; the cost of zone maintenance is proportional to network mobility and bounded; and, that aggregating and efficiently encoding link state updates can substantially reduce the overhead of zone maintenance. The experiments in this chapter, at the scales required to reach the appropriate conclusions, were possible due to the performance and scalability of JiST and SWANS.

Chapter 7

Related Work

The work on JiST, SWANS, and bordercast spans multiple domains of research, including systems, simulation, networking, and languages. This chapter discusses the body of related work in each of these areas.

7.1 Simulation languages

Simulation research has a rich history dating back to the early 60s, when it prompted the development of Simula [30], an Algol-based language that embodied many new design ideas, including object-oriented principles, dynamic binding, and co-routines. It had far-reaching impact on simulation and also throughout the emerging discipline of computer science. Many other simulation languages and systems have since been designed, focusing on performance, distribution, concurrency, speculative execution, and new simulation application domains.

General-purpose simulation languages are often closely related to popular existing languages and contain extensions for events, synchronization, and other simulation primitives. For instance, Csim [99], Yaddes (or Parsimony) [87], Maisie [10], and Parsec [11] are all derivatives of either C or C++ that support process-oriented simulation. Each of these languages is compiled via a C or C++ source-code intermediate, and the resulting executables can be run using a variety of language runtimes. Each of these projects emphasizes different capabilities and ideas. Csim is a commercial product that provides advanced statistics gathering functionality and a library of interesting simulation objects, such as resources, facilities, storages, buffers, mailboxes, etc. Maisie is a general-purpose parallel programming language

that introduces entities, messages, guards, and various synchronization primitives into the syntax of the language. Parsec (for **PAR**allel **S**imulation **E**nvironment for **C**omplex systems) is a follow-on of Maisie that provides more intuitive and flexible syntactic constructs and radically changes the runtime model to execute entities on separate stacks, improving performance by more than an order of magnitude, but at the expense of memory. Finally, Yaddes (for **Y**et **A**nother **D**istributed **D**iscrete **E**vent **S**imulator) defines entities as state machines and the communication channels among them in a declarative, Lex-Yacc-like style. Each of these projects, like JiST, recognizes the benefits of compiling via a standard language compiler. However, all the high-level simulation-specific information is lost after pre-processing, meaning that it is not available for the purposes of either static or dynamic optimization.

Simulation research has also focused on applying object-oriented concepts to simulation-specific problems. Projects, such as Sim++ [8], Pool [4], ModSim II [24], and Moose [108] have investigated various object-oriented possibilities for concurrency, synchronization, and distribution in the context of simulation. Sim++, an extension of the popular, C-based SimPack library, is a sequential, object-oriented C++ library with support for basic simulation functionality, such as events, resources, and statistics gathering. Pool (for **P**arallel **O**bject **O**riented **L**anguage) is a strongly typed, distributed, garbage-collected, synchronous message-passing language. Among other novel ideas, the POOL project investigated the benefits and ramifications of decoupling sub-typing from inheritance. ModSim is a strongly-typed, object-oriented descendant of Simula and Modula, supporting process-oriented simulation, both synchronous and asynchronous events, and numerous object oriented features, including multiple inheritance and polymor-

phism. Finally, the Moose language [108], also a derivative of Modula-2, utilizes inheritance in an interesting way: to specialize implementations of simulation objects with specific knowledge and simulation algorithms that improve execution efficiency. JiST inherits the traditional and popular object-oriented properties of the Java language. It extends the Java object model and execution semantics to support both concurrent object and process-oriented simulation.

Various object-oriented languages, such as Rosette [105] and Act++ [59], structure the concurrency of simulations using actors, as opposed to the traditional concurrent object or process-oriented execution models. The focus of the Rosette work, for example, was on dynamic optimizations within the Actor model [25], providing mechanisms and policies for monitoring and controlling running simulations. While JiST does not support an actor-like functionality, its design does not preclude such an addition.

Other simulation languages, such as Apostle [23] and TeD [83] have taken a more domain-specific language approach. TeD, for example, is an object-oriented language developed mainly for modeling telecommunications network elements and protocols. It supports high-level language constructs for the configurations of routers and switches. OOCSMP [33] is another high-level simulation language designed for continuous models expressed as partial differential equations. While JiST is currently a general-purpose discrete-event simulation platform, the design could certainly subsume domain-specific extensions without loss of generality. Domain-specific extensions could take the form of class libraries or language extensions. In fact, the inherent flexibility of bytecode level rewriter would facilitate the latter approach.

7.2 Simulation libraries

The most popular approach to building simulators involves the use of simulation libraries. A frequently stressed benefit of this approach is that such libraries are usable within existing general-purpose languages, most often C or C++.

Sim++ [8] (mentioned previously) is an object-oriented class library for sequential simulation. Libraries, such as OLPS [1], Speedes [101], and Yansl [58], provide support for parallel event-driven simulation. SimKit [40] is a simulation class library that supports logical processes. And, the Compose [68] simulation library allows individual concurrent objects to dynamically adapt their execution modes among a number of conservative and optimistic synchronization protocols.

With the widespread adoption of Java, there has also been research interest in using this language for simulation, as described in [62]. SimJava [52] and Silk [50] are two early Java-based libraries for process-oriented discrete-event simulation. However, both of these solutions utilize native Java threads within *each* process or entity to capture simulation time concurrency and therefore do not scale. The IDES library [78] is more reasonably designed, but the project was focused on using Java for simplifying distributed simulation and did not address sequential performance. Likewise, the Ptolemy II [17] system provides excellent modeling capabilities, but utilizes a sophisticated component framework that imposes overheads in the critical event dispatch path. Finally, the Dartmouth Scalable Simulation Framework (DaSSF) [64] includes hooks to allow for extensions and event handlers written in Java. These projects combine the features of prior simulation library initiatives with the benefits of the Java environment, such as garbage collection, type safety, and portability.

However, regardless of the language chosen, the primary disadvantage of library-based approaches is that the simulation program becomes more complex and littered with simulation library calls and callbacks. This level of detail not only obscures simulation correctness, but also impedes possible high-level compiler optimizations and program transformations. In other words, library-based approaches lack transparency. Noting the advantages of writing simulators in standard language environments, JiST was designed to work within Java. However, JiST provides its simulation functionality using a language-based approach, rather than via a library.

7.3 Simulation systems

Researchers have built simulations using special operating system kernels that can transparently run processes in virtual time. The landmark simulation kernel work is the TimeWarp OS [54]. Projects such as GTW [32], Warped [67], Parasol [69], and others, have investigated important dynamic optimizations within this model. ParaSol, for example, supports process-oriented simulation over a variety of distributed computing software environments, such as MPI, PVM, P4, and operates both across homogeneous clusters and shared memory multi-processors. Features of Parasol include support for multiple optimistic schedulers, transparent checkpointing of simulation entities, and their transparent migration and location. Simulation kernels associated with specific programming models and language projects, such as Yaddes, Parsec, and others, have already been discussed in section 7.1. JiST provides protection and transparency at finer granularity by using safe language techniques and eliminates the runtime overhead of process-level isolation.

In summary, JiST merges simulation ideas from both the systems and languages camps by leveraging virtual machines as a simulation platform. To the best of my knowledge, JiST is the first system to integrate simulation execution semantics directly into the execution model of a standard virtual machine-based language.

7.4 Languages and Java-related

Java, because of its popularity, has become the focus of much recent research. The Java virtual machine has not only undergone extensive performance work, it has also become the compilation target of many other languages [104]. Projects such as JKernel [49] have investigated the advantages of bringing traditional systems ideas of process isolation and resource accounting into the context of a safe language runtime. The Jalapenõ project [3] has also demonstrated performance advantages of a language-based kernel. Using a common language reduces the boundary-crossing overhead and opens up more opportunities for optimization. JiST makes similar claims and advances in the context of simulation.

A vast number of projects have used Java bytecode analysis and rewriting techniques for a variety of purposes. The Byte-Code Engineering Library [31], Soot [107], and other libraries considerably simplify this task. AspectJ [61] exposes a rewriting functionality directly within the language. Others, including cJVM [6] and Jessica [66], have used Java bytecode rewriting techniques to provide an abstraction of a single-system image abstraction over a cluster of machines. The MagnetOS project [12] has extended this idea to support transparent code migration in the context of an operating system for ad hoc networks [13]. The JavaParty [85] and the xDU [39] projects have looked at similar techniques to perform Java application partitioning and distribution. The JavaGoX [100] and PicoThreads

[14] projects among others, have considered the problem of efficiently capturing stack information without modifying the JVM, as proposed in [20]. KaRMI [84] improves RPC performance using a fast drop-in replacement for Java RMI that uses static bytecode analysis to generate specialized marshaling routines. Finally, [18] and [81] have performed static analysis to determine the mutability of Java objects for a variety of optimizations. JiST brings these and other ideas to bear on the problem of high-performance simulation.

7.5 Network simulation

The networking community depends heavily on simulation to validate its research. The ns2 [70] network simulator has had a long history with the community and is widely trusted. It was therefore extended to support mobility and wireless protocols [56]. Though it is primarily used sequentially in the community, researchers have extended ns2 to PDNS [93], allowing for conservative parallel execution. GloMoSim [111] is a newer simulator written in Parsec [11] 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 [88]. Another notable and commercially-supported network simulator is OPNet [79]. Recently, the Dartmouth Scalable Simulation Framework (DaSSF) has also been extended with SWAN[1] [65] to support distributed wireless ad hoc network simulations. Likewise, TeD [83] has been extended with WiPPET [60], though it is focused on cellular networks. SWiMNet [21] is another parallel wireless simulator focused on cellular networks.

[1]Not to be confused with SWANS.

I have described JiST running wireless network simulations of one million nodes on a *single* commodity machine with a 2.2GHz processor and 2GB of RAM. Running a more realistic networking protocol, such as ZRP, requires more memory per node. JiST was able to scale to network models of 500,000 ZRP nodes on a 4GB machine running a hugemem Linux kernel. To the best of my knowledge, this exceeds the performance of every existing sequential network simulator. Based on the literature, this scale of network is beyond the reach of many parallel simulators, even when using more hardware. Clearly, memory consumption depends on what is being simulated, not just on the number of nodes in the network. For example, in the NDP simulation the state of the entire stack of each node consumes less than 1K of memory, but this will clearly increase if additional simulation components are added. Likewise, simulation performance depends on network traffic, node density and many other parameters. Therefore, merely as a point of reference, [92] summarizes the state of the art in 2002 as follows: using either expensive multi-processor machines or clusters of commodity machines connected with fast switches, DaSSF, PDNS, and WARPED can simulate networks of around 100,000 nodes, while TeD and GloMoSim have shown results with 10,000 node networks. More recently, the PDNS website [91] states that "PDNS has been tested on as many as 136 processors simulating a 600,000+ node network", but without further details. This same group continues to push the envelope of parallel and distributed simulation further still [35], with GTNetS [90].

Various projects, including EmuLab [109], ModelNet [72] and PlanetLab [86] provide alternatives to simulation by providing emulation and execution test-beds. JiST simulation is complementary to these approaches. However, the ability to

efficiently run standard network applications over simulated networks within JiST blurs the distinction between simulation and emulation.

Finally, J-Sim (JavaSim) [106] is a relatively new and highly-optimized sequential network simulator written in Java, using a library that supports the construction of simulations from independent `Component`s with `Port`s that are connected using `Wire`s. The system is intelligently designed to reduce threading overhead, synchronization costs, and message copying during event dispatch, resulting in performance only just slightly slower than JiST. However, the memory overhead for the various JavaSim infrastructure objects, results in a memory footprint that is larger than JiST by an order of magnitude for network models of equal size.

7.6 Wireless ad hoc networking

Chapter 6 was dedicated to the analysis of the bordercast query propagation algorithm at large scale. Much prior research on ad hoc networks has focused on multi-hop routing algorithms, such as DSR [57], AODV [82], TORA [80], TBRPF [15], OLSR [29], and many others. Each of these is designed and optimized under different networking assumptions. Many such protocols can be made more scalable by replacing their flood-based query propagation mechanisms with bordercast, as in ZRP [48]. Excellent surveys include [95] and [22], and [2] for sensors networks specifically.

The broadcast storm problem in mobile ad hoc networks [75] has been investigated, with numerous solutions proposed and compared [74, 110]. Specifically, various protocols based on distributed approximate connected dominating set algorithms [44] are surveyed in [103]. Recently, probabilistic broadcast schemes [47, 98] have also been proposed to overcome the same problem. I have shown that border-

cast, a query propagation protocol that predates all of this work, actually already has density-independent properties, and have performed experiments at scales that were previously not possible.

Chapter 8

Conclusion

8.1 Summary

In this dissertation, I have proposed and advocated for virtual machine-based simulation, a new, unifying approach to building simulators. I have outlined the rationale for this new design and contrasted it with the existing language-based and systems-based approaches to building simulators.

In particular, I have introduced the JiST prototype, a new general-purpose Java-based simulation platform that embodies the virtual machine-based simulator design. JiST embeds simulation execution semantics directly into the Java virtual machine. The system provides all the standard benefits that the modern Java runtime affords. In addition, JiST is efficient, out-performing existing highly optimized simulation runtimes, and inherently flexible, capable of transparently performing cross-cutting program transformations and optimizations. I have leveraged this flexibility to introduce additional concepts into the JiST model, including process-oriented simulation and simulation time concurrency primitives.

Finally, I have constructed SWANS, a wireless ad hoc network simulator, atop JiST, as a validation of the JiST approach, and have demonstrated that SWANS can scale to wireless network simulations of a million nodes even on a single commodity machine. I have also utilized SWANS to perform a scalable analysis of the bordercast query propagation protocol, showing that it is density-independent.

8.2 Future work

The JiST work can be extended in a number of research and engineering directions:

- *Parallelism* - As discussed in section 3.7, the current implementation has focused exclusively on sequential performance. JiST, however, was explicitly designed and implemented with parallelism in mind. It would be natural to extend the kernel to allow multiple `Controller`s to operate concurrently. It should be relatively easy to leverage the full processing power of commodity multi-processor machines.

- *Distributed simulation* - JiST was also designed with distributed simulation in mind. Entity separators can transparently support a single system abstraction by tracking entity locations as they are dynamically migrated across a cluster to balance computational and network load. The JiST kernel should be extended to support conservatively synchronized, distributed, cooperative operation with peer JiST kernels. This work would increase the available simulation memory and allow larger models to be processed. Interesting issues to address include efficient serialization and migration algorithms.

- *Speculative execution* - The cost of synchronization is critical to the performance of a distributed simulator. As discussed in section 3.7, JiST can already transparently support both checkpointing and rollback of entities. Speculative execution poses a rich space of design and research problems. It would be interesting, for example, to investigate possible interactions between the GVT-scheduler and the local garbage collector, as well as different checkpointing or rollback schemes that are tailored through static bytecode analysis and simulation profiles.

- *Simulation research platform* - The bytecode level rewriter component within the JiST design represents a point of flexibility, wherein much additional functionality may be inserted. In addition to the classic simulation extensions just described, and much like the manner in which continuations were implemented in section 3.5, the rewriter can be used to introduce new research ideas into a existing base of simulations. One such example mentioned in section 3.8 is reverse compilation. In general, the rewriter permits cross-cutting simulation transformations and a separation of concerns that makes JiST an attractive platform for ongoing simulation research.

- *Simulation-specific language extensions* - While it was decided, for software-engineering reasons, that JiST-based simulations should be compiled with a standard Java compiler, this restriction could be relaxed to allow for more complex syntactic structures that may be required by simulations. Complex simulation constructs may be difficult or impossible to express within the Java language syntax. An extended Java-like language may be implemented either as a text pre-processor or as a bytecode-generating compiler. Even relatively simple JiST constructs such as:

  `class Foo implements JistAPI.Entity {`

  could then be written more elegantly as:

  `entity Foo {`.

  Also, more complex language constructs, such as Parsec-like message guards, come to mind. However, their utility, too, is questionable. Static bytecode analysis could readily determine message guards from early termination conditions within the method body. Nevertheless, the space of language extensions is a rich, unexplored research area.

- *Simulation-specific virtual machine extensions* - As discussed in section 5.7, JiST performance suffers from a number of Java design decisions. Specifically, the virtual machine does not handle reflection of primitives types efficiently and lacks an API for type-safe stack access. While both of these issues are being addressed for mainstream reasons, there may be other simulation-specific optimizations that are possible only through modifications to the virtual machine.

- *Java-like simulation time synchronization* - Mapping Java synchronization primitives – `wait()`, `notify()`, `synchronized` – to simulation time equivalents would allow a larger set of Java applications to be embedded within various JiST-based simulators.

- *Declarative simulation specifications* - JiST simulators tend naturally to be component-oriented and simulations are often configured as graphs of connected entities that have highly compressible structure. It would be useful to construct such entity graphs via a declarative specification, rather than the current approach of using imperative driver programs.

- *New simulation primitives* - JiST currently supports a few high-level simulation primitives, such as `Channel`s and `Thread`s. A library of well-known simulation primitives commonly found in other simulation systems can be implemented.

- *New simulation domains and applications* - The functionality of SWANS can be further expanded by the networking research community. Simulators in other application domains should also be developed.

- *Debugging, or interactive simulation* - One of the significant advantages of leveraging the Java language and runtime is the ability to adopt existing Java tools, such as debuggers, often lacking in simulation environments. Event-driven programs are particularly difficult to debug, compounding the problem. An existing Java debugger could readily be extended to understand simulation events and other kernel data structures, resulting in functionality that is unparalleled in any existing simulation environment. Since Java is a reflective language, JiST simulations may be paused, modified in-flight, and then resumed. The appropriate tools to perform such inspection effectively would facilitate interactive simulation and present interesting opportunities. For example, one could use the debugger to control the distributed simulation kernel and the GVT-scheduler, not only to obtain consistent cuts of the simulation state, but also to permit stepping *backwards* in simulation time to understand root causes of a particular simulation state.

To conclude, I hope that the performance of JiST, its ability to merge ideas from the systems-oriented and the language-oriented approaches to simulation, its flexibility and utility as a simulation research platform, and the popularity of the Java language will facilitate its broader adoption within the simulation community.

Appendix A

The JiST API

Many of the important kernel functions have already been mentioned. For completeness, the full JiST application interface (listed partially in Figure 2.3) is included and described 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`.

Continuation exception - tags an entity method as blocking, which means that these entity method invocations will be performed in simulation time, but with continuation.

e.g. `jist.swans.app.UdpSocket.receive(DatagramPacket)`.

Continuable exception - explicitly tags a regular method as possibly blocking, useful only for instances when static analysis can not propagate the blocking property to all callers due to dynamic dispatch.

e.g. the abstract method: `jist.swans.app.io.InputStream.read()`.

Timeless interface - explicitly tags a simulation object as timeless, which means that it will not be changed across simulation time and thus need not be copied when transferred among entities.

e.g. `jist.swans.node.Message`.

getTime() - returns the local simulation time. The local time is the time of the current event being processed plus any additional `sleep` time requested during the processing of the current event.

END - Simulation end time constants, greater than any legal simulation time.

sleep(time) - advance the local simulation time.

end() - end simulation after the current time-step.

endAt(time) - end simulation after given absolute time.

callStaticAt(method, params, time) - invoke a static method at given simulation time.

runAt(runnable, time) - invoke a runnable object at given simulation time.

THIS - entity self-referencing separator, analogous to Java `this` object self-reference. Should be type-cast before use.

ref(entity) - returns a separator of a given entity. All statically detectable entity references are automatically converted into separator stubs by the rewriter, so this operator should not be needed. It is included only to deal with rare instances of creating entity types dynamically and for completeness.

isEntity(o) - returns whether given reference is an entity reference.

toString(o) - returns the string representation of an object or entity.

Proxiable interface - an interface used to tag objects that may be proxied. It serves to improve proxying performance by eliminating the need for a relaying wrapper entity.

proxy(target, interface) - Returns a proxy separator for the given target object and interface class. The proxying approach depends on whether the target is an existing entity, a regular object or a proxiable object. The result is an object whose methods will be relayed to the target in simulation time.

proxyMany(target, interface[]) - Same as proxy call and allows for multiple interfaces.

run(type, name, args, prop) - Start a new simulation with given name and arguments at the current time. The supported simulation loader types are Java applications (RUN_CLASS), BeanShell scripts (RUN_BSH), and Jython scripts (RUN_JPY). The properties object carries simulation type-specific information directly to the simulation loader.

createChannel() - Create a new CSP Channel Entity.

setSimUnits(ticks, name) - Set the simulation time unit of measure and length in simulation ticks. The default is 1 tick.

getTimeString() - Return time string in simulation time units.

CustomRewriter interface - Defines an installable rewriting phase.

installRewriter(rewriter) - Installs a custom class rewriting phase at the beginning of the JiST rewriting machinery. Used for simulation specific rewriting needs. For example, SWANS uses this interface to rewrite the networking library calls of applications to operate over the simulated network.

Logger interface - Defines an custom event logger.

setLog(logger) - Set the simulation logger.

log(object) - Log an object, usually a string.

Appendix B

SWANS Components

This appendix enumerates the various SWANS components that are available (as of this writing). Additional components can be readily implemented. Users are encouraged to contribute components, either with or without source, to the research community.

B.1 Physical

The SWANS physical layer components are responsible for modeling signal propagation among radios as well as the mobility of nodes. Radios make transmission down-calls to the simulation "field" and other radios on the "field" receive reception up-calls from it, if they are within range of the signal. Both the path loss and fading models are functions that depend on the source and destination radio locations. Path loss models include free-space, two-ray and table-driven path loss. Fading models include none, Raleigh and Rician fading. Node mobility is implemented as an interface-based discretized model. Upon each node movement, the model is queried to schedule the next movement. The mobility models that are implemented include static and random-waypoint.

jist.swans.field.*

interface	**class**	**description**
FieldInterface	Field	centralized node container that performs mobility and signal propagation with fading and path loss
Fading	Fading.None	zero fading model

jist.swans.field.*

interface	**class**	**description**
	Fading.Raleigh	Raleigh fading model
	Fading.Rician	Rician fading model
Pathloss	Pathloss.FreeSpace	path loss model based purely on distance
	Pathloss.TwoRay	path loss model that incorporates ground reflection
Spatial	Spatial.Linear	signal propagation and location update performed via linked list of radios
	Spatial.Grid	as above, but performed using a more efficient flat grid structure of small "Linear" bins
	Spatial.HierGrid	as above, but performed using a more consistently efficient hierarchical grid structure
	...TiledWraparound	tile inner spatial structure in 3x3 grid so as to wrap field edges around into a torus
Placement	Placement.Random	uniformly random initial node placement
Mobility	Mobility.Static	no mobility
	...RandomWaypoint	pick a random "waypoint" and walk towards it with some random velocity, then pause and repeat.

jist.swans.field.*

interface	**class**	**description**
	...RandomWalk	pick a direction, walk a certain distance in that direction, with some fixed and random component, reflecting off walls as necessary, then pause for some time and repeat.
	Mobility.Teleport	pick a random location and teleport to it, then pause for some time, and repeat.

The SWANS radio receives up-calls from the field entity and passes successfully received packets on to the link layer. It also receives down-calls from the link layer entity and passes them on to the field for propagation. An implementation of an independent interference radio exists, as in ns2, as well as an additive interference radio, as in GloMoSim. The independent interference model considers only signals destined for the target radio as interference. The additive model correctly considers all signals as contributing to the interference. Both radios are half-duplex, as in 802.11b. Radios are parameterized by frequency, transmission power, reception sensitivity and threshold, antenna gain, bandwidth and error model. Error models include bit-error rate and signal-to-noise threshold.

jist.swans.radio.*

interface	**class**	**description**
RadioInterface	RadioNoiseIndep	interference at radio consists only of other signals above a threshold that are destined for that same radio

jist.swans.radio.*

interface	**class**	**description**
	RadioNoiseAdditive	interference consists of all signals above a threshold
none	RadioInfo	unique and shared radio parameters

B.2 Link

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

jist.swans.mac.*

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

B.3 Network

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

jist.swans.net.*

interface	**class**	**description**
NetInterface	NetIp	IPv4 implementation
MessageQueue	NoDrop	prioritized, no drop IP message queue
PacketLoss	Zero	zero network layer packet loss
	Uniform	independent, random drop with fixed probability

jist.swans.net.*

interface	class	description
none	NetAddress	network address

B.4 Routing

The routing entity recieves up-calls from the network entity with packets that require next-hop information. It also receives up-calls that allow it to peek at all packets that arrive at a node. It sends down-calls to the network entity with next-hop information when it becomes available. SWANS implements the Zone Routing Protocol (ZRP) [46], Dynamic Source Routing (DSR) [57] and Ad hoc On-demand Distance Vector Routing (AODV) [82].

jist.swans.route.*

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

B.5 Transport

The SWANS transport entity receives up-calls from the network entity with packets of the appropriate network protocol and passes them on to the appropriate registered transport protocol handler. It also receives down-calls from the application entity, which it passes on to the network entity. The two implemented transport protocols are UDP and TCP, which encapsulate packets with the appropriate

packet headers. UDP socket, TCP socket and TCP server socket implementations actually exist within the application entity. The primary reason for this decision is that these implementations are modeled after corresponding Java classes, which force the use non-timeless objects. The `DatagramSocket`, for example, uses a mutable `DatagramPacket` to provide data. In all other respects, including correctness and performance, this decision, to move the socket implementations into the application entity, is inconsequential.

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

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

jist.swans.trans.*

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

B.6 Application

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

jist.swans.app.*

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

B.7 Common

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

jist.swans.misc.*

interface	**package**	**implementations**
Message	jist.swans.misc	MessageBytes, MessageNest

jist.swans.misc.*

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

Appendix C

Event Micro-benchmarks

This section provides simplified versions of the benchmark programs used to measure the event throughput of the JiST, Parsec, GloMoSim, and ns2 systems in section 5.2. To the best of my 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 timing the run. They closely reflect the style of code that a simulation developer would generate.

C.1 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();
 }
}
```

C.2 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.3 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);
}
```

C.4 ns2-C

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
```

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_--;
 }
}
```

BIBLIOGRAPHY

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

[2] Akyildiz, I. F., Su, W., Sankarasubramaniam, Y., and Cayirci, E. Wireless sensor networks: A survey. *Computer Networks 38*, 4 (2002), 393–422.

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

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

[5] Anderson, D., Balakrishnan, H., Kaashoek, F., and Morris, R. Resilient overlay networks. In *ACM Symposium on Operating Systems Principles* (Oct. 2001).

[6] Aridor, Y., Factor, M., and Teperman, A. cJVM: A single system image of a JVM on a cluster. In *International Conference on Parallel Processing* (Sept. 1999).

[7] Arpacioglu, O., Small, T., and Haas, Z. J. Notes on scalability of wireless ad hoc networks. `http://wnl.ece.cornell.edu/Publications/draft-irtf-ans-scalability-defi%nition-01.txt`, Dec. 2003.

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

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

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

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

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

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

[14] Begel, A., MacDonald, J., and Shilman, M. PicoThreads: Lightweight threads in Java. Tech. rep., UC Berkeley, 2000.

[15] Bellur, B., and Ogier, R. G. A reliable, efficient topology broadcast protocol for dynamic networks. In *IEEE INFOCOM '99* (Mar. 1999).

[16] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *ACM Symposium on Operating Systems Principles* (Dec. 1995).

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

[18] Biberstein, M., Gil, J., and Porat, S. Sealing, encapsulation, and mutability. In *European Conference on Object-Oriented Programming* (June 2001), pp. 28–52.

[19] Booth, C. J. M., and Bruce, D. I. Stack-free process-oriented simulation. In *Workshop on Parallel and Distributed Simulation* (June 1997), pp. 182–185.

[20] Bouchenak, S., and Hagimont, D. Zero overhead java thread migration. Tech. Rep. 0261, INRIA, 2002.

[21] Boukerche, A., Das, S. K., and Fabbri, A. SWiMNet: A scalable parallel simulation testbed for wireless and mobile networks. *Wireless Networks 7* (2001), 467–486.

[22] Broch, J., Maltz, D. A., Johnson, D. B., Hu, Y.-C., and Jetcheva, J. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Mobile Computing and Networking* (Oct. 1998), pp. 85–97.

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

[24] Bryan, Jr., O. Modsim II - an object-oriented simulation language for sequential and parallel processors. In *Winter Simulation Conference* (Dec. 1989), pp. 122–127.

[25] Byrd, R. J., Smith, S. E., and deJong, S. P. An actor-based programming system. In *Proceedings of the SIGOA Conference on Office information systems* (1982), pp. 67–78.

[26] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. Efficient optimistic parallel simulations using reverse computation. In *Workshop on Parallel and Distributed Simulation* (May 1999), pp. 126–135.

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

[28] Chandy, K. M., and Sherman, R. The conditional event approach to distributed simulation. In *Distributed Simulation Conference* (1989).

[29] Clausen, T., Jacquet, P., Laouiti, A., Muhlethaler, P., Qayyum, A., and Viennot, L. Optimized link state routing protocol. In *IEEE INMIC* (2001).

[30] Dahl, O.-J., and Nygaard, K. Simula, an Algol-based simulation language. *Communications of the ACM* (1966), 671–678.

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

[32] Das, S. R., Fujimoto, R. M., Panesar, K. S., Allison, D., and Hybinette, M. GTW: A time warp system for shared memory multiprocessors. In *Winter Simulation Conference* (Dec. 1994), pp. 1332–1339.

[33] de Lara, J., and Alfonseca, M. Visual interactive simulation for distance education. *Simulation: Transactions of the Society for Modeling and Simulation International 1*, 79 (2003), 19–34.

[34] Default TTL Values in TCP/IP. `http://secfr.nerim.net/docs/fingerprint/en/ttl_default.html`.

[35] Fujimoto, R., Riley, G., Perumalla, K., Park, A., Wu, H., and Ammar, M. Large-scale network simulation: how big? how fast? In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication* (Oct. 2003).

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

[37] Fujimoto, R. M. Parallel and distributed simulation. In *Winter Simulation Conference* (Dec. 1995), pp. 118–125.

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

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

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

[41] Gomes, F., Unger, B., and Cleary, J. Language-based state-saving extensions for optimistic parallel simulation. In *Winter Simulation Conference* (Dec. 1996), pp. 794–800.

[42] Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*. Addison-Wesley, 1996.

[43] Grossglauser, M., and Tse, D. Mobility increases the capacity of wireless adhoc networks. *IEEE/ACM Transactions on Networking 10* (Aug. 2002), 477–486.

[44] Guha, S., and Khuller, S. Approximation algorithms for connected dominating sets. *Algorithmica 20* (1998), 347–387.

[45] Gupta, P., and Kumar, P. The capacity of wireless networks. *Trans. on Info. Theory 46* (Mar. 2000), 388–404.

[46] Haas, Z. J. A new routing protocol for the reconfigurable wireless networks. In *IEEE Conference on Universal Personal Comm.* (Oct. 1997).

[47] Haas, Z. J., Halpern, J. Y., and Li, L. Gossip-based ad hoc routing. In *IEEE INFOCOM '02* (June 2002).

[48] Haas, Z. J., and Pearlman, M. R. The performance of query control schemes for the Zone Routing Protocol. *IEEE/ACM Transactions on Networking 9*, 4 (Aug. 2001), 427–438.

[49] Hawblitzel, C., Chang, C.-C., Czajkowski, G., Hu, D., and von Eicken, T. Implementing multiple protection domains in Java. In *USENIX Annual Technical Conference* (June 1998), pp. 259–270.

[50] Healy, K. J., and Kilgore, R. A. Silk : A Java-based process simulation language. In *Winter Simulation Conference* (Dec. 1997), pp. 475–482.

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

[52] Howell, F., and McNab, R. SimJava: A discrete event simulation library for Java. In *International Conference on Web-Based Modeling and Simulation* (Jan. 1998).

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

[54] Jefferson, D. R., and et. al. Distributed simulation and the Time Warp operating system. In *ACM Symposium on Operating Systems Principles* (Nov. 1987), pp. 77–93.

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

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

[57] Johnson, D. B., and Maltz, D. A. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*. Kluwer Academic Publishers, 1996.

[58] Joines, J. A., and Roberts, S. D. Design of object-oriented simulations in C++. In *Winter Simulation Conference* (Dec. 1994), pp. 157–165.

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

[60] Kelly, O., Lai, J., Mandayam, N. B., Ogielski, A. T., Panchal, J., and Yates, R. D. Scalable parallel simulations of wireless networks with WiPPET. *Mobile Networks and Applications 5*, 3 (2000), 199–208.

[61] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-oriented programming. *European Conference on Object-Oriented Programming 1241* (1997), 220–242.

[62] Kilgore, R. A., Healy, K. J., and Kleindorfer, G. B. The future of Java-based simulation. In *Winter Simulation Conference* (Dec. 1998), pp. 1707–1712.

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

[64] Liu, J., and Nicol, D. M. Dartmouth Scalable Simulation Framework (DaSSF) 3.1 user's manual, Apr. 2001.

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

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

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

[68] Martin, J. M., and Bagrodia, R. L. Compose: An object-oriented environment for parallel discrete-event simulations. In *Winter Simulation Conference* (Dec. 1995), pp. 763–767.

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

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

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

[72] Modelnet. `http://issg.cs.duke.edu/modelnet.html`.

[73] Naoumov, V., and Gross, T. Simulation of large ad hoc networks. In *ACM MSWiM* (2003), pp. 50–57.

[74] Ni, S., Tseng, Y., and Sheu, J. Efficient broadcasting in a mobile ad hoc network. In *IEEE International Conference of Distributed Computing Systems* (Apr. 2001), pp. 16–19.

[75] Ni, S.-Y., Tseng, Y.-C., Chen, Y.-S., and Sheu, J.-P. The broadcast storm problem in a mobile ad hoc network. In *ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)* (Aug. 1999), ACM Press, pp. 151–162.

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

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

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

[79] Opnet. `http://www.opnet.com/`.

[80] Park, V. D., and Corson, M. S. Temporally-ordered routing algorithm. Internet Draft, Aug. 1998.

[81] Pechtchanski, I., and Sarkar, V. Immutability specification and its applications. In *Java Grande* (Nov. 2002).

[82] Perkins, C. E., and Royer, E. M. Ad hoc on-demand distance vector routing. In *Workshop on Mobile Computing Syst. and Apps.* (Feb. 1999), pp. 90–100.

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

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

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

[86] Planetlab. `http://www.planet-lab.org/`.

[87] Preiss, B. R. The Yaddes distributed discrete event simulation specification language and execution environment. In *SCS Multiconference on Distributed Simulation* (1989), pp. 139–144.

[88] Qualnet. `http://www.scalable-networks.com/`.

[89] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. A scalable content addressable network. In *SIGCOMM* (2001).

[90] Riley, G. The Georgia Tech Network Simulator. In *SIGCOMM Workshop on Models, methods and tools for reproducible network research* (2003), pp. 5–12.

[91] Riley, G. PDNS, July 2003. `http://www.cc.gatech.edu/computing/compass/pdns/`.

[92] Riley, G., and Ammar, M. Simulating large networks: How big is big enough? In *Conference on Grand Challenges for Modeling and Sim.* (Jan. 2002).

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

[94] Rowstron, A., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *ACM Conference on Distributed Systems Platforms* (Nov. 2001), pp. 329–350.

[95] Royer, E., and Toh, C.-K. A review of current routing protocols for ad-hoc mobile wireless networks. *IEEE Personal Comm.* (Apr. 1999), 46–55.

[96] Sakamoto, T., Sekiguchi, T., and Yonezawa, A. Bytecode transformation for portable thread migration in Java. In *International Symposium on Mobile Agents* (2000).

[97] Samar, P., Pearlman, M., and Haas, Z. Independent zone routing: An adaptive hybrid routing framework for ad hoc wireless networks. *IEEE/ACM Transactions on Networking* (August 2004 (expected)).

[98] Sasson, Y., Cavin, D., and Schiper, A. Probabilistic broadcast for flooding in wireless mobile ad hoc networks. In *IEEE Wireless Communications and Networking Conference* (Mar. 2003).

[99] Schwetman, H. Csim18 - the simulation engine. In *Winter Simulation Conference* (Dec. 1996), pp. 517–521.

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

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

[102] Stoica, I., Morris, R., Karger, D., Kaashoek, F., and Balakrishnan, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on applications, technologies, architectures, and protocols for computer communications* (2001), pp. 149–160.

[103] Stojmenovic, I., Seddigh, M., and Zunic, J. Dominating sets and neighbor elimination-based broadcasting algorithms in wireless networks. *IEEE Transactions on Parallel and Distributed Systems 13*, 1 (2002), 14–25.

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

[105] Tomlinson, C., and Singh, V. Inheritance and synchronization in enabled-sets. In *Object-Oriented Programming Systems, Languages and Applications* (Oct. 1989), pp. 103–112.

[106] Tyan, H.-Y., and Hou, C.-J. JavaSim: A component based compositional network simulation environment. In *Western Simulation Multiconference* (Jan. 2001).

[107] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., and Co, P. Soot - a Java optimization framework. In *CASCON* (1999), pp. 125–135.

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

[109] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *ACM Symposium on Operating Systems Design and Implementation* (Dec. 2002).

[110] Williams, B., and Camp, T. Comparison of broadcasting techniques for mobile ad hoc networks. In *ACM International Symposium on Mobile Ad Hoc Networking and Computing (MOBIHOC)* (2002), pp. 194–205.

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

[112] Zhao, B., Kubiatowicz, J., and Joseph, A. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Tech. Rep. UCB/CSD-01-1141, UC Berkeley, Apr. 2001.