Contents

Chapter 1

Scalable Wireless Ad hoc Network Simulation

Rimon Barr, Zygmunt J. Haas, Robbert van Renesse

Computer Science and Electrical Engineering,
Cornell University, Ithaca NY 14853
{barr@cs, haas@ece, rvr@cs}.cornell.edu

Ongoing research into dynamic, self-organizing, multi-hop wireless networks, called ad hoc networks, promises to improve the efficiency and coverage of wireless communication. Such 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. However, the ability to scale such networks to large numbers of nodes remains an open research problem. For example, routing and transmitting packets efficiently over ad hoc networks becomes difficult as they grow in size.

Progress in this area of research is fundamentally dependent on the capabilities of simulation tools and, more specifically, on the scalability of wireless network simulators. Analytically quantifying the performance and complex behavior of even simple protocols in the aggregate is often imprecise. Furthermore, performing actual experiments is onerous: acquiring hundreds of devices, managing their software and configuration, controlling a distributed experiment and aggregating the data, possibly moving the devices around, finding the physical space for such an experiment, isolating it from interference and generally ensuring *ceteris paribus* are but some of the difficulties that make empirical endeavors daunting. Consequently, the vast majority of research in this area is based entirely upon simulation, a fact that underscores the critical role of efficient simulators.

1.1 Background

Discrete event simulators have been the subject of much research into their efficient design and execution (surveyed in [17, 8, 20, 9]). However, despite a plethora of ideas and contributions to theory, languages, and systems, slow sequential simulators remain the norm [21]. In particular, most published ad hoc network results are based on simulations of few nodes, for a short time duration, and over a limited size coverage 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. At a minimum, one would like to simulate networks of many thousands of nodes.

The two most popular simulators in the wireless networking research area are ns2 and GloMoSim. The ns2 network simulator [16] has a long history with the networking community, is widely trusted, and has been extended to support mobility and wireless networking protocols. 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 substantial memory overhead and increases the complexity of simulation code. Researchers have extended ns2 to conservatively parallelize its event loop [22], but 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 substantial hardware resources and effort, to simulations of a few thousand nodes [21].

GloMoSim [27] is a newer simulator written in Parsec [3], 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. Moreover, the aggregation of state also renders the lookahead techniques impractical, as has been noted by the authors. GloMoSim has been shown to scale to 10,000 nodes on large, multi-processor machines.

In this chapter, we describe the design of SWANS, a new **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator. SWANS is a componentized, virtual machine-based simulator [5] built atop the JiST (**J**ava **i**n **S**imulation **T**ime) platform, a general-purpose discrete event simulation engine. SWANS significantly out-performs ns2 and GloMoSim both in time and space. We show results with networks that are more than an order of magnitude larger than what is possible with the existing tools at the same level of simulation detail. SWANS can also, unlike any existing network simulator, efficiently embed existing network applications and run them over simulated networks.

1.2 Design highlights

The SWANS software is organized as independent software components, called *entities*, that can be composed to form complete wireless network or sensor network simulations, as shown in Figure 1.1. Its capabilities are similar to ns2 [16] and GloMoSim [27] described above. There are entities 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 type of entity are shown italicized in Figure 1.1.

The SWANS simulator runs atop JiST, a Java-based discrete-event simulation engine that combines the benefits of the traditional systems-based (e.g., ns2) and languages-based (e.g., GloMoSim) approaches to simulation construction. JiST converts a standard virtual machine into a simulation platform by embedding simulation time directly into the Java object model and into the virtual machine execution semantics. Thus, one can write a simulator in a standard systems language (i.e. Java) and transparently perform optimizations and cross-cutting program transformations that are found in specialized simulation languages. JiST extends the Java object model with the notion of simulation entities. The simulation entities represent components of a simulation that can progress independently through simulation time, each encapsulating a disjoint subset of the simulation state. Simulation events are intuitively represented as method invocations across entities. This programming model is convenient, efficient, and flexible. We encourage the interested reader to further learn about JiST through the documentation and software available online [4].

SWANS is able to simulate much larger networks and has a number of other advantages over existing tools. We leverage the JiST design within SWANS to: 1. achieve high simulation throughput; 2. reduce its memory footprint; and 3. run existing, Java-based network applications over simulated networks. In addition, SWANS implements a technique called hierarchical binning to model wireless signal propagation in a scalable manner. The combination of these attributes leads to a flexible and highly efficient simulator. We discuss each of these concepts in turn, and then conclude with a discussion of possible directions for future work.

We shall limit our discussion in the following sections to techniques implemented within SWANS for maximizing *sequential* simulation performance. Others, in projects such as PDNS [22], SWAN-DaSSF [15], WiPPET-TeD [12] and SWiMNet [7], have presented algorithms and techniques to achieve scalability through distributed, concurrent, and even speculative simulation execution. These techniques can sometimes provide around an order of magnitude improvement in scale, but may require multi-processor hardware or fast inter-connects to reduce synchronization costs. More importantly, such techniques are orthogonal to the ideas presented here. A truly scalable network simulator requires raw sequential performance as well as effective distribution and parallelism.

1.3 Throughput

Conventional wisdom regarding language performance [2] argues against implementing a simulator in Java. In fact, the vast majority of existing simulators have been written in C and C++, or their derivatives. SWANS, however, performs surprisingly well: aggressive profile-

driven optimizations combined with the latest Java runtimes result in a high-performance system.

We selected to implement JiST and SWANS in Java for a number of reasons. Firstly, Java is a standard, widely deployed language and 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 [26] 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.

In addition, the following are some aspects of SWANS and of the underlying JiST design that contribute to its high computational throughput:

- **dynamic compilation** – The simulator runtime, which is a standard Java virtual machine, continuously profiles running simulations and dynamically performs important code optimizations, such as constant propagation and function inlining. Dynamic optimizations provide significant performance benefits, because many stable simulation parameters are not known until the simulation is running. In general, a dynamic compiler has more information available to it than a static compiler and should, therefore, produce better code. Despite virtual machine overheads for profiling, garbage collection, portability, runtime safety checks, and dynamic compilation, significant speedups can be achieved. For example, greater than $10\times$ speedups have been observed within the first few seconds of simulation execution.
- **code inlining** – Long running simulations often exhibit tight computation loops through only a small fraction of the code. The dynamic compiler can aggressively inline these "hot spots" to eliminate function calls in the generated code. Since both the simulator, SWANS, and the simulation kernel, JiST, are both written in Java, inlining can occur both within simulation entities and also across the simulation kernel-entity boundary. 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. A similar idea was first demonstrated by the Jalapeño project [1]
- **no context switch** – JiST provides protection and isolation for individual simulation entities from one another and from the simulation kernel. However, this separation is achieved using safe language techniques, eliminating the runtime overhead of traditional process-based isolation. A simulation event between two co-located source and target entities can be dispatched, scheduled, delivered, and processed without a single context switch.
- **no memory copy** – Each JiST entity has its own, independent simulation time and state. Therefore, to preserve entity isolation, any mutable state transferred via simulation events across entity boundaries must be passed by copy. However, temporally stable objects are an exception to this rule. These objects may safely be passed across entities by reference. To that end, JiST defines the notion of a *timeless* object as one that will not change over time. This property may either be automatically inferred

through static analysis or specified explicitly. In either case, the result is zero-copy semantics and increased event throughput.

- **cross-cutting program transformations** – The timeless property just introduced is an apt example of a cross-cutting optimization: the addition of a single tag, or the automatic detection of the timeless property, affects all events within the simulation that contain objects of this type. Similarly, the design of JiST entities abstracts event dispatch, scheduling, and delivery. Thus, the implementations of this functionality can be transparently modified. In general, the JiST bytecode-level rewriting phase that occurs at simulation load time permits a large class of transparent program transformations and simulation-specific optimizations, akin to aspect-oriented programming [13].

High event throughput is essential for scalable network simulation. Thus, we present results showing the raw event throughput of JiST versus competing simulation engines. These measurements were all 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. We 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.

The performance of the simulation engines was measured in performing a tight simulation event loop, using equivalent, efficient benchmark programs written for each of the engines. The results are plotted in Figure 1.2 on log-log and linear scales. As expected, all the simulations run in linear time with respect to the number of events. A counter-intuitive result is that JiST, running atop Java, out-performs all the other systems, including the compiled ones.

We, therefore, added a reference measurement to serve as a computational lower bound. This reference is a program, written in C and compiled with `gcc -O3`, which merely inserts and removes elements from an efficient implementation of an array-based priority queue. JiST comes within 30% of this reference measurement, an achievement that is a testament to the impressive JIT dynamic compilation and optimization capabilities of the modern Java runtime. Furthermore, the performance impact of these optimizations can actually be seen as a kink in the JiST curve during the first fraction of a second of the simulation. To confirm this, JiST was warmed with 10^6 events (or, for two tenths of a second) and the kink disappears. As seen on the linear plot, the time spent on profiling and dynamic optimizations is negligible. Table 1.1 shows the time taken to perform 5 million events in each of the benchmarked 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, and GloMoSim and ns2-Tcl are one and two orders of magnitude slower, respectively.

SWANS builds on the performance of JiST. We benchmark SWANS running a full ad hoc wireless network simulation, 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 [24, 10, 11]. Also, this experiment is representative both in terms of 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. We simulate exactly the same network configuration across each of the simulators, measuring the overall computation time required, including the simulation setup time and the event processing overheads.

The throughput results are plotted both on log-log and on linear scales in Figure 1.3. ns2 is highly inefficient compared to SWANS, running two orders of magnitude slower. SWANS outperforms GloMoSim by a factor of 2. However, as expected, the simulation times in all three cases are still quadratic functions of the number of nodes. To address this, we designed a scalable, hierarchical binning algorithm (discussed next) to simulate the signal propagation. As seen in the plot, SWANS-hier scales linearly with the network size.

1.4 Hierarchical Binning

In addition to an efficient simulator design, it is also essential to model wireless signal propagation efficiently, since this computation is performed on every packet transmission. When a simulated radio transmits a signal, SWANS must simulate the reception of that signal at all the radios that could be affected, after considering fading, gain, and path loss. Some small subset of the radios in the coverage area will be within interference range, above some sensitivity threshold. An even smaller subset of those radios will be within reception range. The 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 reception range. This clearly does not scale as the number of radios increases. ns2 has recently been improved with a grid-based algorithm [18]. We have implemented both of these algorithms in SWANS. In addition, we have a new, more efficient and scalable algorithm that uses *hierarchical* binning. The spatial partitioning imposed by each of these data structures is depicted in Figure 1.4.

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

Hierarchical binning improves on the flat binning approach by dividing the coverage area recursively. Node bins are leaves of a balanced, spatial decomposition tree, which is of height equal to the number of divisions, or $h = log_4(\frac{coverage\ area}{bin\ size})$. The hierarchical binning structure is like a quad-tree, except that the division points are not the nodes themselves, but rather fixed coordinates. A similar idea is proposed in GLS, a distributed location service for ad hoc networks [14]. Note that the height of the tree changes only logarithmically with changes in the bin or coverage area sizes. Furthermore, since nodes move only a short distance between updates, the amortized height of the common parent of the two affected node bins is constant in expectation. This, of course, is under the assumption of a reasonable node mobility model that keeps the nodes uniformly distributed and also selects trajectories uniformly. Thus, the amortized node location update cost is constant, including the maintenance of the inner node

counts. When scanning for node neighbors, empty bins can be pruned as we descend. Thus, the set of receiving radios can be computed in time proportional to the number of receiving radios. Since, at a minimum, we 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 [7], even when assuming no cache misses. But, the memory overhead of the hierarchical binning scheme is minimal. Asymptotically, it amounts to $\lim_{n\to\infty} \sum_{i=1}^{log_4 n} \frac{n}{4^i} = \frac{n}{3}$, where n is the number of network nodes. The memory overhead for function caching is also $O(n)$, but with a much larger constant. Furthermore, the memory accesses for hierarchical binning are tree structured, exhibiting better locality.

1.5 Memory footprint

Memory is critical for simulation scalability. In the case of SWANS, memory is frequently the limiting resource. Thus, conserving memory allows for the simulation of larger network models. SWANS benefits greatly from the underlying Java garbage collector. 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:

- **shared state** – Memory consumption can often be dramatically reduced by sharing common state across entities. For example, simulated network packets are modeled in SWANS as a chain of objects that mimic the chain of packet headers added by successive layers of the simulated network stack. Moreover, since the packets are timeless, by design, a single broadcasted packet can be shared safely among all the receiving nodes. In fact, the very same data object sent by an application entity at the top of one network stack would be received by the application entity at a receiving node. In addition to the performance benefits of zero-copy semantics, as discussed previously, this sharing also saves the memory required for multiple packet copies on every transmission. Though this optimization may seem trivial, depending on their size, lifetime, and number, packets can occupy a considerable fraction of the simulation memory footprint. Similarly, if one employs the TCP component within a simulated node stack, then the very same object that is received at one node may be referenced within TCP retransmit buffer at the sending node, reducing the memory required to simulate even large transmission windows. Naturally, this type of memory protocol can also be implemented within the context of a non-garbage collected simulation environment. However, dynamically created objects, such as packets, can traverse many different control paths within the simulator and can have highly variable lifetimes. In SWANS, accounting for when to free unused packets is handled entirely by the garbage collector, which not only greatly simplifies the code, but also eliminates a common source of memory leaks that plague long simulation runs in other non-garbage collected simulators.
- **soft state** – Soft state, such as various caches within the simulator, may be used to improve simulation performance. But, these caches should be reclaimed when memory becomes scarce. An example of soft state within SWANS are routing tables computed from link state. The routing tables may be automatically collected to free up memory and later re-generated, as needed. A pleasing side-effect of this interaction with the garbage collector is that when memory becomes scarce, only the most useful and frequently used cached information will be retained. In the case of routing tables, the

cached information would be dropped altogether with the exception of the most active network nodes. As above, this type of memory management could also be implemented manually, though it likely would be too complex to be practical. A garbage collected environment simplifies the simulator code dramatically and increases its robustness.

To evaluate the JiST and SWANS memory requirements, we perform the same experiments as presented earlier, but measure memory consumption. The simulator memory footprint that we measure includes the base process memory, the memory overhead for simulation entities, and all the simulation data at the beginning of the simulation. Figure 1.5 plot shows the JiST micro-benchmark and SWANS macro-benchmark results. The plots show that the base memory footprint for each of the systems is less than 10 MB. Also, asymptotically, the memory consumed increases linearly with the number of entities, as expected. JiST performs well with respect to the memory overhead for simulation entities, since they are just small Java objects allocated on a common heap. It performs comparably to GloMoSim in this regard, which uses a technique called node aggregation specifically to reduce Parsec's memory consumption. A GloMoSim "entity" is also a small object containing an aggregation identifier and other variables similar to those found in SWANS entities. In contrast, each Parsec entity contains its own program counter and a relatively large logical process stack. In ns2, the benchmark program allocates the smallest split object possible, which duplicates simulation state in both the C and the Tcl memory spaces. JiST provides the same dynamic configuration capability using reflection, without requiring the memory overhead of a split object design.

Since JiST is more efficient than GloMoSim and ns2 by almost an order and two orders of magnitude, respectively, SWANS is able to simulate networks that are significantly larger. The memory overhead of hierarchical binning is shown to be asymptotically negligible. Also, 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 [22, 19]. Table 1.2 provides exact time and space requirements under each of the simulators for simulations of NDP across a range of network sizes.

Finally, we present SWANS with some very large networks. For these experiments, we ran the same simulations, but on dual-processor 2.2GHz Intel Xeon machines (though only one processor was used) with 2GB of RAM running Windows 2003. The results are plotted in Figure 1.6 on a log-log scale. We show SWANS both with the naïve propagation algorithm and with hierarchical binning. We observe linear behavior for the latter in all simulations up to networks of one million nodes. The 10^6 node simulation consumes just less than 1GB of memory on initial configuration, runs with an average footprint of 1.2GB (fluctuating due to delayed garbage collection), and completes within $5\frac{1}{2}$ hours. This size of network exceeds previous ns2 and GloMoSim capabilities by two orders of magnitude.

1.6 Embedding applications

SWANS has a unique and important advantage over existing network simulators. It can run regular, unmodified Java network applications over simulated networks, thus allowing for the inclusion of existing Java-based software, such as web servers, peer-to-peer applications, and application-level multicast protocols, within network simulations. These applications do not merely send packets to the simulator from other processes. They operate in simulation time,

embedded within the same SWANS process space, incurring no blocking, context switch, or memory copy, and, thereby, allowing far greater scalability.

This tight, efficient integration is achieved through a sequence of transparent program transformations, whose end result is to embed a process-oriented Java network application into the event-oriented SWANS simulation environment. The entire Java application is first wrapped within a special Java application entity harness. Since multiple instances of an application can be running within different simulated nodes, the harness entity serves as an anchor for the application context. Like the regular Java launcher, the harness entity invokes an application's `main` method to initiate it.

Before loading an application, SWANS inserts a custom rewriting phase into the simulation kernel class loader. Among other, more subtle modifications, this rewriting phase replaces all Java socket calls within the original application with invocations to corresponding functionality within simulated SWANS sockets. These SWANS sockets have identical semantics to their native counterparts in the standard Java library, but send packets through the simulated network. Most importantly, the input (receive) and output (send) methods are implemented using *blocking* JiST events and simulation time continuations, which explain next.

The semantics of a blocking events, as depicted in Figure 1.7, are a natural extension atop the existing non-blocking events. The kernel first saves the call-stack of the calling entity and attaches it to the outgoing event. When the call event is completed, the kernel notices that the event has caller information and the kernel 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 entity 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 in the interim.

To support these blocking semantics, JiST automatically modifies the necessary application code into a continuation-passing style. This allows the application to operate within the event-oriented simulation time environment. Our design allows blocking and non-blocking entity methods to co-exist, which means that event-oriented and process-oriented simulations can co-exist. Unfortunately, saving and restoring the Java call-stack for continuation is not a trivial task [23]. The fundamental difficulty arises from the fact that stack manipulations are not supported at either the language, the library, or the bytecode level. Our solution draws and improves on the ideas in the JavaGoX [25] and the PicoThreads [6] projects, which also save the Java stack for other reasons. Our design eliminates the use of exceptions to carry state information. This is considerably more efficient for our simulation needs, since Java exceptions are expensive. Our design also eliminates the need to modify method signatures. This fact is significant, since it allows our continuation capturing mechanism to function even across the standard Java libraries. In turn, this enables us to run standard, unmodified Java network applications within SWANS. A network socket read, for example, is rewritten into a blocking method invocation on a simulated socket, so that an application is "frozen" in simulation time until a network packet is delivered to the application by the simulated socket entity.

1.7 Conclusion

In summary, SWANS is a componentized wireless network simulator built according to the virtual machine-based simulator design [5]. It significantly out-performs ns2 and GloMoSim both in time and space. We have shown results with networks that, at the same level of detail, are more than an order of magnitude larger than what is possible with these existing tools. SWANS can also, unlike any existing network simulator, efficiently run existing, Java-based network applications over simulated networks. In general, SWANS inherits the advantages of JiST and the Java platform.

The SWANS simulator is freely available [4], and it could be extended in a number of interesting directions:

- *Parallel, distributed, and speculative execution* - The current implementations of JiST and SWANS have focused exclusively on sequential performance. The system, however, was explicitly designed and implemented with more sophisticated execution strategies in mind. It is possible to extend the simulation kernel to allow multiple processing threads to operate concurrently on the simulation state in order to leverage the full processing power of commodity multi-processor machines. For distributed simulation, JiST entity separators can readily be extended to support a single system abstraction by transparently tracking entity locations as they are dynamically migrated across a cluster of machines to balance computational and network load. The JiST kernel can then be extended to support conservatively synchronized, distributed, cooperative operation with peer JiST kernels, which would increase the available simulation memory and allow larger network models to be processed. Synchronization protocols need not remain conservative. Since the JiST design can already transparently support both checkpointing and rollback of entities, and since the cost of synchronization is critical to the performance of a distributed simulator, it is worthwhile to investigate various speculative execution strategies as well. Each of these three extensions to the simulation kernel – parallel, distributed, and speculative execution – are already supported within the current JiST semantics and can, therefore, be implemented transparently with respect to existing SWANS components. Such extensions to the simulation kernel pose a rich space of design and research problems.
- *Declarative simulation specifications* - SWANS is a componentized simulator and, therefore, tends naturally to be configured as a graph of inter-connected entities that can model some network with given topology and node configurations. Since these component graphs often have repetitive and highly compressible structure, it would be beneficial to construct them using short declarative specifications, rather than the current approach, driven by imperative scripts. In general, it would be interesting to develop high-level, possibly domain-specific, yet expressive, simulation configuration languages.
- *Simulation debuggers and interactive simulators* - A significant advantage of leveraging the Java platform 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 simulation kernel data structures, resulting in functionality that is unparalleled in any existing simulation environment. Since Java is a reflective language, SWANS simulations may be paused, modified in-flight, and then resumed. The appropriate tools to perform such inspection effectively (i.e., a graphical, editable view of the network and the state of its nodes, for example), would facilitate

interactive simulation and present interesting research opportunities. For example, one could use the debugger to control the distributed simulation kernel and its global virtual time 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, we hope that the scalability, performance, and flexibility of SWANS, combined with the popularity of the Java language, will facilitate its broader adoption within the network simulation community.

References

[1] Bowen Alpern, C. Richard Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark F. Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Object-Oriented Programming Systems, Languages and Applications*, pages 314–324, November 1999.

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

[3] Rajive L. Bagrodia, Richard Meyer, Mineo Takai, Yuan Chen, Xiang Zeng, Jay Martin, and Ha Yoon Song. Parsec: A parallel simulation environment for complex systems. *IEEE Computer*, 31(10):77–85, October 1998.

[4] Rimon Barr and Zygmunt J. Haas. JiST/SWANS website, 2004. `http://jist.ece.cornell.edu/`.

[5] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. JiST: Embedding simulation time into a virtual machine. In *Proceedings of EuroSim 2004*, September 2004.

[6] Andrew Begel, Josh MacDonald, and Michael Shilman. PicoThreads: Lightweight threads in Java. Technical report, UC Berkeley, 2000.

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

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

[9] Richard M. Fujimoto. Parallel and distributed simulation. In *Winter Simulation Conference*, pages 118–125, December 1995.

[10] Zygmunt Haas and Marc Pearlman. Providing ad hoc connectivity with reconfigurable wireless networks. In Charles Perkins, editor, *Ad hoc Networks*. Addison Wesley Longman, 2000.

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

[12] Owen Kelly, Jie Lai, Narayan B. Mandayam, Andrew T. Ogielski, Jignesh Panchal, and Roy D. Yates. Scalable parallel simulations of wireless networks with WiPPET. *Mobile Networks and Applications*, 5(3):199–208, 2000.

[13] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *European Conference on Object-Oriented Programming*, 1241:220–242, 1997.

[14] Jinyang Li, John Jannotti, Douglas S. J. De Couto, David R. Karger, and Robert Morris. A scalable location service for geographic ad hoc routing. In *ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM)*, pages 120–130, 2000.

[15] Jason Liu, L. Felipe Perrone, David M. Nicol, Michael Liljenstam, Chip Elliott, and

David Pearson. Simulation modeling of large-scale ad-hoc sensor networks. In *Simulation Interoperability Workshop*, 2001.

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

[17] Jayadev Misra. Distributed discrete event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[18] Valeri Naoumov and Thomas Gross. Simulation of large ad hoc networks. In *ACM MSWiM*, pages 50–57, 2003.

[19] David M. Nicol. Comparison of network simulators revisited, May 2002.

[20] David M. Nicol and Richard M. Fujimoto. Parallel simulation today. *Annals of Operations Research*, pages 249–285, December 1994.

[21] George Riley and Mostafa Ammar. Simulating large networks: How big is big enough? In *Conference on Grand Challenges for Modeling and Sim.*, January 2002.

[22] George Riley, Richard M. Fujimoto, and Mostafa A. Ammar. A generic framework for parallelization of network simulations. In *Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication*, March 1999.

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

[24] Prince Samar, Marc Pearlman, and Zygmunt Haas. Hybrid routing: The pursuit of an adaptable and scalable routing framework for ad hoc networks. In *Handbook of Ad Hoc Wireless Networks*. CRC Press, 2003.

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

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

[27] Xiang Zeng, Rajive L. Bagrodia, and Mario Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, May 1998.

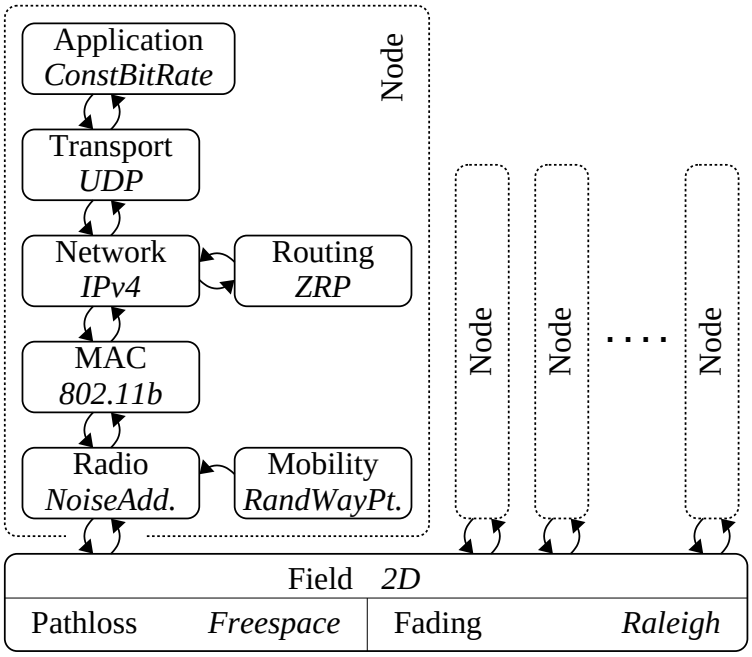

Figure 1.1: The SWANS simulator consists of event-driven components that can be configured and composed to form wireless network simulations.

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 1.1: Time elapsed to perform 5 million events, normalized both against the reference and JiST measurements.

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*

Table 1.2: SWANS outperforms ns2 and GloMoSim in both time and space.

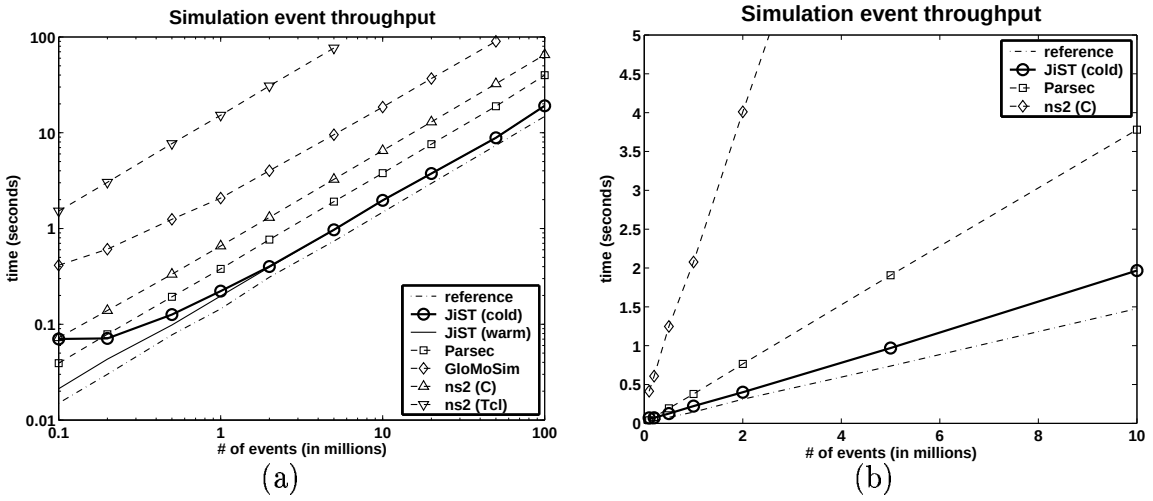

Figure 1.2: JiST has higher event throughput and comes within 30% of the reference lower bound program. The kink in the JiST curve in the first fraction of a second of simulation is evidence of JIT dynamic compilation and optimization at work.

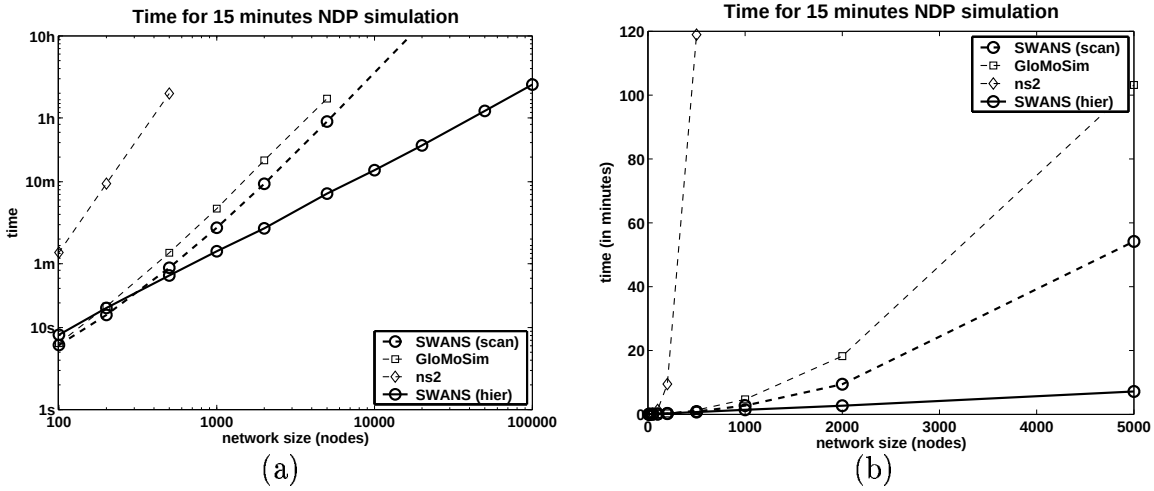

Figure 1.3: SWANS outperforms both ns2 and GloMoSim in simulations of NDP.

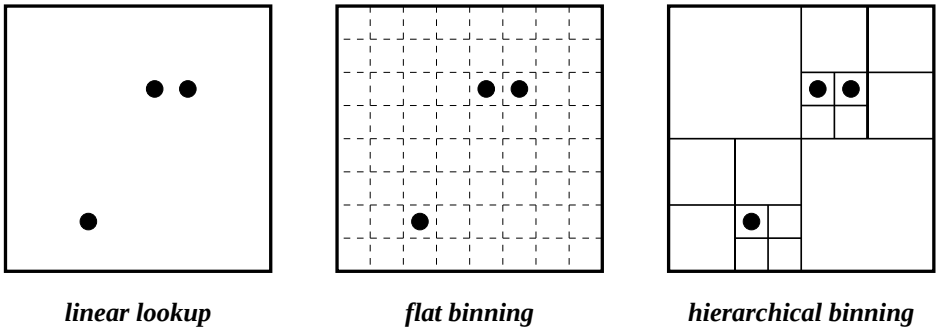

Figure 1.4: Hierarchical binning of radios in the coverage area allows location updates to be performed in expected amortized constant time and the set of receiving radios to be computed in time proportional to its size.

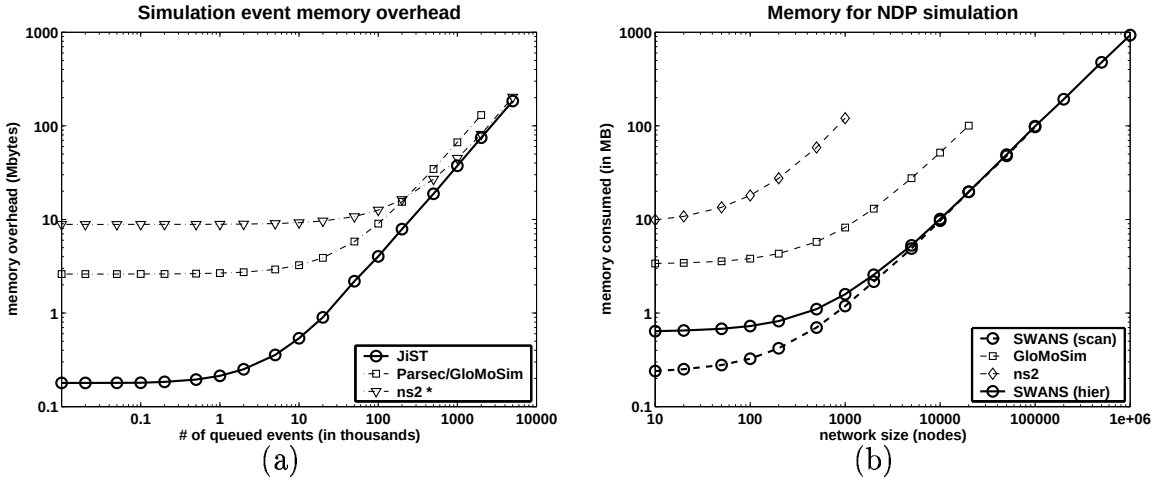

Figure 1.5: JiST allocates entities efficiently: comparable to GloMoSim at 36 bytes per entity and over an order of magnitude less than Parsec or ns2. SWANS can simulate correspondingly larger network models due to this more efficient use of memory and more sophisticated memory management protocols enabled by the garbage collector.

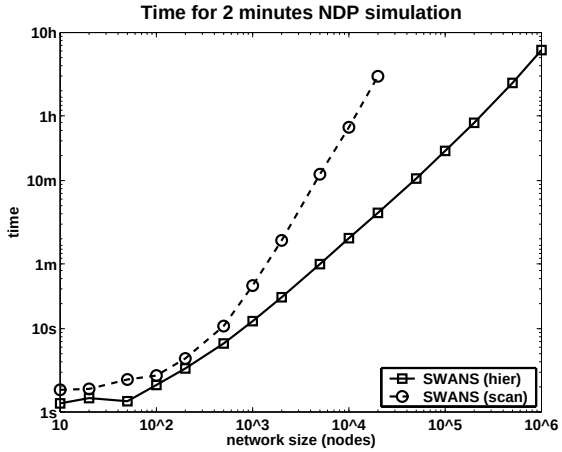

Figure 1.6: SWANS scales to networks of 10^6 wireless nodes. The figure shows the time for a sequential simulation of a heartbeat NDP protocol.

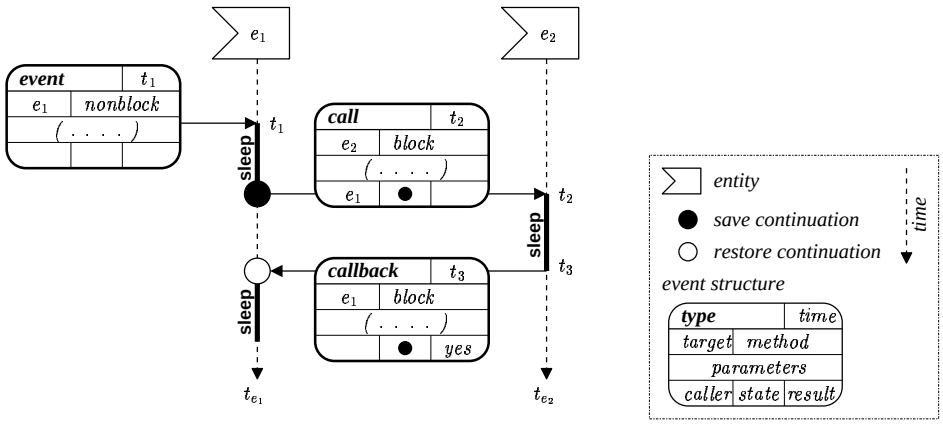

Figure 1.7: The addition of blocking methods allows simulation developers to regain the simplicity of process-oriented development. When a blocking entity method is invoked, the continuation state of the current event is saved and attached to a call event. When this call event is complete, the kernel 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.