SWANS– Scalable Wireless Ad hoc Network Simulator
User Guide

Rimon Barr
barr@cs.cornell.edu

1 Introduction

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.

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 simulators remain the norm [8]. 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.

SWANS is a **S**calable **W**ireless **A**d hoc **N**etwork **S**imulator built atop the JiST platform, a general-purpose discrete event simulation engine. SWANS was created primarily because existing wireless network simulation tools are not sufficient for current research needs. SWANS also serves as a validation of the virtual machine-based approach to simulator construction.

2 Alternatives

The two most popular simulators in the wireless networking space are ns2 and GloMoSim. The ns2 network simulator [5] 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 [9]. However, this technique has proved primarily beneficial for distributing ns2’s considerable memory requirements. Based on numerous published results, it is not easy to scale ns2 beyond a few hundred simulated nodes.

Simulation researchers have shown ns2 to scale, with difficulty and substantial hardware resources, to simulations of a few thousand nodes [8].

GloMoSim [10] is a newer simulator written in Parsec [1], a highly-optimized C-like simulation language. GloMoSim has 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. The aggregation of state also renders speculative execution techniques impractical. GloMoSim has been shown to scale to 10,000 nodes on large, specialized multi-processor machines.

3 Design highlights

The SWANS software is organized as independent software components that can be composed to form complete wireless network or sensor network simulations, as shown in Figure 1. Its capabilities are similar to ns2 [5] and GloMoSim [10], 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.

Notably, the development of SWANS has been relatively painless. Since JiST inter-entity message creation and delivery is implicit, as well as message garbage collection and typing, the code is compact and intuitive. Components in JiST consume less than half of the code (in uncommented line counts) of comparable components in GloMoSim, which are already smaller than their 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 exposed event-based 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 creating relatively small, event-driven components. It also explicitly partitions the simulation state and the degree of inter-dependence between components, unlike the design of ns2 and GloMoSim. It also 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

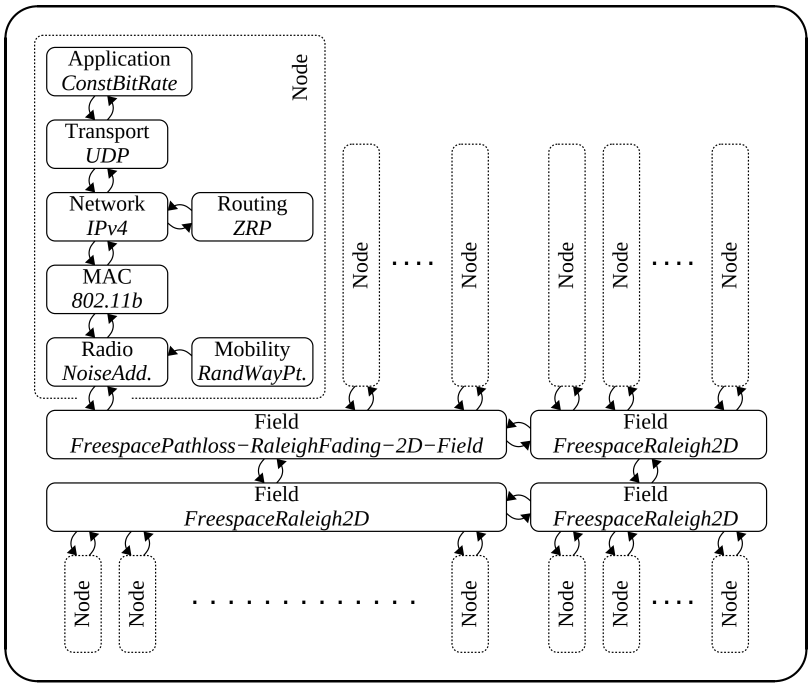


Figure 1: The SWANS simulator consists of event-driven components that can be configured and composed to form the desired wireless network simulation. Different classes of components are shown in a typical arrangement together with specific instances of component implementations in italics.

events are passed along by reference via the simulation time kernel. Simulated network packets are actually 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 we use TCP in our node stack, then the same object will 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 simplifies the memory management protocol, 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 Embedding Java-based network applications

SWANS has a unique and important advantage over existing network simulators. It can run regular, unmodified Java network applications over the simulated network, thus allowing for the inclusion of existing Java-based software, such as web servers, peer-to-peer applications and application-level multicast protocols. 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 disk accesses.

We achieve this integration via a special `AppJava` application entity designed to be a harness for Java applications. This harness inserts an additional rewriting phase into the JiST kernel, which substitutes SWANS socket implementations for any Java counterparts that occur 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 (built using JiST continuations). To support these blocking semantics, JiST automatically modifies the necessary application code into continuation-passing style, which allows the application to operate within the event-oriented simulation time environment.

5 Efficient signal propagation

Modeling signal propagation 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 pathloss. 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 [6]. We have implemented both of these in SWANS. In addition, we have a new, more efficient algorithm that uses *hierarchical* binning. The spatial partitioning imposed by each of these data structures is depicted in Figure 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 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, for example, as in the case of power-controlled transmissions.

We improve 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

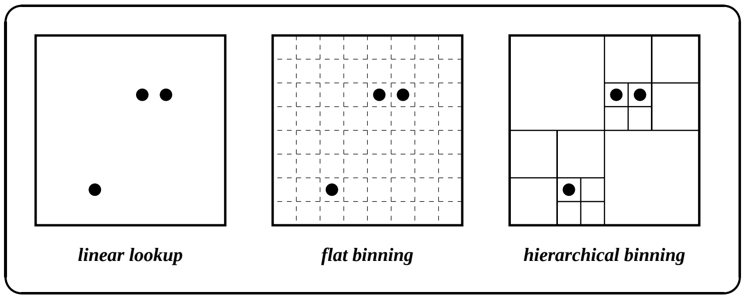


Figure 2: Alternative spatial data structures for radio signal propagation: Efficient signal propagation is critical for wireless network simulation performance. Hierarchical binning of radios on the field allows location updates to be performed in expected amortized constant time and the set of receiving radios to be computed in time proportional to its size.

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 as we descend spatially. 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 [2], 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.

6 Components

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

6.1 Physical

The SWANS physical layer components are responsible for modeling signal propagation among radios as well as the mobility of nodes. Radios make transmission downcalls to the simulation “field” and other radios on the “field” receive reception upcalls from it, if they are within range of the signal. Both the pathloss and fading models are functions that depend on the source and destination radio locations. Pathloss models include free-space, two-ray and table-driven pathloss. Fading models include 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 pathloss
Fading	Fading.None	zero fading model
	Fading.Raleigh	Raleigh fading model
	Fading.Rician	Rician fading model
Pathloss	Pathloss.FreeSpace	pathloss model based purely on distance
	Pathloss.TwoRay	pathloss model that incorporates ground reflection
Spatial	Spatial.Linear	signal propagation and location update performed via linked list of radios
	Spatial.Grid	as above, but performed using a more efficient flat grid structure of small "Linear" bins
	Spatial.HierGrid	as above, but performed using a more consistently efficient hierarchical grid structure
	...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.
	...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 upcalls from the field entity and passes successfully received packets on to the link layer. It also receives downcalls from the link layer entity and passes them on to the field for propagation. We have implemented an independent interference radio, as in ns2, as well as an additive interference radio, as in GloMoSim. The independent interference model considers only signals destined for the target radio as interference. The additive model correctly considers all signals as contributing to the interference. Both radios are half-duplex, 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
	RadioNoiseAdditive	interference consists of all signals above a threshold

jist.swans.radio.*

interface	**class**	**description**
none	RadioInfo	unique and shared radio parameters

6.2 Link

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

6.3 Network

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

jist.swans.net.*

interface	**class**	**description**
NetInterface	NetIp	IPv4 implementation
PacketLoss	Zero	zero network layer packet loss
	Uniform	independent, random drop with fixed probability

jist.swans.net.*

interface	**class**	**description**
none	NetAddress	network address
	MessageQueue	outgoing message queues

6.4 Routing

The routing entity recieves upcalls from the network entity with packets that require next-hop information. It also receives upcalls that allow it to peek at all packets that arrive at a node. It sends downcalls to the network entity with next-hop information when it becomes available. SWANS implements the Zone Routing Protocol (ZRP) [3], Dynamic Source Routing (DSR) [4] and ad hoc On-demand Distance Vector Routing (AODV) [7].

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

6.5 Transport

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

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

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

jist.swans.trans.*

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

6.6 Application

At the top of our network stack, we have the application entities, which make downcalls to the transport layer and receive upcalls from it, usually via SWANS sockets or streams that mimic their Java equivalents. The most generic and useful kind of application entity is a harness for regular Java applications. 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

6.7 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.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.

7 Performance

In this section, we show that SWANS perform surprisingly well: we compare SWANS with the two most popular ad hoc network simulators: ns2 and GloMoSim. We selected these because they are widely used, freely available sequential network simulators designed in the systems-based and language-based approaches, respectively.

We present macro-benchmark results running full SWANS simulations. 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. 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. 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).

7.1 Beaconing

In the following experiment, we benchmarked JiST running a full SWANS ad hoc wireless network simulation. We measured the performance of simulating 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 [3, 4]. 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 that we have 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, we were also able to validate the simulation results against analytical estimates.

We constructed the following identical scenario 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 [4] 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.

We 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, we constructed a stack of 802.11b MAC, IPv4 network, UDP transport, and NDP application entities.

We ran this network model for 15 simulated minutes and measured overall memory and time required for the simulation. For memory, we included the base process memory, the memory overhead for simulation entities, and all the simulation data at the beginning of the simulation. For time, we included the simulation setup time, the event processing overheads, and the application processing time.

The throughput results are plotted both on log-log and linear scales in Figure 3. 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.

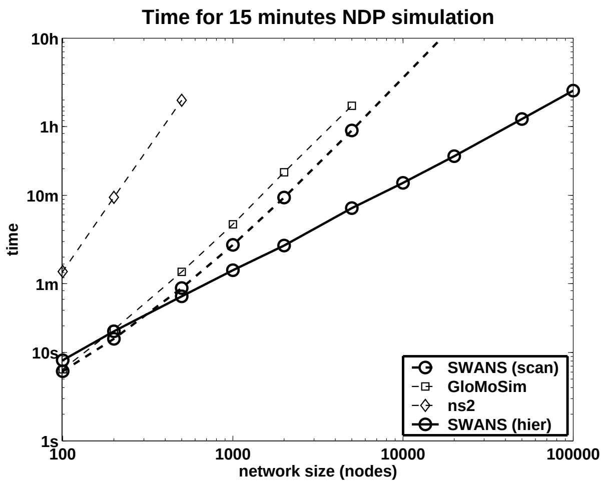
The memory footprint results are plotted in Figure 4 on log-log scale. JiST is more efficient than GloMoSim and ns2 by almost an order and two orders of magnitude, respectively. This allows SWANS to simulate much larger networks. The memory overhead of hierarchical binning is asymptotically negligible.

Finally, we tested SWANS with some very large networks. We ran the same simulations 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 5 on a log-log scale. We show SWANS both with the naïve propagation algorithm and with hierarchical binning, and we 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 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 only commodity hardware.

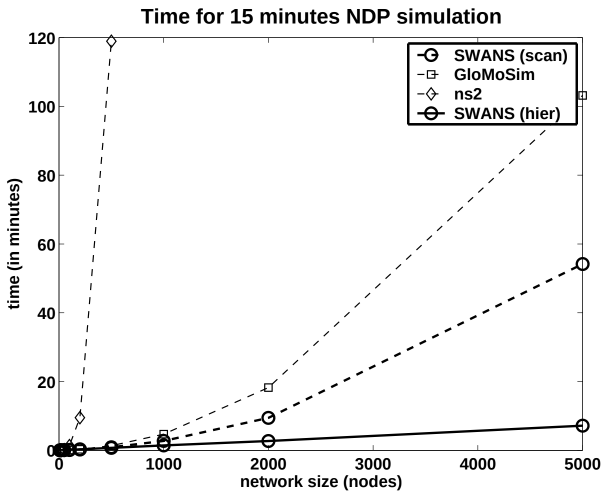
7.2 Zone routing protocol

Next, we ran some large experiments with an actual ad hoc wireless network routing protocol, ZRP (Zone Routing Protocol). Figure 6 shows the time and memory required to simulate different size networks at a fixed density of 10 neighbors per node. The experiments were run on a dual-processor 2.8 GHz Intel Xeon machine with 2GB of RAM and a similar software configuration to the previous Linux machine. The memory requirements grow linearly with the size of the network. The time required grows slightly faster than linear due to garbage collection overhead (using default GC parameters).

It is not clear that simulating such a large flat ad hoc network is meaningful, except to exhibit SWANS scalability and performance. However, one could certainly simulate many smaller, connected flat ad hoc networks with comparable workload. Regardless, these results far exceed GloMoSim or ns2 capabilities, by approximately an order and two orders of magnitude, respectively. Smaller networks run proportionally faster and in proportionally less memory in JiST than in either GloMoSim or ns2.

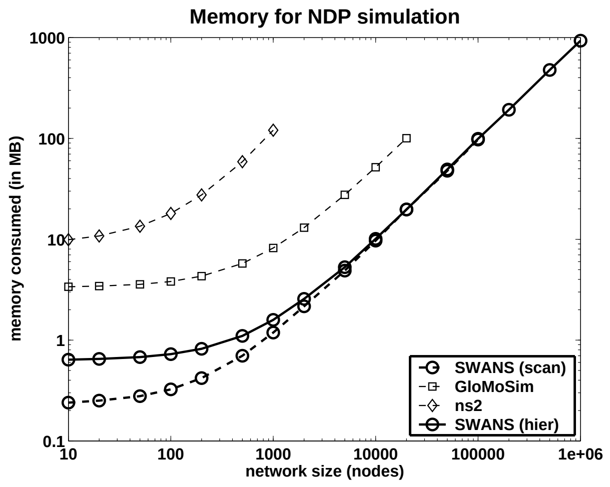


(a) log-log scale



(b) linear scale

Figure 3: SWANS significantly outperforms both ns2 and GloMoSim in simulations of the node discovery protocol.



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*

Figure 4: SWANS can simulate larger network models due to its more efficient use of memory.

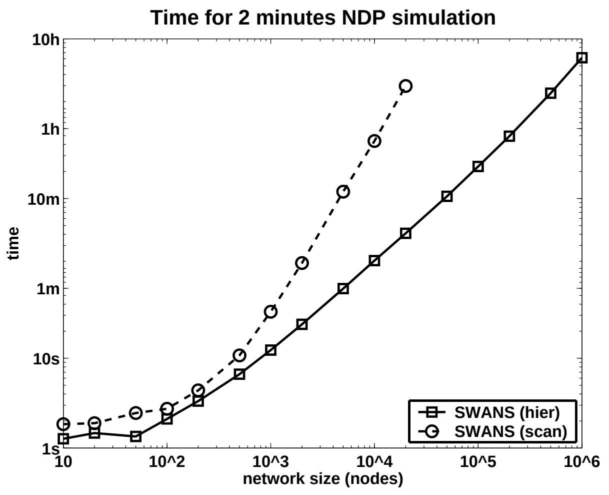
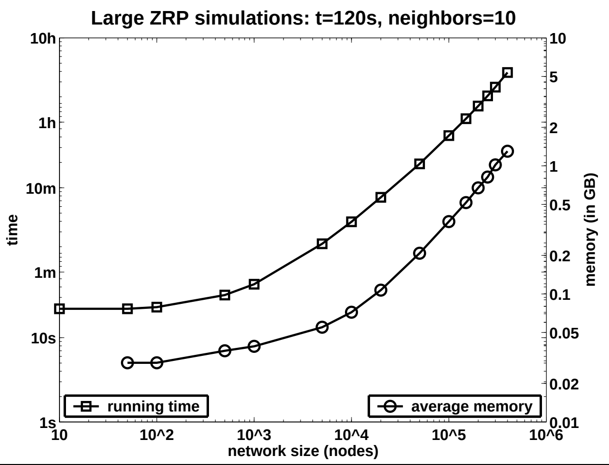


Figure 5: SWANS scales to networks of 10^6 wireless nodes. The figure shows the time for a sequential simulation of a node discovery protocol in a wireless ad hoc network running on a commodity machine.



nodes	time	avg.memory	max.memory
10,000	3m57s	72MB	94MB
100,000	41m36s	367MB	476MB
400,000	3h52m	1.30GB	1.57GB

Figure 6: SWANS simulates 400,000 nodes at a density of 10 neighbors per node, each running ZRP, on a 2.8GHz machine with 2GB of memory.

8 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. We leverage the JiST design 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 $10\times$ 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, such as web servers and peer-to-peer applications, over the simulated network without modification. The application is automatically transformed to use simulated sockets and into a continuation-passing style. The original network applications are run within the same process as SWANS, 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, since this computation is performed on every packet transmission. The hierarchical binning data structure is allows node location updates in expected amortized constant time and receiver node set computations in time proportional to the number of receivers. The combination of these attributes leads to a flexible and efficient simulator. We hope that the performance of SWANS will facilitate further research into scalable ad hoc network protocols.

References

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

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

[3] Z. J. Haas. A new routing protocol for the reconfi gurable wireless networks. In *IEEE Conference on Universal Personal Comm.*, Oct. 1997.

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

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

[6] V. Naoumov and T. Gross. Simulation of large ad hoc networks. In *ACM MSWiM*, pages 50–57, 2003.

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

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

[9] G. Riley, R. M. Fujimoto, and M. A. Ammar. A generic framework for parallelization of network simulations. In *MASCOTS*, Mar. 1999.

[10] X. Zeng, R. L. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *PADS*, May 1998.