Simulation of Ad Hoc Networks: ns-2 compared to JiST/SWANS

Elmar Schoch, Michael Feiri, Frank Kargl, Michael Weber
Ulm University, Institute of Media Informatics
89069 Ulm, Germany
elmar.schoch | michael.feiri | frank.kargl | michael.weber@uni-ulm.de

ABSTRACT

For the evaluation of ad hoc network protocols, researchers traditionally use simulations because they easily allow for a large number of nodes and reproducible environment conditions. But still, simulations are costly in terms of required processing time and memory. New approaches like the Java-based simulator JiST/SWANS promise to provide significant performance advantages compared to existing, well-known network simulators like ns-2. Though its creators have done comparison studies regarding processing time and memory requirements of JiST/SWANS, they did not test the validity of the results produced by the SWANS stack as well as its performance in detail.

In this work, we focus on the comparability of implemented protocols and models in SWANS. Using the corresponding counterparts both in SWANS and ns-2, with an identical set of parameters, we are able to produce results and compare them afterward. In addition, we also provide a performance analysis using these simulations. By showing that JiST/SWANS produces equivalent simulation results as ns-2 in less time and with less memory, we support JiST/SWANS as a potential alternative.

Categories and Subject Descriptors

I.6 [**Simulation and Modeling**]: General

General Terms

Performance, Measurement

Keywords

Simulation, Comparison, ns-2, JiST/SWANS

1. INTRODUCTION

Simulations play an important role in the development and evaluation of future systems. For ad hoc networks, simulations are traditionally used for testing since they allow reproducible environment conditions and a large number of nodes. In real world tests, wireless transmission conditions can vary from experiment to experiment, which makes it impossible to compare the performance of different protocols under identical circumstances. In simulations, a scenario can be repeated exactly as often as required. Moreover, setting up test equipment is usually also a time-consuming task. Another problem arises when researches want to study protocol scalability with a large number of nodes, which is typical for ad hoc network protocol evaluation. When scenarios consist of hundreds of nodes, cost and time constraints usually do not allow to set up real world tests.

For simulation of ad hoc networks, a number of required elements can be identified.

- **Simulation engine**
  As a core component, the simulation engine orchestrates the execution of code parts representing simulated logic. Most of the existing simulators for ad hoc networks like ns-2, OPNET Modeler, Omnet++ as well as JiST/SWANS are based on discrete events in simulation time. With simulation time, the simulation engine introduces a notion of time that is decoupled from the real execution time of the simulation.
- **Physical environment model**
  For simulating ad hoc networks, the physical model primarily has to reproduce radio propagation. In addition, a more detailed model of the world in which the nodes reside is often required, e.g. by including rooms, buildings or streets.
- **Node model**
  As the central elements, the simulator must specify how nodes interact with their environment and with other nodes. The interaction with the environment typically includes sending out messages and moving around. For the interaction with other nodes, the same components need to be implemented that would also be available in real devices, e.g. medium access according to 802.11, a routing protocol like AODV, a transport mechanism like TCP or UDP and applications that mainly generate data traffic.
- **Execution and evaluation support**
  Because simulations consist of a lot of components, which all need to be configured, keeping track of all parameters of a simulation is not easy. The same holds for traces and aggregated results of a simulation. A

SIMUTOOLS 2008, March 03-07, Marseille, France
Copyright © 2008 ICST 978-963-9799-20-2
DOI 10.4108/ICST.SIMUTOOLS2008.3021

solution to handle input and output as well as to efficiently execute multiple simulations with varying parameters is valuable.

As mentioned before, simulations are generally restricted by two main constraints: **accuracy** and **runtime requirements**. On the one hand, studies need to model the simulated network as close to reality as possible. On the other hand, processing time and memory requirements need to be kept on a reasonable level. This trade-off between accuracy and runtime requirements is a typical problem to be balanced.

A number of simulation tools including suitable libraries is available for MANET simulations. One of the most prominent is ns-2 [1], which was originally designed for simulation of wired networks. However, since CMU published their "wireless extensions" [6], it is also widely used for MANETs. The simulator relies on a split-object model, where parts of simulation code can be written in OTcl, whereas most logic is implemented in C/C++. Due to its long tradition, a large amount of extensions is available for ns-2. However, ns-2 simulations tend to require very much processing time and memory, when simulations exceed several hundreds of nodes.

To thwart that problem, a number of improvements have been developed, for instance the parallelized version called PDNS [9]. With PDNS, ns-2 simulations can be distributed to a compute cluster, and [8] shows that PDNS was able to simulate a network with 600,000 nodes on a cluster with 136 processors, high-end hardware and a very fast interconnection network.

Compared to standard ns-2, GloMoSim [13] is slightly more performant on standard hardware and allows for simulations of networks consisting of roughly 10,000 nodes. GloMoSim is based on Parsec, a C-like programming language for discrete-event simulation developed at UCLA. It includes a library for MANET simulations which already provides a number of protocol implementations, but by far not that many that ns-2 delivers. A parallel version of GloMoSim called QualNet [10] is available as a commercial product, which the manufacturer claims to be able to scale up to "10s of thousands of nodes".

Another commercially available simulator is OPNET Modeler [7]. It allows for creating and configuring simulations using a mixture of graphical editors, state-diagrams and C++. In contrast to ns-2 and GloMoSim, simulations are compiled into a dedicated executable before running. Both in terms of protocol support and performance, OPNET is comparable to ns-2. For instance, in [12], the authors show that OPNET Modeler also has a relatively poor scalability.

Hence, if researchers want to conduct simulations with high accuracy and large number of nodes in reasonable time, only few alternatives remain. Recently, the new Java-based simulator JiST/SWANS developed by Rimon Barr e.a. at Cornell University has attracted more and more attention in the MANET community. Though the simulator is still in an early stage and available components are poor so far, the interest is continuously growing due to its performance benefits and several additional advantages.

The authors of JiST/SWANS already provided an in-depth performance comparison between JiST and other simulators. However, there is currently no work that particularly focuses on the SWANS part, i.e. the question is not answered yet, whether JiST/SWANS simulations lead to results comparable to other simulators, which is important for the acceptance of the simulator. In this paper, it is our goal to provide answers to that question by investigating various elements of SWANS in comparison to their ns-2 counterparts.

The following section gives an introduction on ideas and concepts of JiST/SWANS. We are going to review performance measurements and give a state-the-art on ongoing JiST/SWANS activities. In Section 3, we look at basic architectural differences between JiST/SWANS and ns-2 which explain performance gains of JiST/SWANS. After that, Section 4 provides a comparison of ad-hoc network simulations conducted with both simulators.

2. JIST/SWANS

The JiST [3] abbreviation stands for "Java in Simulation Time", which already denotes the basic idea, namely to transform the Java virtual machine into a scheduler for events. Basically, this is done by modifying the way how methods are called between simulation entities. On top of this discrete event simulation engine, the authors have developed the "Scalable Wireless Ad Hoc Network Simulator" SWANS [2], which provides environment models and communication protocols for simulating MANETs.

2.1 JiST

JiST simulation entities are implemented and compiled like regular Java classes. Therefore, all advantages of Java like platform independence, large library support or widespread familiarity with language, apply as well for JiST. To introduce simulation time semantics, JiST partitions simulations into "Entities", which are actually Java objects as well, but whose public methods are always called via the scheduler. Because this is not standard Java behavior, these method calls are redirected to the simulation engine upon invocation. The redirection step is introduced by applying byte-code rewriting before the start of a simulation. During the simulation, when a method of an entity is called, the call returns immediately, a new entry is inserted in the scheduling queue and executed later.

To implement this behavior, every such entity class is marked either with the interface `JistAPI.Entity` or the interface `JistAPI.Proxiable`. Before execution of a simulation, calls to methods of classes marked with such an interface are subject to be rewritten. Whenever code within an entity is executed, it runs at a certain point in the global simulation time t_s. Within the entity, local entity time t_e can be advanced by calling the `JistAPI.sleep()` function. Thus, if an entity starts at $t_s = 1$ and then calls `JistAPI.sleep(1)`, t_e will be set to 2. Any following method call of another entity will then be scheduled to be executed at $t_s = 2$. From the Java perspective, however, such a call to another entity's method returns immediately, whereas regular Java would branch into it when executed without JiST.

The example in Listing 1 depicts a basic JiST entity. In the main method, an object of the class `Hello` is created, which in fact is an entity, as the class is marked with `JistAPI.Entity`. Thus, the following call of `myEvent()` is scheduled to be run at the current time, which is $t_s = 0$ at the beginning. After that, `main()` continues and terminates. Then, the scheduler calls the queued event, i.e. the `myEvent()` entity method. Upon the call, the local entity

```
import jist.runtime.JistAPI;
class Hello implements JistAPI.Entity {
  public static void main(String[] args) {
    System.out.println("Simulation start");
    Hello h = new Hello();
    h.myEvent();
  }
  public void myEvent() {
    JistAPI.sleep(1);
    myEvent();
    System.out.println("hello world,
        t="+JistAPI.getTime());
  }
}
```

Listing 1: A simple JiST Entity

time t_e is set to the scheduled time 0. As first action, the method increases t_e by one time unit, which means that now $t_e = 1$. Then, the method `myEvent` calls itself again. Note that this call is not recursive as it would be in regular Java. As it is a call to an entity method, it simply results in a new scheduler queue entry at $t_s = 1$. Finally, `myEvent` returns after the `println()` command. Now the scheduler will execute the next pending event, which is the one at $t_s = 1$. Again, the same procedure is repeated, and this loop continues infinitely.

In contrast to public methods, non-public methods of entity objects are called directly. Additionally, all public methods of an entity are not allowed to return any values as they are called by the scheduler. This is also one reason, why the interface `JistAPI.Proxiable` was introduced, which allows for facade interfaces that contain the public methods to rewrite on invocation.

The described JiST approach has a number of significant benefits:

- **Type-safety**: As the "delivery of events" is simply a method call, the virtual machine always checks the compatibility of source and destination. When receiving events, no additional type-checking needs to be done by the receiving entity.

- **No marshaling/demarshaling**: As there are no explicit event data structures, there is no overhead for marshaling/demarshaling data. Instead, the entities just pass references within the VM. Using RMI, references can even be passed beyond VM boundaries.

- **Java**: Using Java provides a large amount of benefits: robustness due to type safety, large library, ease of use, well-known to many developers, platform independence, garbage collection, good IDE support, etc.

- **Byte-code rewriting**: Because class files are directly transformed, no source access is necessary. Existing libraries and protocol implementations can directly be used for simulations.

- **Parallel distribution**: The JiST event scheduling can support parallel and optimistic execution, which crosses VM boundaries and is transparent for the application. However this is not implemented in the current version.

2.2 SWANS

SWANS is a complete library for simulation of MANETs running on the JiST engine. As already described in the introduction, MANET simulations need a model for the environment and for the nodes. In SWANS, the `Field` entity provides node mobility and radio propagation. Nodes consist of a number of entities implementing various protocol layers, where the radio entity is connected to the global field entity. Packets traverse the protocol stack entities usually as simple references, at virtually no cost. Duplication is only done where necessary, e.g. if a packet is broadcasted and needs to be changed by the forwarders.

Regarding radio propagation, one can choose a free space or a two-ray-ground pathloss model, together with Rayleigh fading, Rician fading or without fading. Moreover, a statistic packet dropping can be applied. As node mobility, the standard distribution supplies teleporting, random walk and random waypoint models. For the composition of nodes, SWANS brings basic radio noise models, an implementation of 802.11b MAC, IPv4, AODV, DSR and ZRP MANET routing, as well as TCP and UDP transport and several applications. As a special feature, SWANS also allows to run legacy Java network applications as part of simulations.

2.3 Performance

When designing a new simulation tool, evaluation of performance is a primary concern. Performance evaluations have been conducted by the authors of JiST/SWANS in [3], specifically regarding the performance of the JiST engine, but also including SWANS simulations.

For the JiST benchmark, a small simulation similar to the above example in Listing 1 was carried out. The program schedules an identical event step by step for 5 million times, so that only the raw event throughput is relevant. Then, the authors compare the runtime behavior with the same small program implemented in ns-2 and GloMoSim. As result of this competition, JiST performed this task 1.97 times faster than Parsec, 3.36 times faster than the ns-2 program (when implemented completely in C-code), 9.84 times faster than GloMoSim, and even 78.97 times faster than an ns-2 implementation in Tcl. As a performance baseline, the authors also implemented a plain C program that imitates the scheduler behavior by inserting and removing elements from an efficient array-based priority queue. Compared to that, JiST performs 31% slower. Regarding memory consumption, JiST performs similarly well.

Considering the efficiency of SWANS, the authors have implemented a simple neighbor discovery protocol in SWANS, ns-2 and GloMoSim. When using an optimized radio binning model, JiST/SWANS is able to outperform the other two simulators. For example, with 500 nodes on the field, SWANS runs 43 seconds and consumes $1,101$ kB of memory, whereas GloMoSim takes 82 seconds and $5,759$ kB and ns-2 needs $7,136$ seconds and $58,761$ kB memory. SWANS even can compute simulations with 50,000 nodes rather efficiently in $4,377$ seconds using $49,262$ kB of memory.

2.4 Ongoing efforts

Since the publication of JiST/SWANS, more and more researchers have started to use, to improve and to extend it. One of the earlier available extensions was STRAW [5], the "Street Random Waypoint" mobility model for simulation of vehicles driving on roads. Currently, this activity has been extended to an open source project called SWANS++. In addition, also numerous researchers have used JiST/SWANS in their work.

In our own work on JiST/SWANS, we have developed extensions of SWANS as well as a framework for automated execution of simulations. As mentioned in the introduction, simulations usually require to control input parameters and output data. Thus, a more sophisticated solution is desirable to handle these data. These tasks can be facilitated by our framework called DUCKS. DUCKS first of all implements a generic "driver" for SWANS simulations, which takes all necessary information from a configuration file and generates a set of corresponding simulations. For the execution, a central component is available which can distribute simulations to an arbitrary number of simulation servers. Finally, results from the simulation servers are collected and stored in a database, from where they can easily be queried.

Emerging from our research in the field of vehicular ad hoc networks, we also simplemented several extensions of SWANS, particularly geographic forwarding. The same algorithms and protocols have been implemented in ns-2 before, and therefore this will also be used in later sections. The logical equivalence of this code particularly allows us a meaningful comparison of the two simulators. Moreover, other extensions from the field of VANETs include a raytracing-based radio propagation model for urban environments and more are under development [1].

In summary, the popularity of JiST/SWANS is continuously growing in the community. In the following sections, we compare JiST/SWANS to the well-known ns-2, both in terms of architecture, implementation details and simulation results.

3. COMPARISON

Before we compare actual simulations we want to highlight some of the important aspects that differentiate JiST/SWANS and ns-2. Both packages provide network simulations using a discrete event-based approach. This fundamental similarity allow us to focus only on the key aspects that differentiate these two simulators. We assume that readers will be familiar with the basic principles of discrete event based simulators and ns-2 in general.

3.1 Architectural differences in JiST

The most obvious difference between ns-2 and JiST/SWANS is the general approach to implement simulation logic. While ns-2 provides interesting possibilities thanks to the integration of scripting capabilities with Tcl one can say that the approach of writing an object oriented simulator core in C++ is a standard approach to implementing event driven simulators. Alternatives to this approach would be domain specific programming languages or simulation environments that enforce timed execution by taking over system services such as threading. The developers of JiST call their solution a virtual machine based simulator. By rewriting appropriately marked Java bytecode during class loading it is possible to have registration and temporally ordered execution of simulation events. While the virtual machine itself does not know (or need any changes to support) the temporally ordered execution of simulation events we see that JiST is implemented at a very low level which keeps the overhead for maintenance of simulation logic relatively small. In fact the core logic of JiST is implemented by just a scheduler component, a sophisticated bytecode rewriter and some marker interfaces and utility functions. No support for special protocols or event serializations are necessary to run simulations with "Java in Simulation Time".

The idea of using a Java virtual machine as a simulation platform and bytecode rewriting of method calls as a means to register events is a middle ground between having a custom language and a classic simulator library. Since the virtual machine is left unmodified JiST can benefit from all the features provided by modern just in time compilation like profiling based optimizations and aggressive inlining. For example profile based inlining during runtime is a feature which is clearly interesting for simulators because codepaths might not be predictable during compile time. A compiled simulator driven by a scripting language would have a hard time to replicate such functionality. The easy integration of scripting facilities on top of the virtual machine by means of libraries like Jython is another nice benefit of running inside a Java virtual machine. Other noteworthy features are efficient garbage collected memory handling, comfortable cross platform support, opportunities to benefit from future enhancements of virtual machine technologies and prospective free optimizations for architectures that might not even exist yet.

Memory management in particular is an area where JiST not only benefits from things like parallel generational garbage collection but where it also provides means for developers to reduce the resources needed for simulations. So called *timeless* objects are objects that can safely be exchanged between entities without the need to clone them. In order to guarantee the temporal integrity of the simulation a developer must guarantee the immutability of an object when declaring it as timeless. This restriction however is not enforced by Java so it is the sole responsibility of a simulation developer to fulfill this guarantee. The amount of memory that can be saved in this way justifies this risk though. Other simulators like ns-2 have to clone objects very aggressively in order to guarantee temporal integrity across entities.

3.2 Architectural differences in SWANS

When focusing on SWANS, we also see some distinctive advantages compared to ns-2. The implementation of movement and smarter data structures for faster wireless signal transmission are two main highlights that differentiate SWANS from ns-2.

The mobility of nodes in a field is implemented in SWANS not as a strictly event based subsystem but rather with a time-stepped paradigm. A user can specify a desired granularity and SWANS will update the position of nodes on the field accordingly. This is a deviation from a strict event based model as it is practiced in ns-2. Here the exact positions of nodes on a field are computed when an event needs to know positions on the field. Heuristics could be used to limit the calculation of position updates to an area where a particular event is relevant. But the naive method is to compute all positions on the field. As long as the number of nodes and the amount of traffic is low this is not a disadvantage. A time-stepped model can even perform worse

[1]Extensions will be made publicly available under http://www.vanet.info/

than a strict event based model because position updates always happen as explicit events at the specified granularity, whereas ns-2 calculates positions only when needed (e.g. because a packet is to be sent). With large numbers of traffic and/or nodes it is obvious that the limited frequency of position updates in a time-stepped model scales better than the strict event based model which has to update positions of numerous or even all nodes for every event that needs position information. Unrelated to the time-stepped approach to mobility we see another advantage in the fact that SWANS calculates position updates not only based on precomputed trace files but usually computes mobility on demand during the simulation. This setup allows more flexible and interactive mobility models. For example, an application where cars exchange and use congestion warnings across vehicular networks can be simulated easily in SWANS because changing the direction of a node based on simulation data is readily possible in SWANS.

The time-stepped mobility subsystem also benefits from another optimization that reduces the cost of sending radio signals. Wireless nodes in ns-2 register themselves on a channel to receive radio signals when a node is sending. It is then the responsibility of each node to compute the signal strength in relation to the sender position and to decide if a signal can be received or not. SWANS employs a method called *hierarchical binning* to limit the number of nodes that have to compute the signal strength at their receiver to those that are actually within the signal range of a sender. A data structure that keeps track of the positions of nodes on the field is kept organized in a way that enables SWANS to easily address nodes within a maximum range of a given signal transmission. Only the nodes in this area have to use the full signal propagation algorithms to calculate actual reception success, noise accumulation and so forth. Due to the fact that the time-stepped mobility subsystem keeps the positions of nodes on the field updated we do not have to do any position update at all as part of such sending operations.

The differences between JiST/SWANS and ns-2 are certainly not limited to the highlighted features we outlined above. In numerous cases the authors of JiST/SWANS have for example modeled the design of subsystems after the GloMoSim simulator and not after ns-2. In other cases JiST/SWANS for example tends to use single precision floats where ns-2 prefers double precision. Also some functionality might be missing in one simulator or might only be available as a third party patch.

3.3 Infrastructure, community and prospects

An aspect that is not architectural but concerns the infrastructure surrounding JiST/SWANS and ns-2 might be an issue for prospective users of JiST/SWANS. While the availability of extensions and additional features seems to be less of a problem as the user base of JiST/SWANS grows it might be a problem that the core simulator itself sees little maintenance from its authors. This has already led to small incompatibilities with recent releases of the Java runtime environment, which required a set of patches against the rewriter and the bytecode engineering library in JiST. Also an automated process of validating SWANS against known scenarios does not exist for JiST/SWANS. While it would be unfair to discredit the potential quality of JiST/SWANS merely by the absence of such tests, it is nevertheless not inspiring confidence.

Independent investigations into the validity of results generated by JiST/SWANS compared to equivalent results from other simulators are thus useful and might help to induce confidence into JiST/SWANS. Confirmation of the advertised performance advantages and documentation of experiences in using JiST/SWANS for applications beyond the scope of the original codebase might also be useful for a wider audience to help qualify the usefulness of JiST/SWANS. For some situations, like simulations of very large scenarios of MANETs, JiST/SWANS seems to be one of the only available options. However, an assessment of the applicability of JiST/SWANS to scale typical MANET simulations to large sizes is an open question, which we will address in the next section.

4. SIMULATIONS

As the main focus of our studies lies in the simulation of vehicular networks we decided to put an emphasis on the scalability and validity of MANET simulations with JiST/SWANS compared to the baseline of ns-2.

4.1 Setup

On one side we used a standard distribution of JiST/SWANS version 1.0.6 which we enhanced with a custom driver mechanism, the CGGC routing protocol [4], and support for more recent releases of the Sun Java runtime environment. CGGC is a greedy-based geographic routing protocol that employs temporary storage of data packets as a recovery strategy from local dead ends, which makes it particularly suitable for inter-vehicle communication. These changes, among other enhancements, are publicly available from our webpages. On the other side we used ns-2 at version 2.29 in conjunction with the well known Monarch Wireless and Mobility Extensions. This version was built with a recent version of gcc version 3.4. For JiST/SWANS we used the ecj compiler provided by the Eclipse SDK version 3.2 in conjunction with a Sun Java Runtime Environment at version 6.0.1. All simulations were executed on a Gentoo Base System version 1.12.6 running a Linux Kernel version 2.6.15 on a machine with Pentium 4 CPU at 3.0 GHz and 882 MB of RAM.

4.2 Scenario

The ability to scale simulation scenarios to large numbers of nodes is an important concern for our future ability to simulate large scenarios of vehicular networks. Therefore we decided to focus our investigations on a scenario with a growing number of nodes in a simulation. To validate the correctness of simulations performed with JiST/SWANS we set up matching scenarios in JiST/SWANS and ns-2 to compare the results. At the same time we used these simulations to collect data about the performance characteristics of the two simulators.

To keep the number of variables down we decided to set a fixed average connectivity of 7. This value was chosen arbitrarily in order to ensure that routing protocols have a good chance for successful packet delivery. The field size for our scenarios grew subject to the average connectivity of the nodes and a wireless range of 250 meters, using equation (1).

$$\rho = \frac{(\pi * r^2)}{\left(\frac{w*h}{n}\right)} \tag{1}$$

The actual measured connectivity in our simulation was between 9 and 10, which is mainly due to border effects and the tendency of random waypoint model to concentrate nodes in the center. The number of nodes was set in a range between 200 to 1000 nodes. Again these values were chosen arbitrarily, this time in order to keep the runtimes of the simulations down, in particular because of ns-2. During our experiments, we have easily simulated scenarios with more than 10.000 nodes using JiST/SWANS. For the purposes of this paper we provide simulation results as the average of 10 redundant passes with identical parameters but differing random seeds. A trivial test with fixed random seeds showed that our version JiST/SWANS produces exactly repeatable simulations. JiST/SWANS and ns-2 use their respective default sources for pseudo random numbers, which in case of JiST/SWANS is the standard java.util.Random facility provided the the Java runtime.

Nodes	200-1000
Transmission Range	250m
Field	2368-5296m
Mobility	Random Waypoint
Max speed	20m/s
Min speed	1m/s
Pause	0s
Duration	120s
Warmup	20s
Cooldown	10s
Noise	Independent
Pathloss	Tworay
Fading	None
Packetloss	None
Traffic	1 packet/min
Routing	CGGC, AODV
Beaconing	1 Hz
Packet caches	Unlimited
Destination radius	100-300m

Table 1: Key settings of the simulated scenarios

It would take too much space to list all settings exhaustively. Therefore we close this section with Table 1, which lists some key settings. We hope that this specification allows meaningful reproduction of our simulations [11].

4.3 Qualitative comparison

We applied the previously described scenarios equally to ns-2 and JiST/SWANS and used two routing protocols to run the actual simulations. As mentioned before, the first protocol is a position-based routing protocol called CGGC. Geographic routing is an important area of research for our group and for research in vehicular ad hoc networks and was thus of special interest to our needs. Our implementation of geographic routing is based on the equivalent logic both in SWANS and ns-2 to be able to really compare the qualitative results. The second protocol we are testing is the more commonly known AODV protocol, using standard implementations that come with the distributions of both JiST/SWANS ans ns-2. Hence, we can compare on a level where we know that the routing code is equivalent and on a level where we just use a provided implemention, without looking into detail, like many beginners would do.

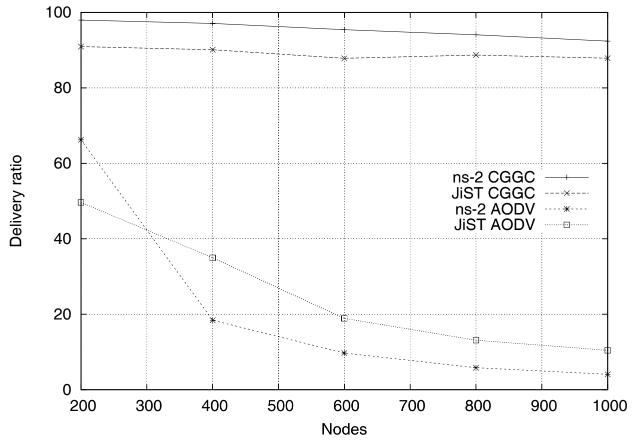

Figure 1: Delivery success ratio

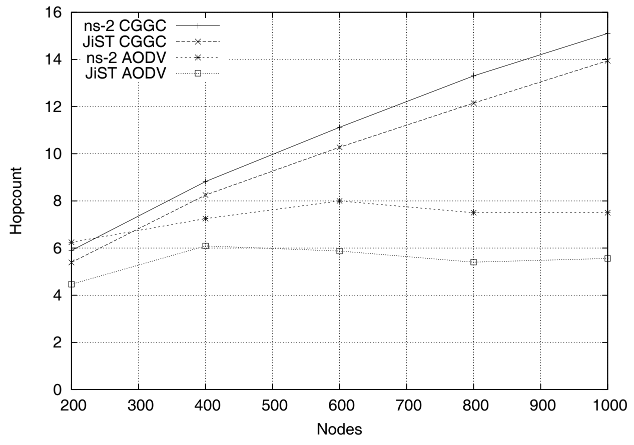

Figure 2: Average hopcount of message transfer

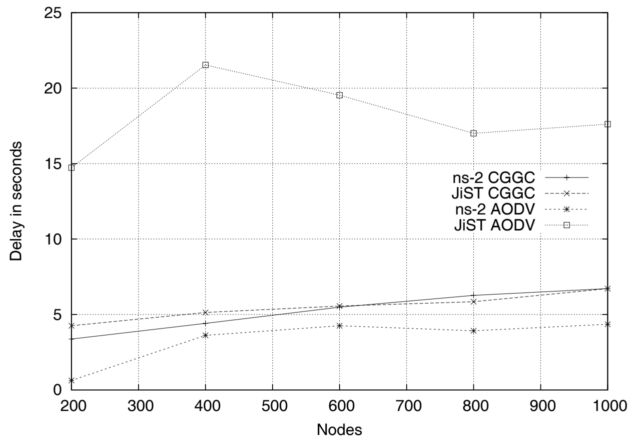

Figure 3: Average message delay

Based on previous experiences we expected to see the geographic routing protocol to perform very nicely in a situation with good connectivity between the nodes and relatively high mobility of the nodes on the field. The AODV routing protocol on the other hand was expected to suffer with increasing number of hops between origin and destination. Indeed these expectations were fulfilled in both JiST/SWANS and ns-2 with qualitative differences of only a few percent points. For CGGC, this is true for metrics such as delivery ratio (Figure 1), hop count (Figure 2) and average delay from source to destination (Figure 3). In case of AODV, the picture is a bit different. In particular, delays are significantly higher in SWANS, though the number of hops is at least comparable. Therefore, we assume that the AODV implementation in SWANS employs more relaxed caching settings. This is partly backed by the number of successfully delivered packets, which decreases slower for AODV in SWANS compared to AODV in ns-2.

4.4 Performance analysis

The performance characteristics of ns-2 nd JiST/SWANS do differ quite a bit. Due to the different implementations of movement in the simulators it is not easy to directly compare the raw execution times. As JiST/SWANS calculates the movements of all nodes during the runtime of the simulation there is no way to subtract the time it takes to calculate these movements from the other parts of the simulation. For ns-2 though, the subprogram that calculates the movements of nodes is not part of the main simulation. In fact it is commonly the case that movements which have previously been rendered into a trace file are reused repeatedly in different simulation runs. This practice is encouraged by the fact that a subprogram like the commonly used *setdest* application seem to be rather unoptimized in terms of runtime performance. In Figure 4 we compare the cumulative execution times of all subprograms that compose a simulation of CGGC routing. We list ns-2 two times, once excluding the time required to generate movements for ns-2, in order to reflect a common usage pattern for ns-2 that reuses movement traces across different simulations. The relevance of this practice in relation to performance characteristics of ns-2 is also clearly visible.

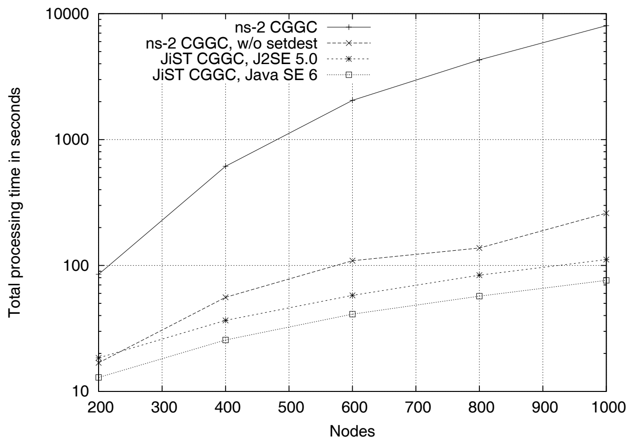

Figure 4: Processing time of CGGC routing

When we compare JiST/SWANS against ns-2 without the time required to precompute movements, we see that CGGC routing in JiST/SWANS scales almost linearly with the number of nodes on the field. Remember that the connectivity as a function of node count and field size is kept constant as we increase the numbers of nodes on the field. Since we also keep additional factors like packet traffic tied to the node count of the scenario, this behavior represents an expected optimal scaling behavior for a reactive routing protocol like CGGC. With ns-2 on the other hand we see execution times grow faster than the number of nodes we add to the simulation. Even worse, when we add the time required to compute the movements in ns-2, as is done in JiST/SWANS, we see that ns-2 shows significantly worse scalability compared to JiST/SWANS. This graph is drawn on a logarithmic scale for better readability.

In our opinion a good part of the performance advantage of JiST/SWANS can be traced to the underlying Java virtual machine. To visualize this assumption we included performance data of JIST/SWANS using a different Java runtime. Exactly the same binaries/bytecode have been used to run simulations with JiST/SWANS under a Sun JRE version 1.5.0_11 in addition to our current default environment, a Sun JRE at version 1.6.0_01. Clearly the performance of the underlying VM is a significant but mostly constant factor. As has been mentioned in a previous chapter, several architectural advantages such as runtime profiling, efficient memory management and the reduced overhead for managing simulation logic by embedding the simulator as a bytecode rewriter contribute to the comparatively good performance of JiST/SWANS. The effect of merely updating the Java VM without changing any algorithmic aspects of JiST/SWANS illustrates this influence nicely. Also note that the original authors of JiST/SWANS used only a Sun JRE at version 1.4 to benchmark their original release.

A second important aspect is the performance of the mobility subsystem. It is clear in Figure 4 that ns-2, and the setdest subprogram to be more precise, perform particularly bad when movements have to be calculated for each simulation. The efficiency of the time-stepped mobility in JiST/SWANS and the usage of hierarchical binning strategies are one explanation for the significant differences in performance. On the other hand we do not see good reasons for the setdest subprogram in ns-2 to perform as badly as it does. Our observations indicate that setdest calculates large amounts of metadata that is not necessary for the raw simulation. We also assume that setdest simply was not optimized for runtime performance. As it stands however the current incarnation of setdest is a serious obstacle to simulating large numbers of varying or even dynamically changing movement models with ns-2.

When it comes to performance of AODV, we find that the gap between the simulators is not too big, at least when we consider only the effective simulation time without the mobility calculation in case of ns-2. However, since SWANS implicitly includes mobility, the results slightly support SWANS. In any case, with growing number of nodes, ns-2 requires more than linearly increased processing time (Figure 5).

A third relevant aspect is memory usage of the respective simulators. In Figure 6, we see the memory consumption of the two simulators is drastically different. The fact that ns-2 has to manage such huge amounts of memory is a distinctive

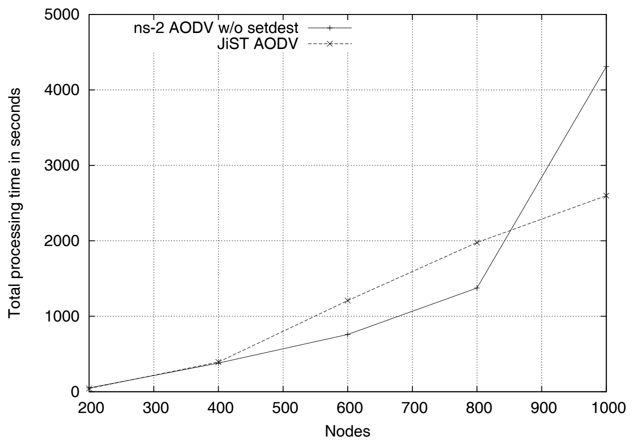

Figure 5: Processing time of AODV routing

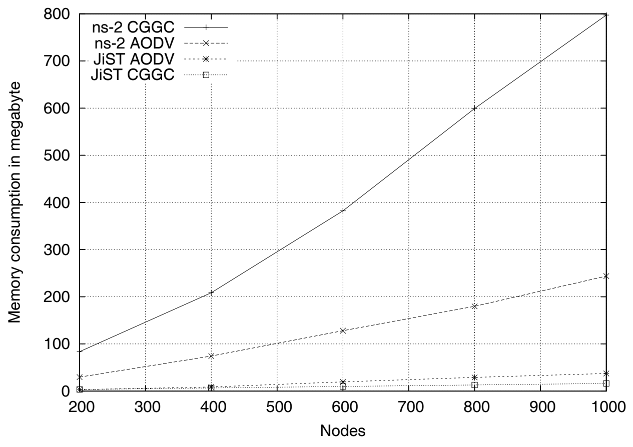

Figure 6: Memory usage

disadvantage compared to JiST/SWANS. The possibility to use Timeless Objects in JiST/SWANS enables large savings in memory usage compared to ns-2 where objects are copied far more aggressively in order to protect the integrity of the simulation. The aspect of memory usage is not just an issue of performance. Memory requirements tend to be a limiting factor in the ability to simulate large scenarios with several thousand participants. Our simulation with ns-2 and CGGC routing with 1000 nodes already required close to one gigabyte of memory. While there is likely some room for improvement in our implementation of CGGC for ns-2 we can safely say that JiST/SWANS scales much more gracefully in this area than ns-2.

5. CONCLUSION

With our simulation study, we show that JiST/SWANS is able to produce very similar results to those of ns-2 when we use the geographic routing protocol. This is mainly due to the fact that both implementations were developed by ourselves. In contrast to that, the AODV implementations that come with both simulators partly show diverging behavior. In summary, the qualitative comparison clearly indicates a basic problem of simulations: Comparing results between different implementations on different simulation tools is delicate. Even if implementations are conceptually identical, other components still have some influence on the results.

However, in general, one can see that JiST mostly shows superior performance, which allows for more detailed simulations in shorter times. Even huge simulations with thousands of nodes become possible, mainly because of the very low memory footprint. Nevertheless, the particular implementations also have implications on performance.

In summary, we could provide a number of arguments that support the usage of JiST/SWANS for ad hoc network simulations, though there are still questions remaining. For instance, results from other layers of the simulation, like medium access or radio propagation have not been addressed. In the future, we also plan to extend JiST/SWANS with more accurate environment models like raytracing-based radio propagation.

6. REFERENCES

[1] Network Simulator ns-2. http://www.isi.edu/nsnam/ns/, 2004.

[2] R. Barr, Z. Haas, and R. van Renesse. *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad hoc Wireless, and Peer-to-Peer Networks*, chapter 19 - Scalable Wireless Ad Hoc Network Simulation, pages 297–311. Auerbach, 2005.

[3] R. Barr, Z. Haas, and R. van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software Practice & Experience*, 35(6):539–576, 2005.

[4] Christian Maihöfer, Reinhold Eberhardt, and Elmar Schoch. CGGC: Cached Greedy Geocast. In *Proc. 2nd Intl. Conference Wired/Wireless Internet Communications (WWIC 2004)*, volume 2957 of *Lecture Notes in Computer Science*, Frankfurt (Oder), Germany, Feb. 2004. Springer Verlag.

[5] David R. Choffnes and Fabián E. Bustamante. An Integrated Mobility and Traffic Model for Vehicular Wireless Networks. In *Proc. of the 2nd ACM International Workshop on Vehicular Ad Hoc Networks (VANET)*, Sept. 2005.

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

[7] OPNET. OPNET Modeller Homepage. `http://www.opnet.com/products/modeler/home.html`.

[8] G. Riley. PDNS Website. `http://www-static.cc.gatech.edu/computing/compass/pdns/`.

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

[10] SNT. QualNet Product Homepage. `http://www.scalable-networks.com/products/qualnet.php`.

[11] Stuart Kurkowski, Tracy Camp, and Michael Colagrosso. MANET Simulation Studies: The Incredibles. *Mobile Computing and Communications Review*, 9(4):50–61, Oct. 2005.

[12] B. Zeigler and S. Mittal. Modeling and Simulation of Ultra-large Networks: Methodology Responds to Challenges. In *ULN Workshop*, November 2001.

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