

Comparison of Network Simulators Revisited

David M. Nicol
Dartmouth College

May 20, 2002

Abstract

In January 2002 a paper was published discussing the simulation tool **JavaSim** and its relationship to **ns2** and to **SSFNet**. The intent of the experiments were to assess the scalability of these tools. We re-examine those experiments, and show that the experimental methodology does not increase intrinsic workload as the model size grows. We furthermore find that the experimental design pushes *all* the simulators into regimes for which they were not designed, resulting in atypical behavior for each. After modifying the simulators, we re-examine the original experiments, and new experiments that increase the intrinsic workload with increasing model size. We find that (i) **ns2** is fastest, but requires the most memory, (ii) **JavaSim** is significantly slower than the other simulators, but requires significantly less memory than the Java implementation of **SSFNet**, (iii) that a C++ implementation of **SSFNet** is nearly as fast as **ns2**, and uses the least memory among all the simulators.

1 Introduction

JavaSim[5] is a simulation package developed at Ohio State University (and now at home at University of Illinois) implementing a component-based architectural approach to system description. It is a rich system, with many modeling components related to modeling Internet protocols.

In November 2001, the **JavaSim** team widely distributed an announcement of a study comparing **JavaSim** with other simulators, drawing the conclusion that **JavaSim** has the best scaling properties. They published a paper on this in the 2002 Communication Networks and Distributed Systems Modeling Conference [4]. Both the posting and the paper

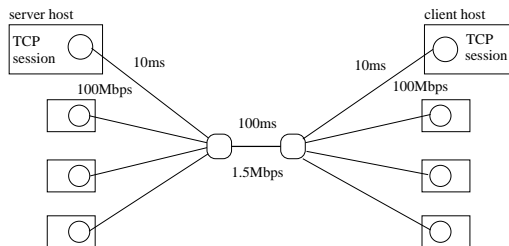


Figure 1: Dumbbell architecture: multiple TCP sessions sharing a bandwidth-limited bottleneck link between routers

describe experiments that quantify performance differences. **ns2**[2] and **SSFNet**[6] were singled out for comparison.

The architecture studied is one frequently seen in small scale studies, a *dumbbell*, illustrated in Figure 1. We will go through the model in some detail, as it is important in understanding our analysis of the experiments. Parameter N gives the number of TCP sessions. The model assumes a unique host for every TCP session endpoint; there are N hosts on one side of a simple network, sharing TCP sessions with N hosts on the other side of that network. Each host has a dedicated $10Mbps$ connection to a router; all hosts on the same side of the network attach to the same router. Each router has $4096 \times N$ bytes of buffer space. Packets have size 1000 bytes. The link between these two routers has $1.5Mbps$ capacity. The traffic model is that each session is immediately established, after which the sender continuously sends as much as possible, subject to TCP rules.

The experimental data presented by this work typically plots running time as a function of N , up to $N = 10000$. They typically show **JavaSim**'s execution time increasing linearly in N , with **ns2** and

SSFNet's exploding upward somewhere well short of $N = 10000$.

With an eye towards understanding the performance of **SSFNet** described in these reports, we studied carefully the experiments they describe. We find that

- The reported experiments used different TCP parameters for each of the simulators, e.g. maximum send window size, initial value of `ssthresh`, maximum segment size, and so on. We find that nevertheless, with many connections, the simulators' behavior and execution times are insensitive to this difference.
- The experiments do not increase the amount of intrinsic work to do in the model as the model size grows. Consequently the scalability evaluated is fundamentally a simulator's ability to hold a model in memory.
- For large numbers of sessions, the topology studied is atypical. It is atypical in the number of physical connections a router has, and it is atypical in the magnitude of the round-trip-time observed by the simulated TCP sessions. Because of differences in modeling TCP timeout timers, large RTT values cause **SSFNet** to spend almost all of its time managing timer events, whereas **JavaSim** and **ns2** do not. With typical RTT values this difference does not exist.
- On this specific architecture, with a modification to make workload scale with numbers of components,
 - **ns2** is fastest, on those problem sizes it can handle.
 - **JavaSim** is slowest, by an order of magnitude.
 - A Java implementation of **SSFNet** runs at half the speed of **ns2**, but has smaller memory demands.
 - The C++ implementation of **SSFNet** uses the least amount of memory, and has an execution time close to that of **ns2** (relative to the other simulators).

All of the simulation runs we report in this paper were taken on an IBM Thinkpad T23, with 1.0Gb main memory. The operating system is Redhat Linux 7.2, the Java runs use Sun's JDK 1.4.0, beta 3. The

experiments were all conducted with swapping turned off, so that the simulation models are constrained to reside in main memory.

2 Establishing a Plumb Line

In any experimental study that compares the performance of two or more systems, it is essential that the comparison be done when the systems are working at *the same task*. Our first question is whether the reported experiments do.

The **JavaSim** comparison page goes to some pains to describe what was done, in a commendable effort to make their results reproducible. Following directions, we downloaded and built both **JavaSim** and **ns2** without any modification, and began our experiments with these installations.

In order to first validate that all three simulators were indeed doing the same thing on the reported experiments, we instrumented the control scripts to periodically print out, for every TCP writer, the next segment the writer will send (assuming that they increase linearly, beginning at 0), e.g., the upper edge of the send window, in units of MSS. This value is commonly used to describe the progress of a TCP session, as a function of time. We found that the three simulators advance the window at three very different rates. After some investigation we determined that all three tools contain *different* initialization constants for TCP parameters. In particular, they varied in the values used for maximal segment size, maximum receiver window size, maximum sender window size, `ssthresh`—the threshold governing when TCP transitions from slow start to congestion avoidance, and the TCP header size. Since performance comparisons need to be conducted on systems that are doing essentially the same thing, we modified the **JavaSim** and **SSFNet** parameters to be in line with those of **ns2** (simply because we are more familiar with the **JavaSim** and **SSFNet** code body). Following these changes, the simulators produced behavior that was very close to each other, if not exact.

Oddly enough, the differences in initialization constants turn out not to matter in the experiments of interest. The reason is that the congested link forces TCP windows to shrink enough so that `ssthresh` be-

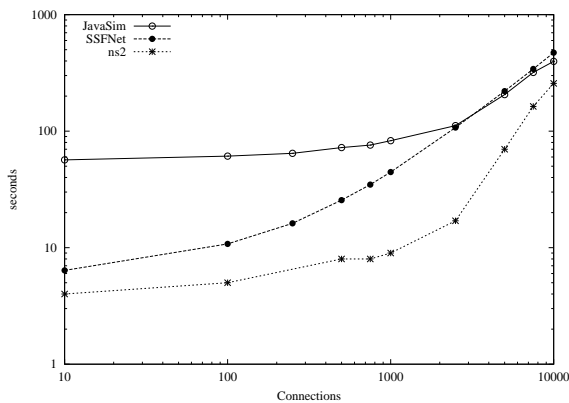


Figure 2: Comparison of model run time on dumbbell architecture, 1000 simulated seconds

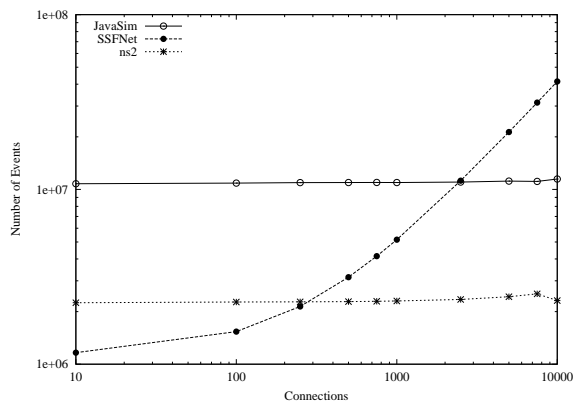


Figure 3: Number of events executed on dumbbell architecture, 1000 simulated seconds

comes small, regardless of its initial value, and upper bounds on window size are never threatened by window behavior. We empirically determined that the *sum* over all sessions of the upper edge of the send window is the same for the different simulators, although the *distribution* of those edges are noticeably different. At the level of throughput one can accept that the simulators are “doing the same thing” for large numbers of connections.

3 Original Performance Studies

We ran the original experiments for 1000 simulated seconds, on model sizes from 10 connections to 10000 connections. (The memory on our test machine is significantly larger than the one on which the original experiments were run. The sort of explosion in **SSFNet** execution time which was the focus of the original study is pushed off to larger numbers of connections). Comparison of run-times on runs of 1000 simulated seconds reveals interesting behavior, shown in Figure 2. **JavaSim** and **ns2** have relatively constant execution times as the problem size grows, and then their run-times increase significantly for large numbers of connections. By contrast, the runtime of **SSFNet** increases steadily. There is a mystery here.

Figure 3 adds to the mystery, by plotting for each simulator the number of events it executes in the course of simulating 1000 seconds. What is interesting here is not the raw event counts (which will

vary between simulators, depending on what constitutes an event), but how the event counts for a given simulator behave as a function of model size. We find answers to these mysteries in the next section.

4 Experimental Architecture

The dumbbell architecture is commonly used in simulation studies of TCP. However, the pitfalls of this architecture for studying scalability was discussed already in [1], where we noted that one must increase link bandwidth as the number of connections N increases if one is to obtain meaningful results. **JavaSim** experiments increase a router’s *buffering capability* as N grows, but not the bandwidth.

This experimental design has ramifications. TCP adapts to the high congestion by shrinking the send window size. To a first approximation, TCP sends a window’s worth of packets every round-trip time. A packet arriving at the buffer will find it nearly full, and will wait in the buffer until everything in front of it has been sent. Since the buffer grows linearly in N (but bandwidth does not), the amount of time a packet stays in the buffer grows linearly in N . Specifically, if a 1000 byte packet arrives at the end of a nearly full buffer, then its wait in the queue is the time needed to transmit $4096 * N - 1000$ bytes. With a bandwidth of $1.5 Mbps$, that takes $0.02184 * N - 0.005$ seconds. Adding this to the round-trip latency of 0.24 seconds, the table below gives estimated RTT times as a function of N , using this formula.

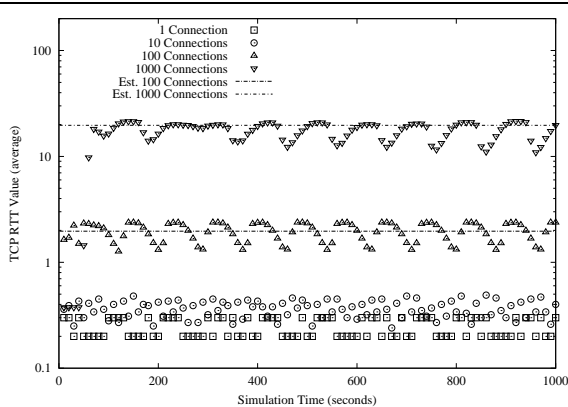


Figure 4: RTT variable from **ns2**, for various numbers of connections

$N =$	10	100	1000	10000
RTT=	0.46s	2.42s	22.08s	218.6s

We see that for anything but the smallest N , the round-trip time is firmly dominated by this wait in the buffer and so is linear in N , say $RTT \approx \alpha N$. Figure 4 plots measured values of the **ns2** `rtt_` variable times its `tcpTick_` variable (the product is the round-trip-time estimate), averaged over all connections, as a function of simulation time, and number of connections. Observe that the y-axis has logarithmic scale. This variable is the TCP session’s estimate of the round-trip delay, a value which is used in a formula to compute a packet timeout period.

If we assume that the computational work associated with sending a window’s worth of packets across the network and receiving their acknowledgements (exclusive of event-list costs) is a constant, W , then the rate per unit simulation time at which the simulator is performing this work for one session is $W/(\alpha N)$, with a total model computational cost rate of W/α . What this says is that the packet-oriented workload per unit simulation time is independent of N . Now we understand why the **JavaSim** and **ns2** event count curves in Figure 3 can be constant functions of N . The question remains why the **SSFNet** event curve is not constant?

The analysis above excludes consideration of the timeout timer. In **JavaSim** and **ns2** at the point a packet is transmitted a timer is scheduled to fire after the calculated time-out interval, and is canceled when the acknowledgement arrives. This im-

plies there is a constant amount of work associated with the timer, per RTT delay. **SSFNet** follows the actual BSD TCP implementation in associating one timer *per host*. This timer fires every 0.5 second, and sweeps through all the active TCP connections, looking for any that might have timed out in the last 0.5 seconds. Thus, for **SSFNet**, there are $2RTT$ timer firing events associated with every round, the number of events executed per unit time, per session, grows linearly in N , just as we have seen. As RTT grows with N an increasing fraction of events are timer events. This explains **SSFNet**’s behavior in Figures 2 and 3.

We have still to explain why, if the number of events performed is insensitive to model size, the execution times of both **ns2** and **JavaSim** increase dramatically for large number of connections. In both cases the causes were found using run-time profilers. In the case of **ns2**, when a packet is dropped from a drop-tail queue, a linear search *from front to back* of the packet queue was executed, comparing the address of the packet to drop with the addresses of packets in queue. Since the packet to be dropped is always at the back, and since the buffer size (and hence number of packets in queue) increases linearly with the number of connections the execution time becomes dominated by this search. George Riley identified the cause from our run-time execution profiles, and provided a patch to correct this. In the case of **JavaSim**, we identified the fact that every **JavaSim** component searches linearly through all its ports when a message passes through. In the model under study, the number of ports in a router increases linearly with the number of sessions, and this scanning eventually dominates run-time. We reported this finding to the **JavaSim** team, and Hung-ying Tyan provided a patch. Likewise, receiving a message **SSFNet** executes a linear scan of ports at a NIC interface to determine its origin. A realistic system will not have hundreds or thousands of ports, nevertheless in this experiment it does. We patched the implementation to use a simple linked list (which gives rise to the linear search) if the connection count is less than 20, and use a hash-table otherwise.

As a result of these observations we modified all three simulators to remove the linearizations that defeat scalable behavior on this particular experimental design. Figure 5 replots the runtimes using the modified simulators. Variation in execution for a given problem was small, so we do not report the

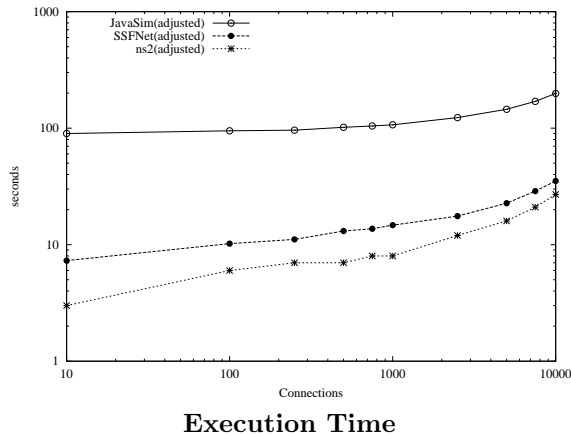


Figure 5: Original dumbbell data, with modified simulators

insignificant deviation. For the problem sizes considered, **JavaSim** consistently runs about 10 times more slowly than **SSFNet**, which in turn runs more slowly than **ns2** by a factor that decreases with problem size. We know that **JavaSim** will run larger still instances of this architecture than will **SSFNet** or **ns2**; however, because of the limitations of this architecture we defer a study of memory demands to an architecture where workload scales with size (and where RTT does not grow without bound).

5 Another Look at Scalability

With our improved understanding of TCP’s behavior on the dumbbell architecture, we modify the original experiments slightly to create a model for studying how execution performances behaves as well. We don’t like the dumbbell as a vehicle for this study, but shirk from trying to write **JavaSim** and **ns2** scripts to build more complex network architectures. It should also be noted that we’ve made no effort to change the model scripts from those identified in the original study.

5.1 Scaling Architecture

We modify the dumbbell description so that the bottleneck bandwidth increases linearly with the num-

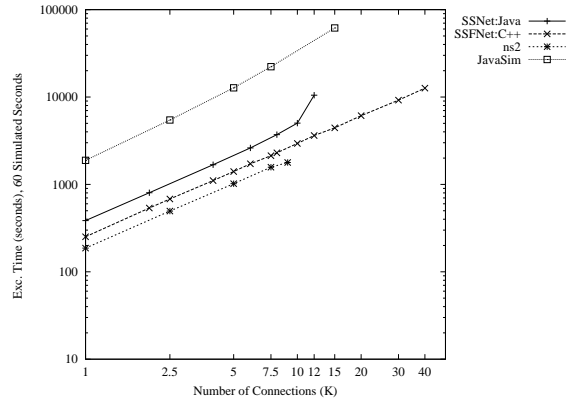


Figure 6: Execution costs of **ns2**, **JavaSim**, and two **SSFNet** implementations, dumbbell architecture with scaled bandwidth

ber of connections (as $N \times 1.5Mbps$), in addition to the original increase in buffer size. Thus, as we add another connection to the model, we add the same buffer and bandwidth resources as are available to the only connection in a one-session model. The flows will interact in the buffers, but TCP session behavior will be much more deterministic.

5.2 Execution Speed

We first look at simulator execution time behavior. We are interested both in how fast the different simulators are, and how that speed changes with model size. For this study we include our C++ implementation of **SSFNet**; this implementation has been validated to give almost exactly the same TCP behavior as the Java implementation of **SSFNet**. We have traced such differences as exit (on models with large numbers of connections) to differences in Java’s math and C++’s, and differences in ordering the execution of events with exactly the same time-stamp (note that for these studies both implementations of **JavaSim** use an extended integer type for the clock).

Figure 6 plots execution time for **ns2**, for both our implementations of **SSFNet**, and for **JavaSim**. We simulate each model size for 60 simulated seconds, plotting execution time after the model is built for each model size. Build time can be significant for large models, but we discount it as build time ultimately can be amortized over a long enough run.

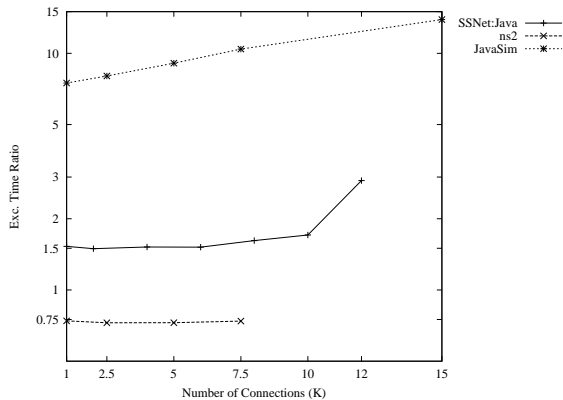


Figure 7: Execution time of **ns2**, **Java-SSFNet**, and **JavaSim** relative to **C++SSFNet**

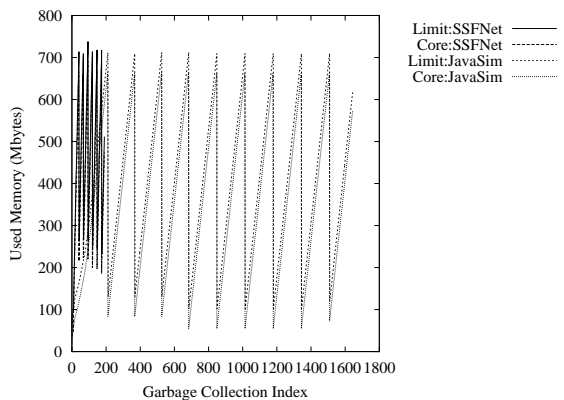


Figure 8: Memory use behavior in **JavaSim** and **SSFNet**

This log-log graph describes model sizes and execution times that span three orders of magnitude. The slope on each simulator’s curve is close to 1 and increases slowly in the number of connections. This means that before memory limitations begin to dominate, each simulator’s execution time is almost linear in the problem size, which indicates that the internal data structures and algorithms are scalable for this problem suite.

Memory and computation limitations show up. The **ns2** plot stops at 9000 connections; a 10000 connection run aborted, due to its attempt to use virtual memory when swapping is turned off. The **JavaSim** run could in principle continue through larger problem sizes, but the last point plotted (at 15000 connections) took over 17 hours to compute; there seemed to be little point in running it for the estimated two days needed to complete a 40000 connection data (obtained by **C++SSFNet** in 3.5 hours). The Java version of **SSFNet** shows the effects of garbage collection overhead as it approaches the largest problem size we attempted for it, 12000 sessions.

Figure 7 is based on this same data, but shows ratios of execution time, using **C++SSFNet** as base (in the denominator). Here we see clearly that on this problem suite **ns2** requires 75% the execution time of **C++SSFNet**, that **Java-SSFNet** requires 150% the execution time of **C++SSFNet**, and that **JavaSim** execution requirements grow more quickly as a function of model size than does **C++SSFNet**. With 15000 connections the **JavaSim** execution time is 14 times larger than the **C++SSFNet** execution time.

5.3 Memory Requirements

Finally we turn to an evaluation of memory demands of these simulators, on the modified dumbbell architecture.

Memory use of the Java simulators can be determined by using a run-time flag to write garbage collection to file. Each line of a trace describes one garbage collection activity, including when it happened, how long it took, the type of action (minor or full reclamation), the number of bytes in use when the garbage collection action was triggered, and then the number after the reclamation and packing. The garbage collector in use on our JDK 1.3 and 1.4 versions is described in detail at [3].

To illustrate behavior, Figure 8 plots the “trigger” and “packed” value time-series for **JavaSim** and **SSFNet**, on the 2500 connection model, over 60 simulation seconds. Data from every garbage collection is recorded; “time” in the time-series is the index of the garbage collection action. The squashed pattern that ends at $x = 200$ represents **SSFNet**, the elongated pattern represents **JavaSim**. For both runs the heap size was pre-allocated to be 750Mb; we see that the big drops in memory use (due to full GC actions) are triggered as the used byte count approaches that limit. The elongated pattern of **JavaSim** tells us that it employs far more objects than **SSFNet**, but that these objects tend to be recovered in the minor reclamation phase of garbage collection, far more so than **SSFNet**. In this plot the number of major reclamations is nearly the same in both simulators.

The used memory at the end of a major garbage collection contains only “live” objects. One can also think of it as a snapshot of intrinsic model memory needs. In a Java system considerably more memory than this is needed if the program is to be run without being overwhelmed by garbage collection overhead. However, for the purposes of computing the intrinsic model demands on memory are, we will measure the average “packed” value, which is independent of the heap allocation (so long as the heap is large enough).

In Figure 8 the low points in the **JavaSim** curves are about a third the value of the low points in the **SSFNet** curve, indicating that on this problem **JavaSim** has a significantly smaller core set of live objects than does **SSFNet**. It is reasonable to assume that the incremental memory cost of adding a connection is constant. Returning to the runs represented in Figure 6 (for which garbage collection data was saved) we computed the average core memory demand for both simulators, as a function of the number of connections. For both simulators we then fit a linear regression to the observed series expressing core memory demand as a function of problem size. The slope of this line gives the incremental memory demand of one connection. For **SSFNet** the cost per connection is 53.3Kb, for **JavaSim** the cost per connection is 21.7Kb. The R^2 factor in both fits is higher than 0.99, indicating an excellent fit.

We also consider heap usage for the C++ simulators. In the case of C++**SSFNet** we automatically get a memory highwater mark from the SSF kernel; so for any given model size we can measure what demands it places on the heap. In the case of **ns2** we do something similar. We employ the C library call `sbrk()`; `sbrk(0)` reports the memory address of the current end of the heap. We instrumented **ns2** to measure the current end-of-heap and save it in a global variable, before running the main function. The stored value tells us where the heap starts before any of the model building or simulation occurs. We further instrumented **ns2** to allow one to send a command to the simulator, the processing of which calls `sbrk(0)`, subtracts off the initial heap position measurement, and returns the current heap size. Calling this function at the end of a run yields the heap highwater mark.

Linear regression on observed **ns2** runs predict a per-connection cost of 93.3Kb. Regression analysis of C++**SSFNet** runs predict a per-connection cost of 18.1Kb. Like the Java simulator regressions, both

Tool	Build	Build/Run
JavaSim	17.3Kb	21.7Kb
Java- SSFNet	27.5Kb	53.3Kb
ns2	52.2Kb	93.3Kb
C++- SSFNet	13.3Kb	18.1Kb

Table 1: Per-connection core memory demands

of these have extremely good fits.

We can also use this same methodology to assess the memory demands of these simulators after building the model but before advancing simulation time. In each case we simply set the simulation termination time to 0, and traffic memory use as before. These measurements provide an interesting contrast to the first set; whereas the first measurements are of a network that is pumping traffic as fast as TCP allows, the second set are of a network doing nothing. These extreme points give us some sense of how much of the memory is being used to describe the architecture, and how much is being used to describe the traffic. Table 1 summarizes the per-connection core costs of all the simulators.

It is reasonable to ask why such differences exist in memory requirements between simulators. We’ve done a detailed analysis of the memory usage of C++**SSFNet**. The interesting thing is that the memory associated with an individual TCP session is small; 376 bytes are used to describe the session structure, 520 bytes to describe the TCP Client, and 528 bytes to describe the TCP Server. All the rest is infrastructure to describe the host machine, the NIC, the protocol stack, managers of multiple TCP sessions and managers of multiple sockets, and so on; this latter factor dominates, by several multiplicative factors. It pays for itself in ease of management, and in amortization when multiple sessions for a given protocol stack, and/or multiple protocol stacks are involved. The structure of Java-**SSFNet** is similar; the higher overall cost is due in part to the higher memory cost of Java objects, and due in part to its use of default container classes, which double in space allocation at every expansion.

The **ns2** team tells us that its high memory use is a result of the infrastructure it provides to examine protocol behavior at runtime. The integration of Tcl and C++, with C++ objects exporting many many state variables to Tcl costs memory, but is memory

devoted to a specific functionality at which **ns2** excels.

In both the case of **ns2** and **SSFNet** it appears to us that memory is being allocated so that the infrastructure does not change much as different models are run, and that the infrastructure is designed to support many different models. In the case of **JavaSim** it appears to us as though one assembles the various “components” needed to provide specific functionality for a specific model, and so achieve a smaller memory footprint than the others on this simple model. However, the flip side is that communication between components in **JavaSim** seems to be very expensive.

6 Conclusions

We have closely re-examined the experiments comparing **JavaSim** to other simulators, in [4]. We find that the architecture and experimental methodology push *all* of the simulators considered into regimes that are atypical, and which as a result create non-scalable behavior in *all* the simulators considered. We account for these artifacts with modifications to the simulators, re-examine the original experiments, and examine a closely related set of experiments that—unlike the original set—increase computational workload as the problem size increases. We study the behavior of computational and memory demands as a function of problem size. We find that there are significant differences in demands—a factor of 5 difference between maximal and minimal memory demands, and a factor of 14 difference between maximal and minimal computational demands. These differences significantly impact the size of a model that can be simulated, and the speed at which it can be simulated. Prior to embarking on a large-scale modeling and simulation effort, it is therefore important to understand what capabilities and limitations each simulator brings with it. This paper sheds some light—but only a little—on this. To truly study scalability it is necessary to study larger and more realistic architectures than the modified dumbbell considered in this paper. We are currently working to define a baseline architectural suite for such studies.

Acknowledgements

Many people contributed to this paper, directly and indirectly. Srdjan Petrovic was the sleuth who noticed that in large instances of the original dumbbell architecture, **SSFNet** spent most of its time and memory managing timer events. Michael Liljenstam and Yougu Yuan were always at the ready to instrument Java **SSFNet** code to provide the measurements I asked for. Michael Khankin and Mehmet Iyigun did the original port of Java’s **SSFNet** to C++ as part of undergraduate theses; Guanhua Yan took on the heroic task of fixing their bugs and validating the C++ version against **ns2** and Java-**SSFNet**. Jason Liu helped considerably in performance tuning of the C++**SSFNet** system. Andy Ogielski, Jim Cowie, John Heidemann, and George Riley made valuable comments on an earlier draft of this paper. Riley made patches to **ns2** and Hung-ying Tyan made patches to **JavaSim** to deal with non-scalability issues we uncovered.

This research is supported in part by DARPA Contract N66001-96-C-8530, NSF Grant ANI-98 08964, NSF Grant EIA-98-02068, and Dept. of Justice contract 2000-CX-K001.

References

- [1] James Cowie, Hongbo Liu, Jason Liu, David Nicol, and Andy Ogielski. Towards realistic million-node internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 1999.
- [2] <http://www.isi.edu/nsnam/ns/>.
- [3] java.sun.com/docs/hotspot/gc/.
- [4] Hung-Ying Tyan and Jennifer Hou. Design, realization, and evaluation of a component-based compositional network simulation environment. In *Proceedings of the 2002 Communication Networks and Distributed Systems Modeling and Simulation Conference*, San Antonio, Texas, January 2002. Society for Computer Simulation.
- [5] www.javasim.org.
- [6] www.ssfnet.org.