Split Protocol Stack Network Simulations Using the Dynamic Simulation Backplane *

Donghua Xu
George F. Riley
Mostafa H. Ammar
Richard Fujimoto

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{xu,riley,ammar,fujimoto}@cc.gatech.edu

Abstract

We introduce and discuss a methodology for heterogeneous simulations of computer networks using the dynamic simulation backplane. *This methodology allows for exchanging of protocol information between simulators across layers of the protocol stack. For example, the simulationist may wish to construct a simulation using the rich set of TCP models found in the* ns *network simulator, and at the same time using the highly detailed wireless MAC models found in the* GloMoSim *simulator. The backplane provides an interface between heterogeneous simulators which allows these simulators to exchange meaningful information across layers of the protocol stack, without detailed knowledge of internal representation in the foreign simulator. With this method of heterogeneous simulation, new and experimental protocols can be validated and tested in conjunction with existing and accepted simulations of lower protocol layers.*

We discuss the particular problems presented by the split protocol stack model, and present our solutions. We give results of our implementation of the split protocol backplane, using the ns *simulator for the higher protocol stack layers, and the* GloMoSim *simulator for the lower layers.*

*This work is supported in part by NSF under contract number ANI-9977544 and DARPA under contract number N66002-00-1-8934.

1. Introduction

The use of simulation is becoming increasingly prevalent in the computer networking research community. The popular *ns*[7] network simulator has a very rich and complete set of TCP models, and is used in many areas of networking research, including transport protocols, queue management, scheduling policies, mobility, and multicast[5]. The *GloMoSim*[15] simulator provides detailed models of wireless MAC protocols and is an ideal choice for simulation and analysis wireless networks. The OPNET simulator has a large library of models representing the configuration and behavior of commercially available routers and terminal equipment. Each of these simulators has strengths and weaknesses which must be evaluated by the protocol designer before selecting the correct simulation platform to study the behavior of a new or experimental network protocol.

Thus when choosing the correct simulation engine for protocol behavior analysis, the researcher is faced with difficult tradeoffs. Since no single simulator can always provide the necessary framework for complete analysis of a particular protocol, the researcher must sacrifice some level of fidelity or detail at one layer in exchange for additional detail at another layer. For example, suppose the experimental protocol to be analyzed is some improvement on HTTP which is intended to be effective in a mobile wireless environment. Clearly, the extensive library of TCP protocol implementations found in the *ns* simulator make this a good choice to insure the new HTTP performs well under a variety of transport protocols. However, the *GloMoSim* simulator provides significantly more detail, fidelity and flexibility in the MAC layer and physical layer simulation than does *ns*, and thus would be a good choice for analyzing performance of the experimental HTTP protocol in a variety of wireless environments. Even running the experiments twice, once on each of the two simulators environment.

To address related issues, we have previously introduced the *Dynamic Simulation Backplane*[12]. The backplane provides an interface between heterogeneous simulators, allowing them to exchange meaningful information without implicit knowledge of the internal event structures of each simulator. In this prior work, we focused exclusively on interconnecting simulations across link boundaries. In other words, packets were exchanged between simulators only at the lowest protocol stack layer. As packets traveled either up or down the stack, the exchange of information between layers was exclusively within a single simulator. While this method achieves our goal for some level of scalability and heterogeneity, the flexibility needed in the above example is lacking. The split protocol stack methodology proposed here provides both scalability (allowing distributed simulation of a single network on a large number of workstations), as well as the flexibility to choose partial protocol stack implementations from the various simulators.

A number of researchers are exploring methods and techniques for scalable network simulations using parallel and distributed simulation. Riley et al.[14, 13] have designed and implemented the *Parallel/Distributed ns* (*pdns*) to provide scaling and improved performance for the *ns* simulator. Perumalla et al. [10, 9] created the *Telecommunications Description Language* (*TED*), which allows multithreaded network simulations on an SMP processor. Nicol et al. [8] propose the *Infrastructure for Distributed Enterprise Simulations* (*IDES*), a Java based simulation engine designed specifically for parallel and distributed simulations (although not necessarily network simulations). Cowie et al.[3, 4] describe the *Scaleable Simulation Framework* (*SSF*) as a method for parallel simulation of large scale networks. Bagrodia et al. implemented the *GloMoSim*[15] simulator previously mentioned. *GloMoSim* is built on top of the *PARSEC*[2] parallel simulation engine, and is designed to improve performance and scalability when run on a shared-memory symmetric multiprocessor. Additionally, Bagrodia[1] has implemented a version of *GloMoSim* that includes portions of the *ns* TCP

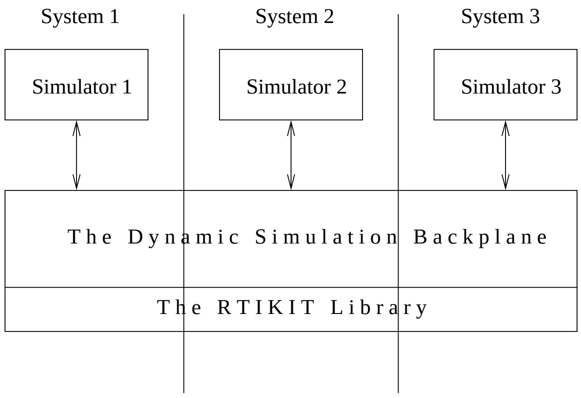

Figure 1. Dynamic Simulation Backplane architecture

protocol. This effort uses the glue approach with sections of source code copied from *ns*, resulting in some difficult software maintenance issues.

The remainder of this paper is organized as follows. In section 2 we give an overview of the dynamic simulation backplane. In section 3 we discuss the split protocol stack method for distributed network simulation. In section 4 we discuss our implementation of the split protocol stack backplane, and give some performance results. In section 5 we give some conclusions and future directions of our research.

2. The Dynamic Simulation Backplane

The *Dynamic Simulation Backplane* is described in detail in [12]. An overview of the operation of the backplane is given here to assist in understanding of the split protocol stack model. Figure 1 shows the overall architecture of a distributed simulation using the Dynamic Simulation Backplane. The figure shows a distributed simulation running on three systems. Each simulator sends and receives event messages from the backplane in *native* format, using the internal representation for events that are specific to that simulator's implementation. The backplane converts the event messages to a common, dynamic format and forwards the events to other simulators. The format of the dynamic messages is determined at runtime, on a message–by–message basis. Details of this dynamic conversion process are given later in this section.

The backplane uses the services provided by a *Runtime-Infrastructure* library, known as

RTIKIT[6]. The RTIKIT assists the backplane by providing the message distribution and simulation time management services required by all distributed simulations. The backplane itself provides services specific to the support for heterogeneous simulations. These services fall into three basic categories: Registration Services, Message Exporting Services and Message Importing Services.

2.1. Registration Services

To make use of the dynamic simulation backplane for message exchange between simulators, each simulator first uses a registration process, where they describe the information that is defined by event messages within that simulator. Clearly, for heterogeneous simulators to exchange meaningful information, there must be some common baseline describing the information to be exchanged. Fortunately, within the networking community, there are well known and widely adopted standards for exchanging data packets between end systems. The *Request For Comments* (RFC's) published by the Internet Engineering Task Force (IETF) define clearly a number of protocols and required data items to be exchanged by those protocols. During the registration process, each simulator specifies which protocols are known, and which data items with the protocols have meaning. Experimental protocols or new experimental data items within an existing protocol can also be specified, thus insuring the backplane is not limited to only known protocols.

After all participating simulators have completed the registration process, a global consensus protocol is performed which results in a complete picture of all protocols and data items that are registered by any simulator. Each protocol and data item is assigned a unique *item identifier*, which is made known to all participating simulators. This item identifier is later used in the creation of the dynamic format messages exchanged between simulators.

2.2. Message Exporting Services

At some point during the execution of a heterogeneous simulation, a given event message must be forwarded from one simulator to another, with no guarantee that the two simulators have a common representation of the event format. The event message transfer might be from a given protocol stack layer on one simulator to the same protocol stack layer on the second (for example from the *IP* layer in *ns* to the *IP* layer in *GloMoSim*). This method is described in [12]. Alternately, the transfer could be between different layers of the same protocols stack (for example from the TCP layer in *ns* to the *IP* layer in *GloMoSim*).

To accomplish this heterogeneous message exchange, the backplane uses a message *Exporting* and *Importing* paradigm. A simulator sending an event to a foreign simulator calls the *ExportMessage* service, which creates a common, dynamic format message. The simulator then forwards the dynamic format message to the foreign simulator. The message format is *dynamic* in that only the data items that are meaningful for a given event message are included in the dynamic message. By querying the simulator with callback functions, the backplane can discover which items are meaningful for each message to be exported.

2.3. Message Importing Services

Once an event message has been exported to the dynamic message format as described above, the message is transfered to another simulator and is ready to be imported by the receiving simulator. The importing process is the inverse of the exporting, and causes a message to be converted from the common dynamic format to the internal event message format of the receiving simulator. This is again accomplished by the backplane using callback functions to inform the simulator of the value of each data item received in the dynamic message.

3. Splitting the Protocol Stack

There are two natural places for heterogeneous simulators to exchange event messages. First is between the bottom layers of the protocol stack. Simulator 1 would build up an event message going down the layers of its internal stack, and would export the message after the lowest layer has processed it. When receiving a message from simulator 1, simulator 2 would import it and then process the message going up the protocol stack starting at the lowest layer. This method of dynamic message exchanging (known as the "across protocol stack" method) is the subject of the research described in [12].

An alternative method, and one providing poten-

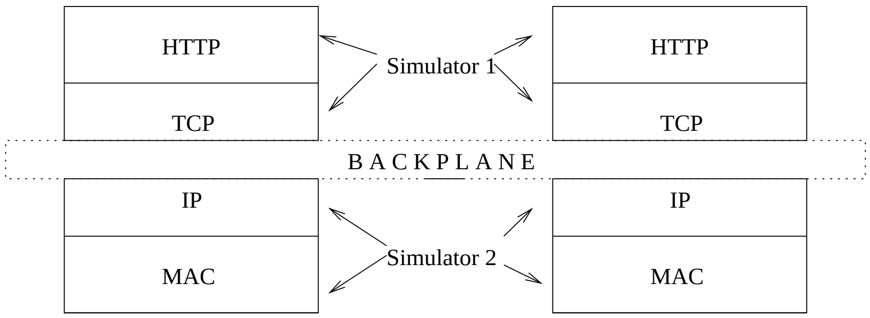

Figure 2. Split protocol stack method

tially more flexibility, is the "split protocol stack" method. In this method, heterogeneous simulators exchange event messages across layers of a single protocol stack. An example of this method is shown in figure 2. Here, simulator 1 processes event messages for the HTTP and TCP layers of the protocol stack, and then passes those partially processed messages to simulator 2 for the lower layers of the stack. When receiving messages, simulator 2 processes the lower layers (MAC and IP), and then passes the message (using the backplane) to simulator 1 for further processing.

This method provides the flexibility to mix and match simulation functionality in a way that more closely suits the needs of the simulationist. Of course, the two methods described above can be combined, using the split protocol stack model in two or more simulators; connected using the across protocol stack method between other simulatorss. However, this method introduces a severe limitation on the overall performance of the distributed simulation, namely the presence of a zero–lookahead message exchange.

Lookahead. In a conservatively synchronized, distributed discrete event simulation, one of the primary factors affecting the performance of the simulation is the presence (or absence) of *lookahead* between the individual simulators. The lookahead between a pair of simulators is defined as a lower bound on the amount of simulation time that advances as messages are exchanged between the simulators. In a typical distributed network simulation using the across protocol stack method, there is naturally some non–zero (and potentially quite large) lookahead between any two simulators. Since messages are exchanged between simulators as packets are transmitted on some communication medium, the transmission time and propogation delay create a naturally non–zero lookahead value. Unfortunately, there is no corresponding natural delay as messages are exchanged between layers of a single protocol stack. Exchanging messages between simulators modelling different layers of the same protocol stack results in a zero–lookahead exchange, with resulting poor performance.

Our solution to the zero–lookahead problem is to nominate one of the two simulators as the master, which will represent both simulators in the overall distributed simulation environment. We chose the simulator modelling the lower layers of the protocol stack, but this choice is somewhat arbitrary. We implemented a simple shared–memory interface between the master and slave simulators to allow a quick and efficient exchange of information between the two. The master will participate in all of the time management computations of the distributed simulation, and represent both simulators in this computation. The remainder of this section discusses the shared–memory interface and algorithms for time management in this environment. In all of this discussion, the *master* is the simulator modelling the lower layers of the protocol stack, and the *slave* is the simulator modelling the upper layers. The processing model for this split protocol environment is that, assuming the zero–lookahead message passing between the master and the slave, there can be no parallel event processing between the two. Either the master can process an event, or the slave can; but neither can process events simulataneously with the other (ignoring the issues of simultaneous timestamp events). Since we are stuck with serial event processing between the master and the slave, our approach is to minimize the waiting time between the two. Additionally, we propose running the two processes on a dual CPU system, such that one process can be processing events while the other is spin–waiting on permission to process events.

The shared–memory interface consists of:

- Two uni–directional circular message passing queues, one for passing messages from the slave to the master (*S2M*), and a second for passing messages from the master to the slave *M2S*). Uni–directional circular queues are ideal for message passing in this environment since they require no interlocking of shared variables or critical section processing.
- *NERCcount* An integer counter specifying the number of times the slave has requested permission to advance simulation time to a new value.
- *TAGCount* An integer counter specifying the number of times the master has granted the slave permission to advance simulation time to a new value.
- *NERTime* A floating point value specifying the simulation time advance requested by the slave.
- *TAGTime* A floating point value specifying the simulation time advance granted by the master.
- *SmallestM2S* A floating point value specifying the smallest timestamped event sent by the master to the slave since the last time advance grant to the slave. This is initialized to a value larger than any possible event in the system.

With the above shared variables, our model is that the slave has permission to process events if *NERCount* equals *TAGCount*, and the master has permission if it does not. We describe the processing of events at the slave first since it is the simpler of the two, followed by the processing at the master.

Slave Processing When the slave has permission to process events (*NERCount* equals *TAGCount*), it simply advances it local simulation time to *TAGTime*, and processes any event with a timestamp less than or equal to the *TAGTime* value. In actuality, with this model there in no possibility that an event with a timestamp less than *TAGTime* exists, since if there were it would have been processed on a previous iteration. All events with timestamp equal to *TAGTime* are processed (which may result in new events with timestamp equal to *TAGTime* being exported and passed to the master via the *M2S* queue). When all such events have been processed, the slave stores the timestamp of the earliest unprocessed event in *NERTime*, and advances *NERCount* by one. At this point, the slave has asked permission to advance time to *NERTime*, and permission to process events has been passed to the master. The slave will spin, waiting for *NERCount* to be equal to *TAGTime*, indicating permission has been given back to the slave to repeat the process. While spinning, the slave will monitor the *M2S* queue, removing messages (and or course importing to internal format using the backplane importing services), and placing them in the queue of unprocessed events in timestamp order. The processing of the event importing while spinning gives some amount of parallelism between the master and slave processes.

Master Processing The master spins waiting for *NERCount* not equal to *TAGCount*, indicating the slave has finished processing for this cycle. The master must participate in a global time management algorithm, such as that discussed in [11] to determine a lower bound on the timestamp of all unprocessed messages (plus lookahead) in the entire system (not including the slave processes). This value is called the *lower bound on timestamp* (*LBTS*). To determine an *LBTS* value, all simulators report the timestamp of their smallest unprocessed event to a global consensus protocol, which computes the global minimum. The value reported by the master to the consensus protocol is determined as follows.

1. Insure the *S2M* queue is empty. if it is not, remove all pending messages from the slave and place them in the queue of unprocessed events (in timestamp order). There is no possibility of a race condition here since at this point the slave is no longer has permission to process events, and is simply spinning waiting for permission. The *S2M* queue should normally be empty at this point, since the master is monitoring the queue while it is waiting for permission to process events.
2. Report the minimum of the master's own smallest unprocessed event, the *NERTime* requested by the slave, and *SmallestM2S* which represents the smallest timestamp sent by the master to the slave in the master's most recent processing cycle.

Once the *LBTS* value is known, the master can process all pending events with timestamp less than or equal to the minimum of the *LBTS* value, *NERTime*, and *SmallestM2S*. In other words, the *LBTS* value sets an upper bound on the simulation time advancement of the master/slave pair, but the master/slave pair must process events serially between them. Processing of these events by the master may cause event messages to be exported and passed to the slave using the *M2S* queue. Each time an event is passed to the slave, the *SmallestM2S* value is set to the minimum of the current *SmallestM2S* value and the timestamp of the message being processed. When the master has processed all eligible events, the *TAGTime* value is set to the minimum of the *NERTime*, *SmallestM2S*, and the *LBTS* value. The *TAGCount* value is then advanced by one, returning permission to the slave.

The net effect of this shared memory approach and the alternating permission protocol is that the local event queues of the master and slave processes appear to the federation as a single event queue. At any point in time, only the smallest event of the two event queues can safely be processed, which mimics the behavior that would be obtained if the two queues were merged to a single queue.

4. The *ns* to *GloMoSim* Split Protocol Stack Simulation

We experimented with the split protocol stack simulation with *GloMoSim* and ns. The protocol stack is split between TCP and IP layers, with *ns* simulating the upper portion of the protocol stack and *GloMoSim* simulating the lower portion. Each *GloMoSim/ns* pair simulates a wireless network that contains a number of wireless nodes. These wireless networks are connected to each other through a backbone network, which is simulated by a number of *pdns* simulators. Figure 3 shows a simulation configuration that consists of four *GloMoSim/ns* wireless networks and four *pdns* backbone networks. Each *GloMoSim/ns* pair connects to exactly one *pdns*, and the *pdns*'s are fully connected to each other. There is FTP traffic between wireless nodes in a wireless network, and also FTP traffic between wireless networks that goes through the *pdns* backbone.

We ran the simulation on a multi-processor shared-memory system, and each *GloMoSim*, *ns* and *pdns* process was running on a separate processor. One processor was assigned to each *pdns* backbone network, and a pair of processors was assigned to each *GloMoSim/ns* pair. The number of processors assigned was increased linearly as the number of wireless networks being modeled was increased.

In the experiments we varied two parameters to measure the time to complete the simulation. The two parameters are, 1) number of wireless networks (i.e. number of *GloMoSim/ns* pairs, which equals to the number of *pdns* simulators in between, since each *GloMoSim/ns* pair connects to exactly one *pdns*), and 2) the percentage of local traffic in the total FTP workload. Note that the total traffic grows linearly with the number of wireless networks modeled. For example, if the total traffic of 1 wireless networks is 1MB, then the total traffic of 8 wireless networks is 8MB, including both the local traffic in the same wireless network and the traffic between wireless networks that goes through the backbone. By growing the traffic linearly with the number of wireless networks being simulated, and by expanding the number of processors in the federation at the same time, a "perfect" speedup ratio would be indicated by identical running times for each of the simulations.

For comparison, we also ran the simulations with monolithic *GloMoSim* with the same configurations. Figure 4 shows the performance when the number of wireless nodes in each wireless network is fixed at 200. The left chart is the performance of monolithic *GloMoSim* simulation, and the right chart is the performance of *GloMoSim/ns* split protocol stack simulation.

The baseline case is one wireles network where 100 percent of the traffic is local traffic. In the right figure, we can see that as the number of wireless networks increases, the time it takes to complete the simulation does increase, but the increase is reasonably small. Generally speaking, larger local traffic percentages lead to better speedup. This is expected, since a large amount of local traffic increases the number of local events at a given simulator that can be processed in a single lookahead window. At the other extreme, even when only 10 percent of the traffic is local traffic, running eight wireless networks plus eight *pdns* backbones still only takes about 2 times as the time to run two wireless networks plus two *pdns* backbones.

This shows that the parallel simulation speeds up the simulation significantly: it takes 2-3 times more time to simulate 8 times larger network and 8 times more traffic. Also, splitting the protocol stack(the right chart) results in 1-2 times slow-down than running monolithic simulator(the left chart), which seems reasonable.

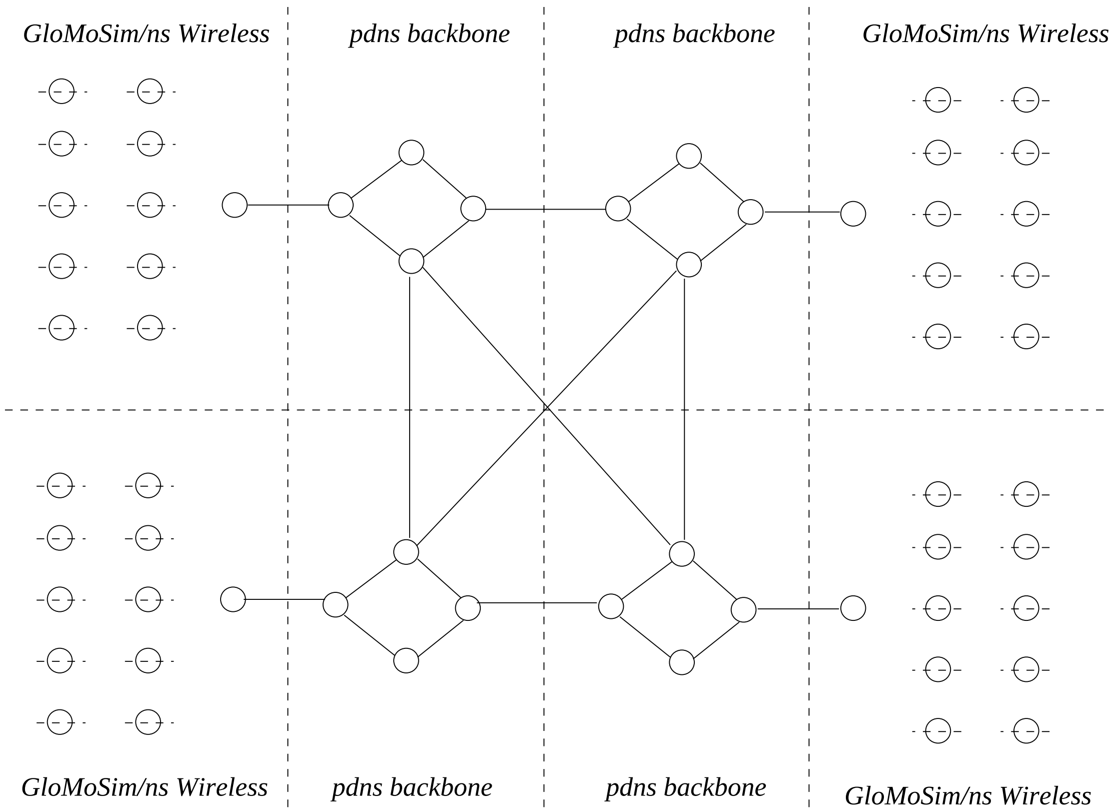

Figure 3. Simulation configuration with four $GloMoSim/ns$ pairs and 4 $pdns$'s

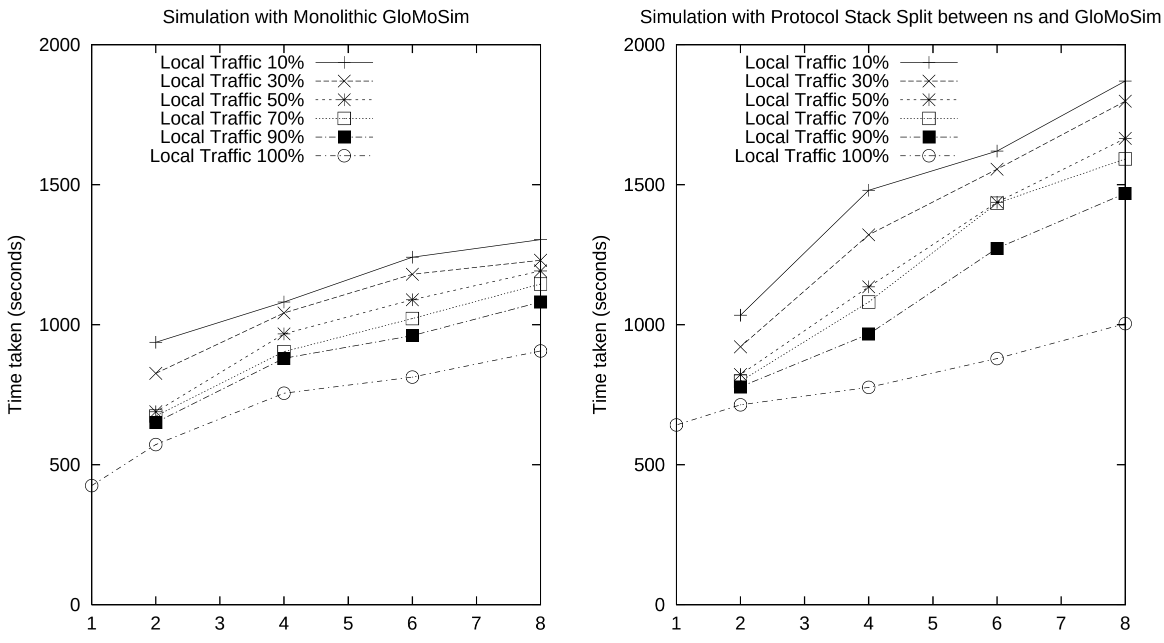

Figure 4. Simulation running time with 200 wireless nodes in each wireless network

5. Conclusions and Future Work

Each different network simulator has different strengths and weakness. To combine the strengths of different simulators, we previously introduced *Dynamic Simulation Backplane*[12] to make different simualtors work together. In this paper we discussed the split protocol stack methodology for network simulation that allows networking researchers to run different simulators together, and take advantage of the strong implementation at different protocol layers of different simulators. Using this methodology, we are able to construct complex parallel heterogeneous network simulation scenarios where *GloMoSim* simulates the MAC layer of wireless networks, *ns* simulates the TCP layer of the same wireless networks and *pdns* simulates the wired backbone between wireless networks. Our experiments show that the parallel running of the *GloMoSim* and *pdns* simulators speeds up the simulation as the simulated network scales up.

For future work, we believe there is potential for more parallelism between the *ns* and *GloMoSim* pairs, since each *ns/GloMoSim* pair typically simulates more than one protocol stack. In addition, in many network simulation scenarios, a simulator knows apriori when packets of a given flow are going to another simulator, and we may be able to exploit this knowledge to achieve more parallelism in the simulation. In our future research we will try to exploit these potential methods for more parallelism, and further improve the performance of complex parallel network simulation scenarios.

References

[1] R. Bagrodia. Private communication, 1999.

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

[3] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards realistic million-node internet simulations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, June 1999.

[4] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, January 1999.

[5] D. Estrin et al. Virtual internetwork testbed: Status and research agenda, July 1998. USC Computer Science Dept, Technical Report 98-678.

[6] Richard Fujimoto. RTI-KIT v0.2 specification, March 1998.

[7] S. McCanne and S. Floyd. The LBNL network simulator.
Software on-line: http://www.isi.edu/nsnam, 1997. Lawrence Berkeley Laboratory.

[8] D. Nicol, M. Johnson, A. Yoshimura, and M. Goldsby. Ides: A java-based distributed simulation engine. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, July 1998.

[9] K. Perumalla, R. Fujimoto, and A. Ogielski. Ted - a language for modeling telecommunications networks. *Performance Evaluation Review*, 25(4), March 1998.

[10] K. S. Perumalla and R. M. Fujimoto. Efficient large-scale process-oriented parallel simulations. In *Proceedings of the Winter Simulation Conference*, December 1998.

[11] K. S. Perumalla and R. M. Fujimoto. Virtual time synchronization over unreliable network transport. In *15th Workshop on Parallel and Distributed Simulation*, May 2001.

[12] G. F. Riley, M. H. Ammar, R. M. Fujimoto, D. Xu, and K. Perumalla. Distributed network simulations using the dynamic simulation backplane. In *Proceedings of the 21st Annual Conference on Distributed Computing Systems*, April 2001.

[13] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of of Computer and Telecommunication Systems*, October 1999.

[14] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. Parallel/Distributed ns. Software on-line: www.cc.gatech.edu/computing/compass/pdns/index.html, 2000. Georgia Institute of Technology.

[15] X Zeng, R. Bagrodia, and M. Gerla. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simlations*, May 1998.