A Federated Approach to Distributed Network Simulation

GEORGE F. RILEY
College of Engineering
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250
and
MOSTAFA H. AMMAR and RICHARD M. FUJIMOTO and ALFRED PARK and KALYAN PERUMALLA and DONGHUA XU
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332-0280

We describe an approach and our experiences in applying federated simulation techniques to create large–scale parallel simulations of computer networks. Using the federated approach, the topology and the protocol stack of the simulated network is partitioned into a number of sub–models, and a simulation process is instantiated for each one. Runtime infrastructure software provides services for interprocess communication and synchronization (time management). We first describe issues that arise in *homogeneous* federations where a sequential simulator is federated with itself to realize a parallel implementation. We then describe additional issues that must be addressed in *heterogeneous* federations composed of different network simulation packages, and describe a *dynamic simulation backplane* mechanism that facilitates interoperability among *different* network simulators. Specifically, the *dynamic simulation backplane* provides a means of addressing key issues that arise in federating different network simulators: differing packet representations, incomplete implementations of network protocol models, and differing levels of detail among the simulation processes. We discuss two different methods for using the backplane for interactions between heterogeneous simulators: the *cross–protocol stack* method and the *split–protocol stack* method. Finally, results from an experimental study are presented for both the homogeneous and heterogeneous cases that provide evidence of the scalability of our federated approach on two moderately sized computing clusters. Two different homogeneous implementations are described: *Parallel/Distributed ns* (*pdns*) and the *Georgia Tech Network Simulator*(*GTNetS*). Results of a heterogeneous implementation federating *ns* with *GloMoSim* are described. This research demonstrates that federated simulations are a viable approach to realizing efficient parallel network simulation tools.

Categories and Subject Descriptors: I.6.8 [**Simulation**]: Parallel Simulation of Computer Networks

General Terms: Experimentation

Additional Key Words and Phrases: Simulation, Networks, Distributed Simulation

---

This work is supported in part by NSF under contract number ANI-9977544, ANI-0136969 and DARPA under contract number N66002-00-1-8934.
Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 20YY ACM 0000-0000/20YY/0000-0001 $5.00

1. INTRODUCTION

Simulation is, and will remain, the method of choice for many computer network analysis problems. Although mathematical modeling and analysis is useful in many situations, the complexity of modern networks combined with the inability to apply simplifying assumptions in many situations (e.g. it is well-known that Markovian traffic assumptions are often inappropriate and can lead to misleading results) limit the applicability of purely mathematical methods. Even when such mathematical methods can be used, simulation is often used to validate the models. An extensive list of networking research that has used simulation methods is presented in [Bajaj et al. 1998]

The modeling and performance analysis of computer networks is particularly well suited for *discrete event simulation* techniques. The flow of data packets through the network can be modeled as a sequence of events occurring at discrete points in time. A packet leaves one end of a communications link at time T, and will later arrive at the other end of that link at time $T + \Delta t$. Similarly, when a packet arrives at a queue connected to an output link, the arrival event can be modeled by a discrete action at time Tq, and the departure event can be modeled at another time $Tq + \Delta tq$. Finally, protocol endpoints can respond to the receipt of a data packet and produce an appropriate response packet to be forwarded to the other endpoint. Each of these actions can easily be modeled.

Discrete event simulators[Fishman 1978; Pooch 1993; Banks 1996] simulate the behavior of the real world system being investigated by maintaining a model of the progression of time through the system. This representation of real world time is called *simulation time*. The simulator then models the overall behavior of the system by keeping an ordered set of timestamped events waiting to be processed. The timestamp of an event is the simulation time at which this event will occur. The set of future events is kept in order of increasing timestamp. At the simplest level, a discrete event simulator can simply examine the earliest unprocessed future event, advance the simulation time to the timestamp value of that event, and process the event. Processing an event may result in changing the state of one or more objects, creation of one or more new future events, or the cancellation of pending future events. The simulation is complete when there are no remaining unprocessed events or when a certain predetermined simulation time has been reached.

The construction of a discrete event simulator to model a computer network is relatively well–understood. One needs a method of describing the topology of the network to be modeled, a method for describing the behavior of the network elements, a simulation engine that manages and processes the pending events queue, and a method for observing the behavior of those elements. However, when constructing a simulator in this fashion one quickly runs into difficulties.

(1) The simulation of any reasonable size network (of more than a few hundred high speed network elements and a moderate traffic load) can take excessive amounts of CPU time to execute. Consider the simulation of a single link modeling an OC48 circuit (2.4 giga-bits per second). Assuming an average packet size of 1000 bytes, such a circuit would generate up to 600,000 simulation events for every second of simulation time. If we were modeling one thousand such links, such a simulation would require more than an hour to simulate one second of the operation of the network on contemporary

simulators such a *ns* executing on a modern workstation. It would take several days to simulate just one minute of operation of this network.

(2) The memory utilized by a network simulation will grow at least linearly with the size of the network. For some tools, it can grow quadratically with the number of network elements being modeled, due to the requirement for routing tables in each routing node of the simulated network. Even on a simulation system with large amounts of physical memory (up to 2GBytes for example), the maximum size of the network that can be represented is on the order of a few thousand nodes for a contemporary simulator such as *ns*.. For many problems, such as modeling the behavior of core Internet routers processing large numbers of simultaneous flows, much larger models are needed.

Distributing the execution of the simulation over multiple processors offers one solution to alleviating these problems. The amount of memory that can be utilized increases linearly with the number of processors, enabling larger models to be represented. The computing speed of the simulator, e.g., as measured by the number of simulator events that can be processed per second of wallclock time, can in principle also increase linearly, allowing more network flows to be simulated in a given amount of time.

The traditional approach to realizing a parallel network simulator is to create a new system "from scratch." This approach is exemplified by tools such as TeD[Perumalla et al. 1998], SSFNet[Cowie et al. 2002], GloMoSim[Zeng et al. 1998], TaskKit[Xiao et al. 1999], and ROSSNet[Carothers et al. 2000], among others. While this approach offers the advantage that the software can be tailored for parallel execution, a major drawback with this approach is the time required to create the tool. In addition to developing the simulation engine, network models, and other associated tools, a substantial verification and validation effort is required. Creation of a new parallel network simulator in this way may require several man-years of effort. Further, from a user's perspective, once one has committed to using a specific simulation tool, one is limited to whatever models and features are provided or can be easily added to that tool.

This paper is concerned with an alternate approach to parallel network simulation that focuses on reusing existing simulation models and software. A network model is decomposed into a set of sub-models, and existing simulation tools are used to instantiate each submodel. The models are then integrated, or federated, using techniques described later in this paper. This federated approach offers a means of leveraging the substantial investment that has already been, and continues to be made in developing sequential simulation tools. Further, by integrating models from *different* network simulators, one is offered the possibility of exploiting the models provided by a different tool if they are not readily available in the tool one has already committed to using. A key challenge in utilizing the federated simulation approach is to realize a high performance implementation. A principal result of the research described here is to provide evidence that federated simulations can indeed provide a viable means to realizing efficient, scalable parallel network simulators.

The concept of federated simulations in not new. The SIMNET project [Miller and Thorpe 1995] and subsequent efforts such as the Distributed Interactive Simulation (DIS) protocols [Hofer and Loper 1995], Aggregate Level Simulation Protocol (ALSP) [Wilson and Weatherly 1994], and the High Level Architecture [Kuhl et al. 1999] have successfully applied this concept to military simulation applications. Early work in the parallel simulation community used this technique to create parallel queuing network simulations[Nicol and Heidelberger 1996]. The work described here differs from these efforts in its empha-

sis on network simulation, which differs from these other applications in several respects. First, decomposition of the model may not fall along physical entity (e.g., tank, ship, or queue) boundaries, since one may wish to federate simulators modeling the same hardware elements, but different levels of the protocol stack; for example, one simulator might model the physical layer, while another models network and transport layers. Network standards may be exploited to define interfaces among the federates. Second, issues concerning knowledge of global state information must be resolved. For example, individual network simulators may assume knowledge of the entire network topology in order to compute routes, an assumption that may no longer satisfied in a federated execution.

The remainder of this paper is organized as follows. Section 2 discusses the basic services needed by the federated simulation environment, including time management and message exchange. Section 3 addresses the case of *homogeneous* federated simulations where a sequential simulator is federated with itself to create a distributed implementation. This section describes issues that must be addressed in the definition of the submodels and describes some solutions. Section 4 describes the more general problem of *heterogeneous* federations composed of different network simulators. The *dynamic simulation backplane* approach for heterogeneous network simulations is described to address these issues. Section 5 describes some experimental results evaluating the performance of our federated simulators on a moderate sized computing cluster. Finally, Section 6summarizes the results of this work and suggests future directions of research.

2. DISTRIBUTED SIMULATION SERVICES

Any distributed simulation needs a basic set of services to manage the overall advancement of simulation time and to exchange information among the federates. Here, we adopt the terminology utilized by the High Level Architecture. The distributed simulation consists of a collection of autonomous simulators, or federates, that are interconnected using runtime infrastructure (RTI) software. The RTI implements relevant services required by the federated simulation environment. The most important services for the purposes of this discussion fall into two basic categories, *Time Management* and *Data Distribution*. The time management services insure that the simulation time in each of the simulator instances stays synchronized with the others, and the data distribution services allow for the transitioning of event messages from one simulator to another. Each of these is described in more detail below. Our implementations use the services provided by the Georgia Tech *Federated Simulations Development Kit*[Fujimoto et al. 2001].

2.1 Time Management

There are two well known ways to deal with the requirement for time synchronization in parallel and distributed simulation.

Using *optimistic* synchronizationsimulators will process events even with no guarantee that they will be processed in timestamp order. If an event is processed out of order (an event with a smaller timestamp is later received from another simulator), the simulator must provide some way to *undo* the effect of the out–of–order event, and return the state of the simulator to that which existed prior to processing the event. This is called a *rollback*, and is the subject of a substantial body of research.

A second approach is called *conservative* synchronization. In this method, the simulators must have some method to determine when events are *safe* to process. An event is safe when it can be determined that no event will be later received with an earlier timestamp.

In this section we discuss the issues concerning conservative methods, including time synchronization algorithms and protocols. A principal advantage of the conservative method is that once an event has been processed, there is no possibility that it will later be found to have been processed in an incorrect order. A disadvantage of this method is the necessity for an efficient method to determine when events are safe. Each simulator must, before processing any event, determine a lower bound on the timestamp of any message that will be received in the future. This lower bound is referred to as the lower bound on timestamp (*LBTS*). Once the *LBTS* value is calculated, each simulator can safely process any pending event with a timestamp less than or equal to the *LBTS* value. Conservative synchronization is the preferred approach for federated simulation because there is not need to add rollback to the simulators participating in the simulation. Conservative synchronization is used exclusively in the work described here.

2.1.1 *Conservative Time Management Methods.* One of the first methods for determining the *LBTS* value in a PDES environment was developed independently by Chandy and Misra[Chandy and Misra 1979] and Bryant[Bryant 1977]. This method, known at the *Null Message* protocol, allows each simulator to make a local decision regarding the *LBTS* and thus each simulator can independently and asynchronously determine the range of safe events. However, this method is prone to generating a large number of null messages, and can be inefficient.

An alternative approach is to utilize a global synchronization to compute *LBTS* values. Mattern [Mattern 1993] proposes a method to compute *LBTS* values using a distributed snapshot computation to determine the minimum timestamp along a consistent cut of the computation. With Mattern's method, each simulator maintains a counter of the number of messages sent, minus the number of messages received since the last *LBTS* computation. The processes periodically enter a barrier, where they report their local minimum timestamp, as well as the message counter. When all simulators have arrived at the barrier, if the sum of all the counters is zero, then all messages sent prior to the barrier have been received and accounted for in the global minimum computation. If the counters do not sum to zero, this is an indication of the existence of transient messages. If this occurs, one can simply enter the barrier repeatedly until the counters sum to zero. This method is simple, and requires little additional state in each simulator. A drawback is the potential for each process to enter the barrier repeatedly.

Fujimoto [Fujimoto et al. 2000] uses a butterfly barrier [Brooks 1986] combined with the running sum message counters proposed by Mattern. This method does not require all LPs to enter the barrier repeatedly. Instead, LPs that receive a transient message start an *update message* through the butterfly, indicating a new message count. This method performs well when there are not excessive numbers of transient messages. This method is used in the experiments described later.

2.1.2 *The Time Management API.* The Federated Simulations Development Kit (FDK) software used here implements the time management services defined in the HLA. Specifically, time advancement is achieved using a simple *Request* and *Grant* methodology. In its simplest form, the API requires the simulation event processing loop to request permission from the time management services before processing any simulation event. This service, known as `NextEventRequest` or *NER*, determines the timestamp of the earliest safe event, insures that all events with earlier timestamps are delivered to the simulator,

and notifies the simulator of the safe time. Pseudocode for a sample main loop is shown below[1].

```
while (not done)
  RequestedTime = INFINITE_TIME
  if (Event List Not Empty ) then
    RequestedTime = (Timestamp of earliest event)
  end if
  GrantedTime = NextEventRequest(RequestedTime)
  if (Event List Not Empty ) then
    if (Timestamp of earliest event <= Granted Time) then
       // Ok to process
       (Remove event from Event List)
       (Process Event)
    end if
  end if
end loop;
```

The pseudocode demonstrates the simplicity of the `NextEventRequest` service. The simulation main loop simply calls the `NextEventRequest` function before processing any event. The *NER* function determines the timestamp of the smallest *safe* event and notifies the simulator with a *Granted* time. If the earliest pending event has a timestamp less than or equal to the granted time, the event can safely be processed. In addition to determining the grant time, the *NER* function also insures that any pending events from other simulators with timestamps less than or equal to the granted time are delivered to this simulator. This pseudocode also illustrates the fact that, in a federated simulation environment, an empty event list does not indicate the simulation has completed. A simulator with an empty event list may in fact receive more events from other simulators, even though it presently has no pending events.

2.2 Data Distribution

Data distribution is concerned with distributing simulation events among producers and consumers during the execution of the federated simulation. Federated simulations differ from classical parallel simulators in that producers of simulator events cannot know a priori the consumers of their events. Simple mechanisms such as specifying the destination process for an event that are commonly used in classical parallel simulators cannot be used, because the destination may reside in a different simulator (federate). This problem is often solved by defining publication/subscription communication services; the HLA uses such an approach, for example. Subscribers specify interest expressions to indicate object or interaction class data they wish to consume, while publishers use other services to declare the type of information they can produce. The RTI software then ensures messages are routed to the appropriate destinations.

Wired network simulations such as those described earlier contain much structure that enables one to utilize a much simpler approach to data distribution than general publication/subscription mechanisms. Specifically communication links define which simulators

[1] The actual code is a bit more complex than what is shown here, due to the use of a callback function indicating the granted time.

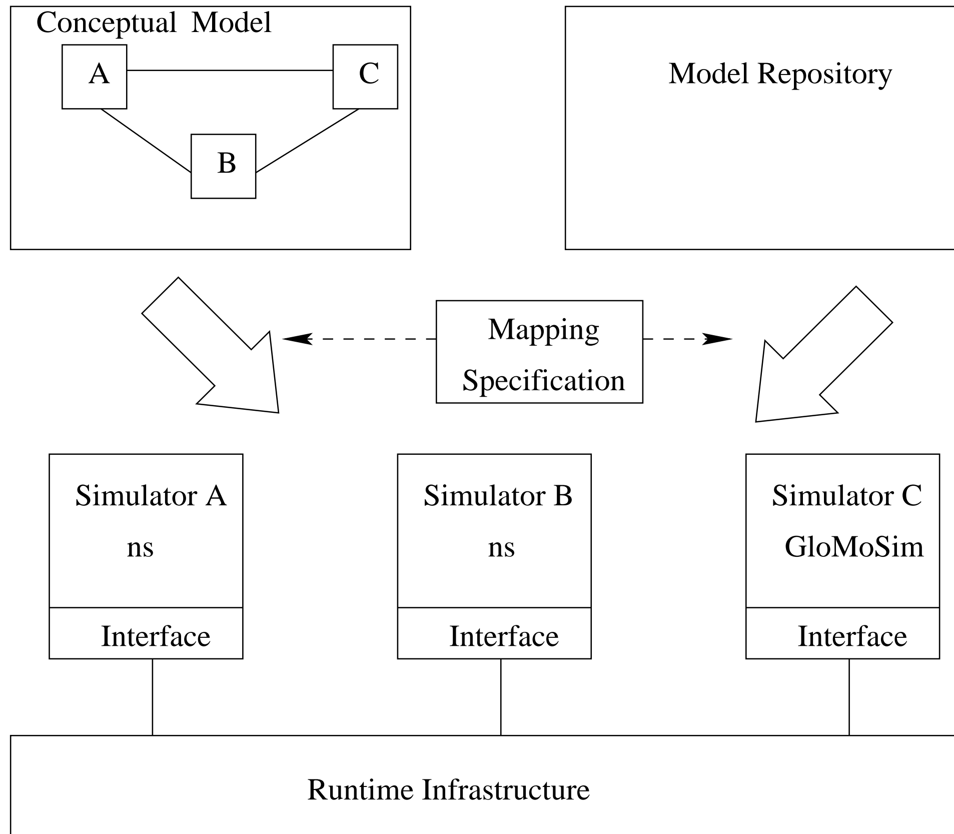

Fig. 1. Conceptual Overview

need to communicate with which others. As discussed momentarily, an abstraction called a remote link is defined to enable federates to refer to objects (in this case links) that represent connections to nodes that are simulated on another federate.

3. HOMOGENEOUS FEDERATIONS: DEFINING NETWORK SUB–MODELS

The federated approach to large–scale network simulation focuses on running the overall simulations on a clusters of workstations with no state sharing between the simulators. Each simulator will be given a network topology and data flow characteristics which describe only a portion of the network being simulated. One advantage of this approach is that a given simulator need not use local processor memory to describe network elements which are managed by other simulators, and thus larger network models can be simulated. Additionally, models that have already been developed can be composed into larger models with minimal impact on the model designer. A conceptual overview of this approach is shown in figure 1.

Here, we assume the network being simulated is partitioned along physical component and/or along protocol layer boundaries. This means the model for a node (e.g., an end host or router), or a protocol layer for a node (e.g., the TCP implementation within an end node)

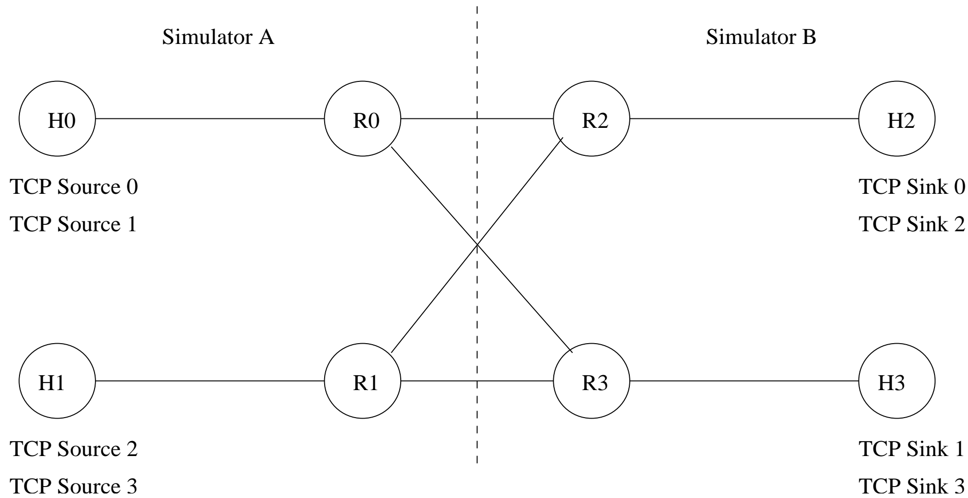

Fig. 2. Simple Network Model

lies entirely within a single simulator (or federate). Partitioning the simulator in this way enables one to exploit network standards to define the interfaces between simulators. The problem of partitioning along other boundaries is an area for future research.

In this section, we first outline issues that must be addressed when federating a sequential network simulator with itself to create a distributed network simulation package and propose some solutions. We describe our experiences with applying these solutions to the popular network simulator *ns*[McCanne and Floyd 1997][Riley et al. 1999], and similar experiences with our newly developed *GTNetS* (*Georgia Tech Network Simulator*)[Riley 2003].

Consider the simple network shown in figure 2 consisting of four end hosts (H0 – H3) and four network routers (R0 – R4), connected with physical communication links as shown by the solid lines. The simple network model also has four logical data flow connections consisting of four "TCP Source" to "TCP Sink" pairs, as shown in the figure. Further consider that the network modeler has decided to run the complete model in parallel on two systems, simulator A and B, splitting the model into sub-models as shown by the dashed line. From the point of view of the modeler, there are three basic problems which must be addressed and solved, specifically:

(1) Defining physical connectivity between sub–models
(2) Defining logical connectivity between sub–models.
(3) Defining routing between sub–models.

Each of these is addressed separately in the following subsections.

3.1 Defining Physical Connectivity

If the network in figure 2 were simulated on a single workstation using a serial simulator, the description of the overall network topology is quite straightforward. Using the semantics of the network simulator being used, the modeler would simply define the simulation elements representing each of the eight nodes, and then define the physical links by refer-

encing those nodes as the endpoints of the links. The ability to define a network topology of nodes and interconnecting links is of course a fundamental requirement of any network simulator.

But now consider splitting the model into submodels A and B (as shown in the figure) and running those in parallel, by composing two network simulators A and B. We assume that only those nodes that will be managed by simulator A will be defined on that simulator, in order to keep the memory requirements for each system as small as possible. Thus it is obvious that it becomes problematic to define a simple physical link, such as the link from $R0$ to $R2$ in figure 2. By our assumption above, simulator A will not have a node entity for node $R2$ and would not be able to define that endpoint for the physical link. A similar problem would exist in sub-model B trying to describe the physical link from R2 to R0 and the link from R2 to R1. The basic problem is that a network simulation package designed to be run in serial on a single system will assume the existence of a variable or object representing each and every network element being modeled, and assume those objects can be referenced by name. When decomposing a model in to distinct sub–models, only those network elements which are within the same sub-model can refer to each other by variable name.

A naming convention is required to specify network elements (in this case physical links) that are external to the portion of the network modeled by the local simulator. Our solution to this problem is to borrow some well known abstractions from the networking community, namely that of an *IP Address* and a *Network Mask*. The syntax of the topology specification in a network simulator can be extended to allow the specification of these values for any link endpoint. In the case where a link connects to a node declared in a different sub-model, then only the local endpoint of the link need be specified. We refer to this as a *Remote Link*, or *rlink*. At runtime, the physical connectivity of the *rlinks* can be determined by matching the network portion of the *IP Address*. Any two *rlinks* with matching network addresses are assumed at runtime to be connected, and any packet transmitted from one end of the link will be delivered to the other endpoint, using the data distribution services discussed earlier. Both *pdns* and *GTNetS* support the definition of a *remote link* with associated an *IP Address*.

3.2 Defining Logical Connectivity

In our model of figure 2, we also are modeling four logical data flow connections consisting of pairs of sources and sinks. A network simulator might allow for the declaration of an entity which generates data (such as *TCPSource0* in our example), the declaration of a receiving end of the data flow (such as *TCPSink0* in our example), and some way to specify that the two ends have logical connectivity.

Again assuming that we want to decompose the model into sub–models as in the previous section, we are faced with a similar problem. We again assume that a submodel will only define and manage data flow endpoints for those nodes that are managed by the local simulator. Simulator A will be unable to identify the remote endpoint *TCPSink0* of data source *TCPSource0* since *TCPSink0* is defined on a different simulator. Since the basic design of simulators such as *ns* requires an object representing both endpoints of a connection, we must design an alternate method to identify remote endpoints.

To solve this problem, we again borrow well known abstractions from the networking community, using an *IP Address* and a *Port number*. Our solution allows the specification of an *IP Address* for a physical link endpoint (as previously mentioned), and also allowing

binding of a logical connection endpoint to a port number unique within a node. Then a logical data connection can be specified by giving the *IP Address* and port number of the remote endpoint (which could be defined and managed on a remote simulator). The basic design of the *GTNetS* simulator inherently solves this problem, as connections are always specified using an *IP Address* and port number.

3.3 Defining Routing Information

The problem of how to determine correct routing paths between sub-models in less obvious than those of the previous sections. Again referring to the model of figure 2, consider what the simulator must do when simulating the arrival of a packet at router R0, with the ultimate destination of end host H2. In a single system simulation the correct route (the link on which to forward simulated packets for each possible destination) can be computed a priori using global knowledge of the topology. In our example, router R0 would have predetermined that any packet being routed to end host H2 must be forwarded on the link connected to R2. With global knowledge of the network topology, this is a simple and straightforward computation (although it can be time consuming for large topologies).

Once we decide to decompose the model into sub-models A and B as in the previous sections, we are faced with a problem. Without global network topology information, sub-model A does not have sufficient information to make a routing decision at routers R0 and R1. Router R0 has two remote links in our example, only one of which is the correct link to forward packets addressed to end host H2, but it has insufficient information to determine which one is correct.

We have considered four possible solutions for this problem. The first and simplest is to put the burden on the modeler to define the appropriate routes. The model topology specification can be enhanced to specify a list of remote *IP Addresses* for each remote link. Anytime a packet is to be forwarded on a simulated node with multiple remote links, the correct route can be determined from the list of specified routes. This method currently used in both *pdns* and *GTNetS*. This solution works as long as the appropriate route will not change over the course of the simulation. Such routing changes could occur due to the simulation of link failures or node failures in the model.

A second solution involves the use of *Ghost Nodes*. With this approach, the entire network topology is instantiated on every simulator, leading to the global state required to compute routing paths. However, only those nodes mapped to each simulator are instantiated with the complete state and functionality needed to model a network node. Those nodes mapped to other simulators are represented with reduced state placeholders, known as *Ghosts*, that simply indicate the existence of the node and a list of the node's neighbors. The basic design of the *GTNetS* simulator supports this approach, but it is not presently implemented.

A third solution is to start with a single model that specifies the entire network topology. With global network knowledge, the correct routes between nodes (even routes between simulators), can be determined automatically by the simulator without effort on the part of the modeler. However, this method requires the entire topology to be defined and processed on each simulator, which is inefficient with respect to memory usage. One approach is the use of a topology preprocessor, which will be given a complete picture of the simulated network and will compute the needed routing information. The preprocessor will then create the submodels (based on either a mapping given by the modeler or by some other optimization), and pass the submodels to the simulators (with the routing informa-

tion included). This solution still lacks the ability to adapt in simulation time to changing network topologies.

A fourth solution is the have the simulator run some existing and well known routing protocols while the simulation is running. Simulated network nodes will start with routing tables generated a priori (with one of the two previous approaches), and will exchange dynamic routing information in the simulation (using for example the *Border Gateway Protocol BGP*[Rekhter and Li 1995]) to adapt to any changes in network topology. This is the approach used by the *SSFNet*[Cowie et al. 2002] simulator.

4. HETEROGENEOUS FEDERATIONS: THE DYNAMIC SIMULATION BACKPLANE

In this section, we turn our attention to the problem of how to construct a larger simulation from smaller components, with the assumption that these components may be designed and written in *different* simulation packages.

It is clear that the issues raised in the previous section still apply to the heterogeneous case, however, several additional issues must also be addressed. Interoperability issues can be broadly classified into three categories:

—*Representation.* Different simulators may represent the same elements of the model in different ways. For example, consider two end nodes using the TCP protocol that are modeled in different simulation packages. One may represent addresses using a simple integer, while the other might utilize IP addresses.

—*Incomplete implementation.* Simulators often do not implement all aspects of the network protocol because certain features were not required for its intended uses. This can lead to problems when they are used in federations. For example, an end node modeled by GloMoSim expects that checksums have been computed, and discards simulated packets that do not contain properly computed checksum values. On the other hand, *ns* does not compute checksums, so *ns* generated packets will be discarded at end nodes modeled by GloMoSim.

—*Level of detail.* Different simulators may model aspects of the network at different levels of abstraction, leading to obvious problems. A simple example is one simulator may model individual packets, while another may model trains of packets in order to improve execution efficiency. Here, we focus on packet-level simulators that assume the packet is the atomic unit of data that is simulated.

We introduce the Dynamic Simulation Backplane, which addresses these and related issues. Distributed network simulations exchange information using event messages that typically model the data packets flowing between the simulated elements. Representational issues are addressed by providing a common event message-passing interface between distributed simulations. The backplane creates a format for network event messages, which is defined dynamically by the backplane using registration calls provided by the simulators. By using the backplane, a simulation engine can exchange meaningful event messages with other simulators, even when they do not share a common event message format. The backplane also supports baggage data, which occurs when a given simulator must retain protocol information of interest only to anther simulator. The address issues concerning incomplete protocol implementations, the backplane defines a common API for simulators to describe which network protocols are supported and which data elements within each

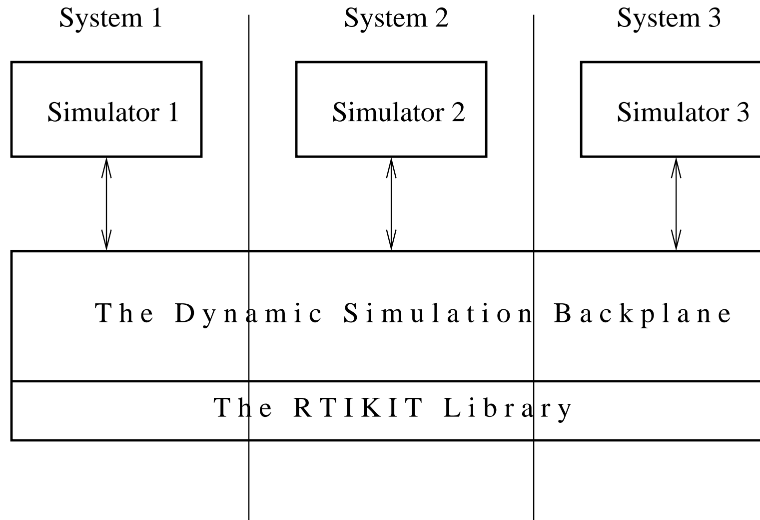

Fig. 3. Dynamic Simulation Backplane Architecture

protocol are required or available by that simulator. A consensus protocol is used to come to an agreement among the simulators concerning what protocols will be modeled, and to check for incompatibilities . The consensus protocol also partially addressed issues that arise when combining simulators modeling the network at different levels of detail.

The backplane has been released to the simulation community, and has been used to implement a heterogeneous emulation environment using *QualNet* and *pdns*[Zhou et al. 2003]. Additionally, the backplane was used to simulate a military scenario using both *GloMoSim*[Zeng et al. 1998] and *pdns*[Perumalla et al. 2002].

Figure 3 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. The backplane uses the services provided by the *Federated Simulations Development Kit* library, discussed earlier. The *FDK* 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.

The backplane services fall into three basic categories:

(1) Protocol/Item Registration Services

(2) Consensus Computation, and

(3) Message Importing/Exporting Services.

Each of these is described next.

4.1 Protocol and Item Registration Services

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. For example, RFC791[13] defines the widely used Internet Protocol (IP) and specifies a total of 14 individual data items within the protocol. These standards are used as the starting point for our registration services. Each simulator will register with the backplane the protocols that are supported, and the data items within those protocols. A unique ASCII string identifies each protocol within the backplane. An ASCII string unique within the protocol defines each data item. We emphasize however that the published standards are simply a starting point, and in no way are all-inclusive. With the backplane, simulators can register any data item for a protocol, as long as the ASCII name is unique within the protocol. Simulators can also ignore items within a published protocol if the particular item has no meaning or use within that simulator. Additionally, simulators can register completely new protocols for which there is no standard.

As protocols and data items are registered, each simulator must specify whether each is *required* or *optional*. A required protocol is one for which all simulators participating in the distributed simulation must provide support, or the distributed simulation cannot continue. An example of a required protocol might be the Internet Protocol. If IP were specified as required by any simulator, then all other simulators must also specify support for IP or the distributed simulation cannot continue. Data items within a protocol also are specified as required or optional. While all simulators might support the IP protocol, they may have differing levels of detail represented. For example, the Header Checksum data item may be modeled in one simulator, but may have no meaning in another. If the simulator supporting the header checksum field has some way to determine a reasonable default value, then that item should be specified as optional. Other items within IP might be required items, such as the Destination Address. When registering data items, the simulators also specify whether or not the data item needs *byte-swapping* or not. The backplane will later use this information to insure that all data items exchanged with peers is in a common byte ordering format. Lastly, simulators specify whether individual data items should be considered *baggage* when they are exported to simulators with no corresponding items. Baggage items are discussed in detail later.

When registering protocols, each simulator specifies the address of a callback function, called the *ExportQueryCallback*, which the backplane later uses to determine if that protocol is to be exported for a given event message. During the registration process, simulators will register all protocols that have some meaning to that simulator. However, any given event message may not in fact have information for all registered protocols. For example, a given simulator may support the HTTP protocol, but a given event message may have only TCP/IP information meaningful. By using the *ExportQueryCallback*, the simulator can inform the backplane, on a message–by–message basis, which of the registered protocols are meaningful, and thus keep the size of the dynamic event messages to a minimum for each message. The dynamic determination of the message format is described later.

When registering protocol data items, the simulator specifies the address of three callback functions, called the *ProtocolItemExport* callback, *ProtocolItemImport* callback and *ItemDefault* callback. The *ProtocolItemExport* callback is used by the backplane during

a message export action to query the simulator for the correct value of the corresponding data item. The *ProtocolItemImport* callback is used by the backplane to inform the simulator of the correct value for data items during a message import action. The *ItemDefault* callback is used by the backplane to inform the simulator that an optional data item has *not* been provided by a peer on a message import. In this case, the simulator can determine a suitable default value. For each of the three callbacks, a corresponding context pointer is specified, which is returned to the simulator when the callbacks are executed. The context can be used to provide details specific to a given item, and allow a single callback function to be used for many data items. Complete details concerning the message exporting and importing are given later.

We discuss the operation of the backplane in terms of protocols and data items within those protocols, since the target application for our research is the simulation of computer networks. As previously mentioned, a protocol in this context might be IP, and the data items associated with this protocol might be Source Address, Destination Address, etc. However, from the point of view of the backplane, a protocol simply refers to a collection of individual data items that can be referred to as an aggregate by a single name. If the target application were an air traffic control application, a protocol could be ”Aircraft Characteristics”, and the individual data items might be ”Maximum Cruising Speed”, ”Fuel Consumption Rate”, and items of that nature. For the remainder of this paper, we will continue to use the simulation of computer networks as the basis for discussion.

4.2 Consensus Computation

After all simulators have specified the protocols and data items needed, a global consensus protocol is performed to find a minimal subset of required items, and a maximal set of optional items. The purpose of the consensus protocol is twofold. First, it insures that all participating simulators support the required protocols. Secondly, each protocol and each item within the protocols is assigned a globally unique *Protocol Identifier* and *Item Identifier*, which all participating simulators are aware of. The identifiers are later used in the creation of the dynamic message format during message exporting, explained later.

To accomplish the global consensus, each simulator calls a *RegistrationComplete* function after all protocols and data items have been registered. This function acts as a barrier, which blocks until all simulators have called the function. A single system is nominated as the *Master* system. In our implementation, each simulator is assigned a unique node identifier in the range $0 \ldots (k-1)$, where k is the number of participating simulators, and the master is then chosen as the system with node identifier 0. Each system, other than the master, reports the list of the registered protocols and data items to the master. For each reported protocol, the master first determines if some other simulator has already reported the same protocol. If not, the master adds this protocol to the list of known protocols. The master also counts the number of simulators reporting a given protocol, and the number of simulators that specify it as required. The same is done for data items within a protocol.

Once all simulators have reported all protocols and data items, the master has a complete view of all reported protocols and items. The first step is to determine that all participants support the required protocols and data items. The various anomalies that can be detected by the master during the registration phase are listed below. Other agreement errors may go undetected until the simulation is actually running and messages are exchanged.

(1) All required protocols are noted as *required* by all participants, and all required data

items are noted as required by all participants. This is the ideal case, in that all simulators agree on the required protocols and data items, and all exchanged messages will contain the required information.

(2) At least one protocol is registered by at least one participant as *required*, but also registered by at least one participant as *optional*. This is less than ideal, but the simulation can still continue. Those participants registering optional protocols are not required to report that the protocol exists during a message export operation, but can accept and represent that information as it is received from their peers. The simulation may detect a failure at runtime, if an optional protocol is not included in a message exported to a peer that lists the protocol as required.

(3) An optional protocol has required data items, but the protocol is not registered by at least one participant. The simulation may continue, but an error may be detected at runtime. A required data item of an optional protocol means that if the protocol is exported, then the required item must be included. A runtime error will occur if a participant exports data items for this protocol, but does not export the required data item.

(4) A protocol is registered by at least one participant as *required*, but the same protocol is not registered by at least one other participant. In this case, the overall simulation cannot continue. A required protocol is unknown to at least one participant, and thus that participant cannot provide data items for the protocol on message exporting. The master system will inform all participants of the error and abort the simulation.

Once the master has determined the validity of the protocol and item registrations as described above, each protocol is assigned a unique protocol identifier by simply numbering them starting from 0. Each item within each protocol is also assigned an identifier, again starting with 0 in each protocol. Once the master system has assigned the identifiers, the complete set of protocols and data items is returned to all participants, along with the assigned identifiers. At this point, all participants agree on the complete set of protocols and data items, along with the unique integer identifiers assigned to each.

4.3 Message Importing/Exporting Services

Once the registration and global consensus phase of the backplane execution has completed, the simulation phase of each participant can begin. The backplane provides a mechanism for exchanging event messages between simulators. Consider the distributed simulation shown in Figure 4. This simulation defines a network model to be simulated, consisting of eight nodes and eight links as shown. The actual simulation execution is distributed on two systems, simulators A and B as shown. A data packet event message will need to be transferred from simulator A to simulator B when a simulated transmit data packet event is generated at simulator A on link 1. The backplane will export this event message, by converting it from an internal format specific to simulator A, to a common dynamic format that can be understood by all simulators. Simulator B will need to import the event message when a simulated receive data packet event is received on link 1. The message import action is the conversion of the dynamic format message received from a peer simulator to an internal representation specific to a given simulator. Details on the exporting and importing actions are given in the next sections.

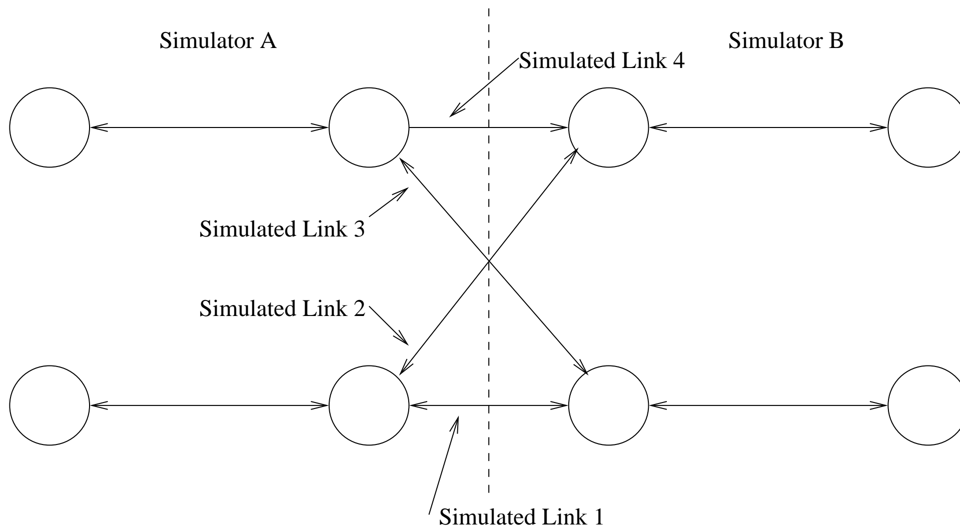

Fig. 4. Simple Distributed Simulation

4.3.1 *Exporting Messages.* When a given simulator must transmit a data packet event to a peer simulator, the *ExportMessage* function of the backplane is called. This function calls the *ProtocolExistsQuery* (*PEQ*) callback for every protocol registered by that simulator, to determine if this particular data packet event contains data items for each protocol. This technique allows a simulator to register all protocols that are known to that simulator, even if all protocols do not exist for all data packet events. If the *PEQ* callback reports that the protocol is present in the packet, the backplane notes in the dynamic format message that data items for this protocol are following. Then the *ProtocolItemExport* callback is called for every item registered for that protocol, and the reported value for each item is noted in the dynamic format message. In response to the *ProtocolItemExport* callback, a simulator can report that no value exists for a given item, allowing all possible items for each protocol to be registered, even if they are not present in all data packets. As data items are copied to the dynamic format message, they are byte-swapped as needed to a common byte-ordering representation.

The *PEQ* callback is called only for those protocols registered by the simulator calling the *ExportMessage* function. Recall that after the global consensus computation each simulator has a complete picture of all protocols and all data items registered by any participant. Clearly, if some simulator has not registered a given protocol, then that protocol cannot exist in native format data packet events for that simulator, and thus the protocol is assumed to be absent.

4.3.2 *Importing Messages.* When a simulator has received a dynamic format data packet event from a peer, the message must be converted back to an internal representation for that simulator in order to be meaningful. The simulator calls the *ImportMessage* function of the backplane to accomplish this conversion. This function scans the dynamic format message, and for each protocol included will determine if this simulator has registered the existence of the protocol. If the protocol has not been registered, and if any peer specified the baggage indicator for the protocol, then all items in the protocol become

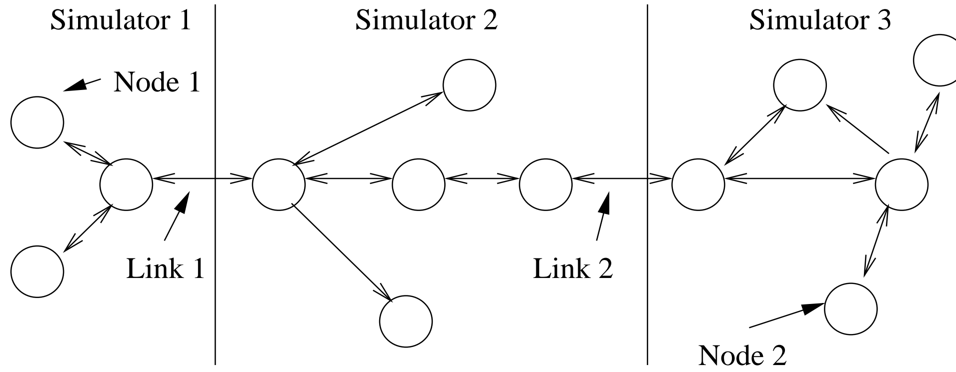

Fig. 5. Baggage Example

baggage (as described in the next section). If the protocol was registered, then the *ProtocolItemImport* or *ItemDefault* callback is called for each registered item. *ProtocolItemImport* is called for each data item included in the dynamic message, and *ItemDefault* is called for each item not included in the dynamic message. For items present in the dynamic message but not registered by the simulator, the item may become baggage.

After all of the callbacks for registered data items have been called, the simulator receiving the dynamic message will have a complete picture, in native format, of the meaningful content of the message that was exported by the peer, plus any defaulted data items.

4.3.3 *Baggage.* Baggage data items are information that must be carried along with a simulated data packet within a given simulator, but in fact have no meaning for that simulator. Consider the distributed simulation shown in Figure 5. For this example, we assume that simulators 1 and 3 have the same level of detail for the TCP protocol, but that simulator 2 has support for IP only and no notion of the TCP protocol. Now suppose that the overall simulation is to model the behavior of a TCP flow from node 1 to node 2. It is clear that when simulated packets arrive at node 2 in simulator 3, the TCP protocol information from node 1 must be included for the simulation to function properly. However, since simulator 2 does not have an internal representation of TCP protocol items, there must be some way for simulator 2 to retain this information that was provided by simulator 1. When packets flow from simulator 1 to simulator 2 (on link 1), the backplane will convert the data packets to the dynamic format, using all of the registered data items from simulator 1 (which will include both TCP and IP information). When simulator 2 receives the dynamic message, the backplane will convert the information back to an internal representation known to simulator 2. Any data item (or protocol) that is included in the dynamic message but is NOT known to simulator 2 will be retained as baggage. In this case the baggage will be all data items from the TCP protocol supplied by simulator 1. The baggage buffer will be returned to simulator 2 as an output of the *ImportMessage* function, and must be retained by simulator 2 as part of the data packet. Simulator 2 does not need to be aware of the meaning of any of the baggage, but rather must just carry the baggage along with the packet as a bag of bits.

The packet will be routed through the simulated network by simulator 2, and eventually be passed to simulator 3, via link 2. When exporting the data packet via the *ExportMessage* function, the baggage buffer is provided to the backplane, and all baggage items are

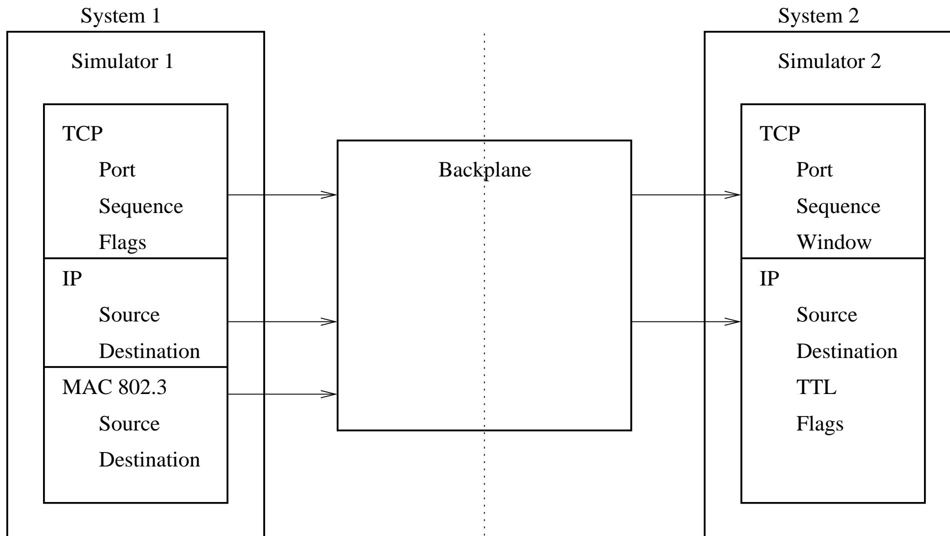

Fig. 6. Exporting/Importing Example

included in the dynamic format message sent to simulator 3. When the data packet arrives at simulator 3 (via link 2) it will contain all of the IP protocol information provided by simulator 2, plus the TCP protocol information provided by simulator 1 that was carried as baggage.

4.3.4 *Importing/Exporting Example.* Figure 6 shows a simple example of message importing and exporting. Simulator 1 has registered three protocols, TCP, IP, and MAC 802.3, each with several data items as shown. TCP and MAC have been registered as optional, and IP has been registered as required. Simulator 2 has registered TCP as optional and IP as required, with three and four data items respectively, again as shown. The IP/Destination item and the TCP/Sequence item have been registered as required by both simulators. All other items are optional. At some point in the distributed simulation, simulator 1 will create a data packet transmission event that must be received by simulator 2. Simulator 1 calls the *ExportMessage* function of the backplane, which creates a dynamic format message as follows. First, the *ProtocolExistsQuery* callback is called for the TCP protocol. Assuming that simulator 1 reports that TCP exists for this message, the *ProtocolItemExport* callbacks are called for the Port, Sequence, and Flags items, and the reported values are stored in the dynamic message. The process is repeated for the IP and MAC protocols, resulting in a total of 7 data items being represented in the dynamic message. Any value for which the byte-swapping specification was included during registration is byte swapped to a common byte ordering representation. The resulting dynamic message is then transmitted to simulator 2 by whatever system interconnect exists between the participants in the distributed simulation.

When simulator 2 receives the dynamic message, it in turn calls the *ImportMessage* function of the backplane, which converts the dynamic message to a format internal to simulator 2. It does this by using the *ProtocolItemImport* callbacks that were specified for TCP/Sequence, TCP/Port, IP/Source, and IP/Destination, and passing the values

Protocols	Items per Protocol	Overhead per Item μ seconds
1	1	0.77
1	10	0.39
10	1	0.58
10	10	0.38
10	100	0.38
100	1	0.38
100	10	0.41

Table I. Micro–Benchmark Results

(byte swapped as necessary) reported for those fields by simulator 1. Since no value for TCP/Window, IP/TTL, or IP/Flags was specified by simulator 2, the *ItemDefault* callbacks for each of those items is called, allowing simulator 2 to determine a suitable default value. Since simulator 2 has no representation for TCP/Flags or MAC 802.3 (or any MAC layer), the simulator will create baggage items for those if they were specified as baggage by simulator 1 when registered. If the baggage flag was not specified, the items are simply discarded.

One of the strengths of the backplane design is that it allows simulators to interact at differing levels of abstraction and still exchange meaningful event messages. In the above example, simulator 1 has less detail in TCP and IP than does simulator 2, but has more detail for the MAC layer. Simulators can interact and exchange messages (provided all required protocols and items are present) through calcuation of reasonable defaults for optional data items and abstracting away optional protocol layers.

4.4 Backplane Overhead

In order to determine the CPU overhead associated with the exporting of data items to the dynamic message format and the importing of data items from the dynamic message format, we created a simple microbenchmark. A small driver program using the backplane services was implemented, which measured the overall *ExportMessage* and *ImportMessage* time, as a function of the total number of protocols and data items registered.

The results are shown in Table I. The amortized time per registered item varies depending on the mix of protocols and items, but is on the order of one-half microsecond per item. This benchmark was run on a 200Mhz Pentium Pro system running Linux.

In order to measure the overall overhead of the backplane, our parallel/distributed *ns* software (*pdns*) was modified to use the backplane for event messages being sent between the instances of the ns simulators. A simple distributed simulation consisting of three local area networks was constructed, and each of the LANs was assigned to a different processor. The *pdns* simulators used the backplane to export and import messages to peer simulators, even though they share a common event message representation. The simulation modeled FTP data flows between a pair of endpoints on different simulators, and the simulation was run for varying amounts of simulation time. For a comparison point, the same simulation was run on the unmodified *pdns*, without using the backplane.

The results are shown in Table II. Given the small overhead measured in the microbenchmark, the difference between the *ns* to *ns* run using the backplane versus the same run without the backplane should be negligible, which it is. In fact, the backplane version runs slightly faster due to the fact that the backplane produces smaller event messages than the standard *ns*. The standard *ns* uses rather large events, where the backplane exports and

Simulation Seconds	CPU Time (Backplane)	CPU Time (No Backplane)
10.0	1.7 Sec	1.7 Sec
100.0	14.5 Sec	15.0 Sec
1000.0	144.5 Sec	154.0 Sec

Table II. Homogeneous Simulation Results

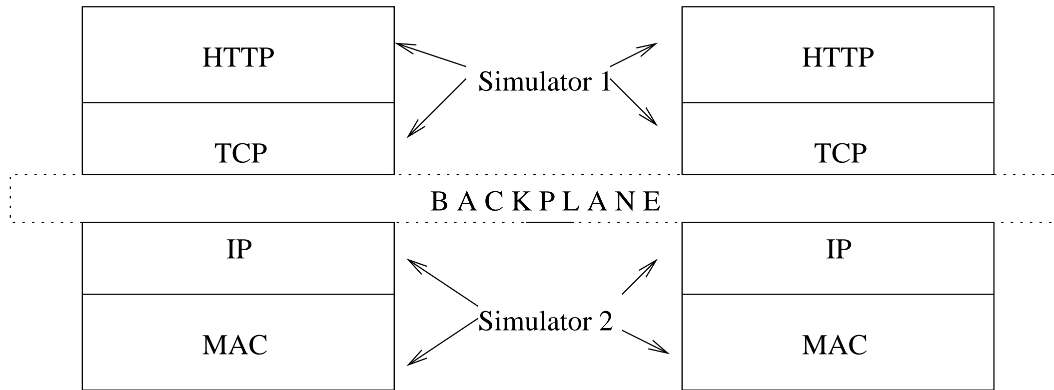

Fig. 7. Split Protocol Stack Method

sends to peers only the used portion of any given event message.

4.5 Splitting the Protocol Stack

The previous paragraphs discuss the *cross–protocol stack* method of heterogeneous simulation. In that method, the protocol stack is homogeneous within a single simulator. In other words, packets are exchanged between simulators only at the lowest supported protocol stack layer. As packets move up or down the stack, they always stay within a single simulator.

An alternative method, and one providing potentially 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 7. 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 simulators. However, this method introduces a severe limitation on the overall performance of the distributed simulation, namely the presence of a zero–lookahead message exchange.

4.5.0.1 *Lookahead.* In a conservatively synchronized, distributed discrete event simulation, one of the primary factors affecting the performance of the simulation is the pres-

ence (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 propagation 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 modeling 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 modeling 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 modeling the lower layers of the protocol stack, and the *slave* is the simulator modeling 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 simultaneously 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.

—*NERCount* 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.

4.5.0.2 *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.

4.5.0.3 *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 [Perumalla and Fujimoto 2001] 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 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

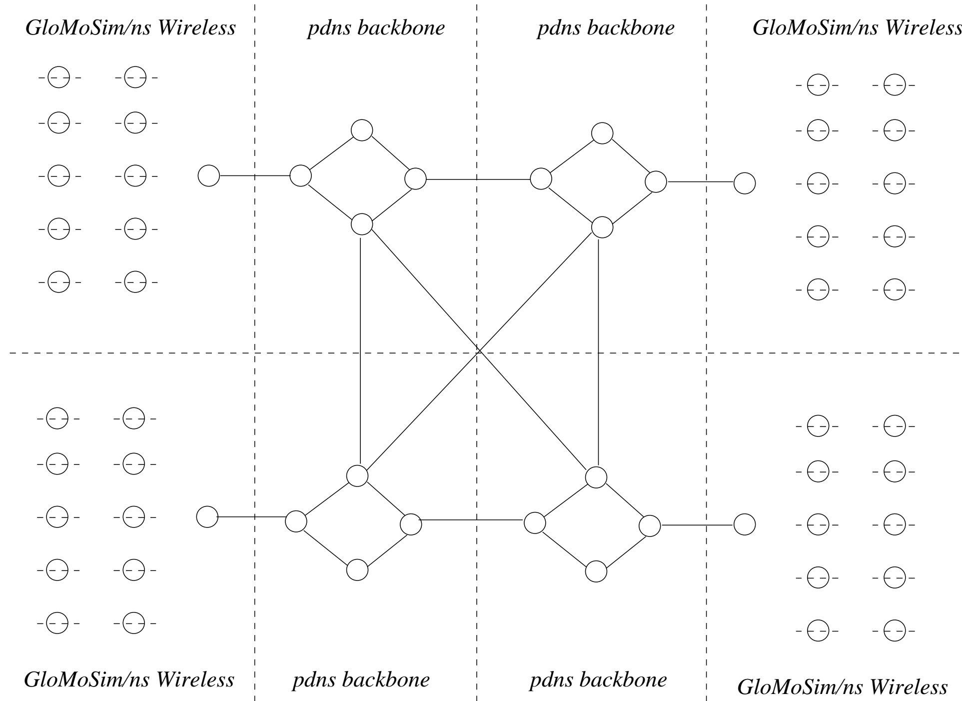

Fig. 8. Simulation Configuration with four *GloMoSim*/*ns* pairs and 4 *pdns*'s

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.

We experimented with the split protocol stack simulation using *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 8 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

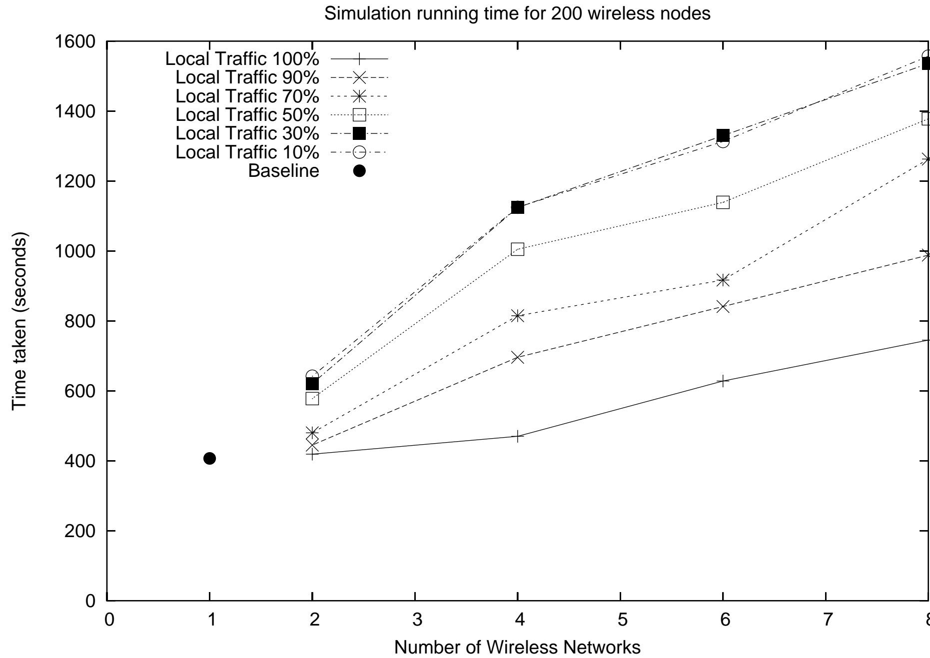

Fig. 9. Simulation with 200 Wireless Nodes in Each Wireless Network

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.

Figure 9 shows the performance when the number of wireless nodes in each wireless network is fixed at 200. The baseline case is one wireless network where 100 percent of the traffic is local traffic. 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

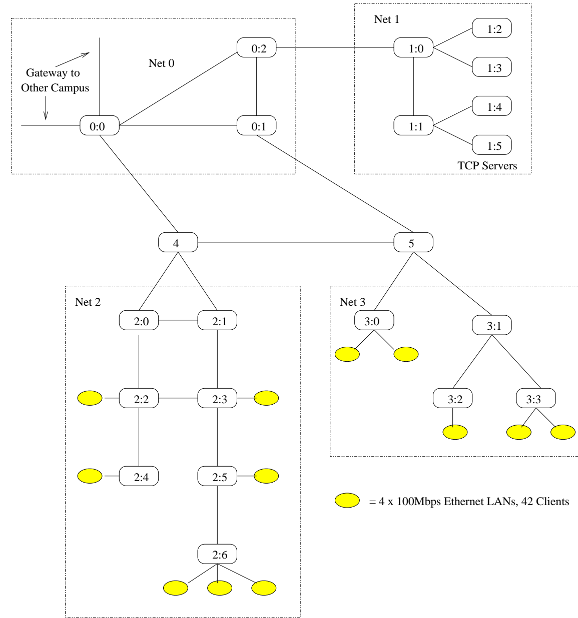

Fig. 10. Campus Network Topology

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 or 30 percent of the traffic is local traffic, running eight wireless networks plus eight *pdns* backbones still only takes about 2.5 times as the time to run two wireless networks plus two *pdns* backbones.

5. FEDERATED SIMULATION EXPERIMENTAL RESULTS

To demonstrate the effectiveness of our federated approach, we developed a scalable version of the *Campus Network* topology (shown in figure 10) developed by Nicol et. al[Nicol 2002]. In the figure, the ovals represent 4 individual 100Mbps local area networks with a total of 42 clients. The four end hosts in network 1 are the TCP servers. For all experiments discussed here, each of the 504 clients creates a TCP connection to a randomly selected server in an adjacent campus network, and sends a bulk data transfer of 500,000 bytes. The transfer starts at a random time between time 0.0 and 1.0. The simulations terminate after 20 simulated seconds (all TCP flows have completed by that time). The experiments were run using both *Georgia Tech Network Simulator* and *pdns*.

To measure the performance speedup of the federated approach, we first created a *baseline* experiment, using a single processor and a sequential simulation. This experiment consisted of four campus networks, for a total of 2152 nodes and 2016 flows. The baseline experiment was run using both *pdns* and *GTNetS* and the execution time of the simulations (not including any initialization time) was recorded. Then we ran the federated simulations, varying the number of CPU's between 2 and 30. In the federated simulation, the size of the topology and number of flows was scaled linearly with the number of CPU's assigned to the federated simulation. In other words, the topology for the 16 CPU case was

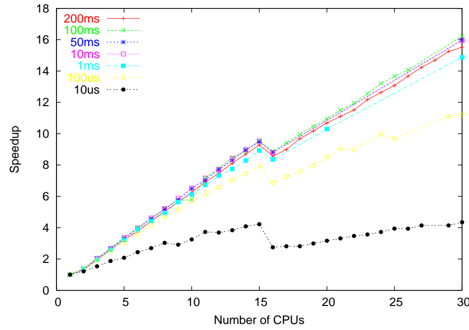

Fig. 11. GTNetS Speedup

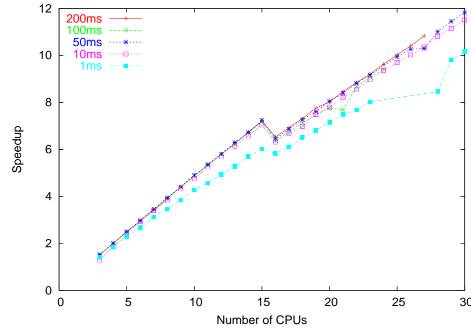

Fig. 12. PDNS Speedup

16 times as large as the sequential case, and the overall simulation performed 16 times as much work. The *speedup factor* was then calculated as follows. Let T_s be the execution time of the sequential run, and T_k be the execution time of the federated simulation on k processors (again not including any initialization or setup time). The speedup factor S is $S = T_s/T_k \times k$. Thus, if the 2 processor case runs exactly as fast the sequential case, the speedup factor is 2 representing perfect speedup (remembering that twice as much work is done). If the 2 processor case runs twice as long as the sequential case, then the speedup factor is 1, indicating no advantage was gained by adding the second CPU.

Another independent variable in the experiments was the *lookahead* between the federates. In this type of distributed simulation, the lookahead is a lower bound on the simulation time delay for events initiated on one federate but processed on another. It is well known within the distributed simulation community that large lookahead values usually lead to good distributed simulation performance. In our federated campus network topology, the lookahead value is dominated by the propagation delay along the gateway links connecting the individual campus networks in a ring. For our experiments, use used lookahead values of 1ms, 10ms, 50ms, 100ms, and 200ms.

All experiments were run on the *Ferrari Cluster* at Georgia Tech. This cluster has 16 Linux workstations, running RedHat version 8.0, each with 2 866Mhz CPU's and 2Gb of main memory.[2] The workstations are connected with a Gigabit Ethernet network and a Foundry BigIron network switch. As mentioned previously, we distributed the simulation on a varying number of CPUs between 2 and 30. For the experiments using between 2 and 15 CPUs, we assigned each of the federates to a single system, using only one of the two CPUs in that system. For the experiments using 16 to 30 CPUs, we assigned two federates to each systems, with the possibility of a single federate on the last system when the federate count is odd.

The measured speedup values for these experiments are shown in figure 11 for *GTNetS* and figure 12 for *pdns*. The results show that the performance of the federated approach scales nearly linearly as the number of federates increases. For example, both of the simulators show a speedup factor of approximately 15 on the 30 CPU case. There are two interesting things to note in these measurements. First is the small performance dip when

[2]One of the systems was being repaired when these experiments were conducted, so we only had 15 systems and 30 CPUs available at that time.

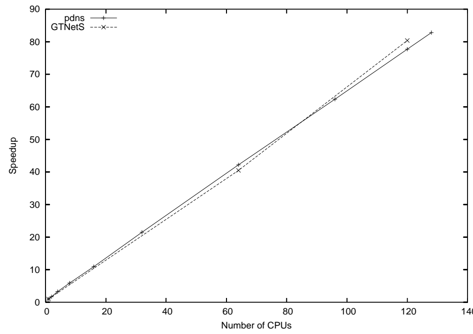

Fig. 13. PDNS and GTNetS Speedup

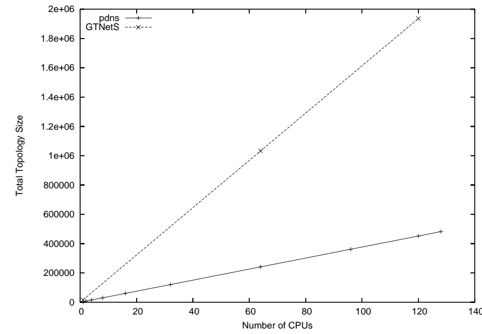

Fig. 14. PDNS and GNetS Topology Size

increasing from 15 federates to 16 federates. As previously mentioned, beginning with 16 federates we assigned two federates per system, using both of the CPUs in each system. Smaller numbers of federates used only a single CPU in each system. When using both of the available CPUs in a system for the simulation, any operating system overhead such as timer interrupts or CPU scheduling algorithms, will necessarily interfere with the progress of the simulation and slow down the overall execution slightly. When using only one CPU per system, the operating system overhead will usually be processed by the idle CPU and not interfere with the simulation.

Secondly, the speedup factors appear to be largely independent of the lookahead values in the range of 1ms to 200ms. In a properly designed conservative simulation, one would expect the number of asynchronous *LBTS* computations to be approximately $1/L \times D$, where L is the lookahead value and D is the duration of the simulation in simulated seconds. Thus for our examples, recalling that D is 20 seconds in our experiments, we would expect approximately 100 LBTS computations for the 200ms lookahead case, and 20,000 LBTS computations for the 1ms case. Given the relatively small overhead of the *LBTS* computations in these experiments (on the order of 100 microseconds per computation), this at first glance seems reasonable. However, this does not convey the whole story.

Recall that to adjust the lookahead values as above, we adjusted the propagation delay on the gateway links connecting the campus networks in a ring. Further recall that in these experiments all TCP flows are from one campus to an adjacent campus in the ring. Thus a secondary effect of reducing the lookahead value is to reduce the round–trip–time of the TCP flows being simulated. We observed that for the 200ms lookahead case, the simulated flows were completing in approximately 16 simulated seconds, where in the 1ms lookahead case all flows had completed by simulation time 5 seconds. However, the overall work performed by the simulator (number of packet transmissions) is the same in all cases. Each flow still sends 500 packets of 1000 bytes each, but with smaller round–trip–times, the TCP connections are able to send them faster. Thus, even though the smaller lookahead values result in more *LBTS* computations and smaller lookahead windows, there are more events available for processing in each lookahead window resulting in more efficiency in the smaller lookahead cases. This discussion shows that comparing simulator performance is tricky. Even when the simulators are identical, and the simulation scenarios are nearly identical, there still can be subtle differences that affect performance in unanticipated ways.

For a final set of experiments, we used a slightly larger computing cluster at Georgia

Tech for some scalability experiments. The *Jedi Cluster* has 16 individual Linux systems running RedHat 7.3. Each system has eight CPU's and 4Gb of main memory. All systems are connected via a Gigabit Ethernet interconnect. We used the same campus network scenario as described above, this time with seven campus networks per federate for the *pdns* runs (as opposed to four campus nets per federate in the previous experiments). The *GTNetS* experiments used thirty campus networks per federate, since the *GTNetS* implementation uses a more memory efficient representation for the simulated topology and can model a larger topology in a single federate. The number of federates (and CPU's) was varied from 1 to 128, and again the total workload of the simulation was increased linearly as the number of federates increased. The speedup factors were calculated as above.

The speedup results for *pdns* and *GTNetS* are shown in figure 13. This set of results are for the 200ms lookahead case. Again, we see linear speedup for both *pdns* and *GTNetS*, up to a total of 128 federates. In fact, the speedup values for the Jedi cluster are slightly better than those measured on the smaller Ferrari cluster. This is due to the faster CPU speed on the Ferrari cluster, but with comparable interconnect hardware. The faster CPU runs the baseline experiment substantially faster, but the time management and event distribution overhead (for the federated experiments) is nearly the same.

Finally, figure 14 shows the scalability of the overall topology size achievable with our federated approach. Both *pdns* and *GTNetS* show linear scalability in topology size, with *GTNetS* successfully simulating a topology of 1,936,800 nodes (and nearly the same number of TCP flows) with the 120 CPU experiment. The *pdns* experiment on 128 CPU's achieved an overall topology size of 482,048 nodes.

6. SUMMARY AND CONCLUSIONS

Ideally, one might envision a simulation environment where one could pick and choose components from different simulation packages and automatically configure them to create models for large-scale networks that execute efficiently on a wide variety of platforms ranging from Supercomputers to cluster computers to networked workstations. While we do not claim to have created such an environment, this paper attempts to describe some of the challenges, a few solution approaches, and our experiences in attempting to realize such a capability.

Key issues that arise in homogeneous federations where a single simulator is federated with itself include dealing with physical and logical interconnections between simulators, and addressing routing issues. Our experiences with two self-federated distributed network simulations, PDNS and GTNets have been positive. Experimental results confirm the ability of the federated simulation approach to achieve large (nearly 2 million network nodes) simulation topologies with linear efficiency up to 128 federates. We believe a central conclusion of this work is that self-federated network simulations represent a viable approach to parallel network simulation. Based on our prior experiences with parallel network simulation tools that were developed "from scratch," we believe the federated approach can yield comparable performance while offering substantial advantages arising from model and software reuse, provided a reasonably efficient sequential simulator is used. We note, however, that our experiences with the self-federated approach has not been uniformly positive for all network simulators. Widespread use of global state severely limited our exploitation of this technique to realize a parallel implementation of the Opnet simulator [Wu et al. 2001].

To extend these results to heterogeneous federations that include different simulation packages, a backplane approach was proposed, and two approaches to federating simulators were realized based on different approaches to partitioning the network model. Key issues in extending these techniques to federate dis-similar simulators include handling issues concerning differing representations, incomplete implementation of protocols, and federating simulators using different levels of abstraction. We have shown that the additional computation overhead to address the first two issues can be reduced to negligible levels. We note that while we were able to successfully federate disparate simulators such as GloMosim and *ns*, a non-trivial amount of effort was required to identify and isolate problems. Although the problems, once identified, were easily addressed, it is clear that we are still far from achieving 'plug 'n play" interoperability among network simulation packages.

REFERENCES

BAJAJ, S., BRESLAU, L., ESTRIN, D., FALL, K., FLOYD, S., HALDAR, P., HANDLEY, M., HELMY, A., HEIDEMANN, J., HUANG, P., KUMAR, S., MCCANNE, S., REJAIE, R., SHARMA, P., SHENKER, S., VARADHAN, K., YU, H., XU, Y., AND ZAPPALA, D. 1998. Virtual internetwork testbed: Status and research agenda. USC Computer Science Dept, Technical Report 98-678.

BANKS, J. 1996. *Discrete Event System Simulation*. Prentice-Hall.

BROOKS, D. E. 1986. The butterfly barrier. *The International Journal of Parallel Programming 14*, 295–307.

BRYANT, R. E. 1977. Simulation of packet communications architecture computer systems. In *MIT-LCS-TR-188*.

CAROTHERS, C. D., BAUER, D., AND PEARCE, S. 2000. Ross: a high-performance, low memory, modular time warp system. In *14th Workshop on Parallel and Distributed Simulation*. 53 – 60.

CHANDY, K. AND MISRA, J. 1979. Distributed simulation: A case study in design and verification of distributed programs. In *IEEE Transactions on Software Engineering*.

COWIE, J., OGIELSKI, A., AND NICOL, D. 2002. The SSFNet network simulator. Software on-line: http://www.ssfnet.org/homePage.html. Renesys Corporation.

FISHMAN, G. S. 1978. *Principles of Discrete Event Simulation*. Wiley.

FUJIMOTO, R., FERENCI, S., LOPER, M., MCLEAN, T., PERUMALLA, K., RILEY, G., AND TACIC, I. 2001. Fdk users guide. Georgia Institute of Technology.

FUJIMOTO, R. M., PERMUALLA, K., AND TACIC, I. 2000. Design of high performance RTI software. In *Distributed Simulation and Real-Time Applications 2000*.

HOFER, R. C. AND LOPER, M. L. 1995. DIS today. *Proceedings of the IEEE 83,* 8 (Aug), 1124–1137.

KUHL, F., WEATHERLY, R., AND DAHMANN, J. 1999. *Creating Computer Simulation Systems: An Introduction to the High Level Architecture*. Prentice-Hall.

MATTERN, F. 1993. Efficient algorithms for distributed snapshots and global virtual time approximation. In *Journal of Parallel and Distributed Computing*.

MCCANNE, S. AND FLOYD, S. 1997. The LBNL network simulator. Software on-line: http://www.isi.edu/nsnam. Lawrence Berkeley Laboratory.

MILLER, D. C. AND THORPE, J. A. 1995. SIMNET: The advent of simulator networking. *Proceedings of the IEEE 83,* 8 (Aug), 1114–1123.

NICOL, D. AND HEIDELBERGER, P. 1996. Parallel execution for serial simulators. *ACM Transactions on Modeling and Computer Simulation 6,* 3 (July), 210–242.

NICOL, D. M. 2002. The baseline campus network explained. http://www.cs.dartmouth.edu/ nicol/ NMS/ baseline/. DARPA Network Modeling and Simulation (NMS).

PERUMALLA, K., FUJIMOTO, F., MCLEAN, T., AND RILEY, G. 2002. Experiences applying parallel and interoperable network simulation techniques in on-line simulation of military networks. In *16th Workshop on Parallel and Distributed Simulation*.

PERUMALLA, K., FUJIMOTO, R., AND OGIELSKI, A. 1998. Ted - a language for modeling telecommunications networks. *ACM SIGMETRICS Performance Evaluation Review 25,* 4 (March).

PERUMALLA, K. S. AND FUJIMOTO, R. M. 2001. Virtual time synchronization over unreliable network transport. In *15th Workshop on Parallel and Distributed Simulation*.

POOCH, U. W. 1993. *Discrete Event Simulation: A Practical Approach*. CRC Press.

REKHTER, Y. AND LI, T. 1995. Internet RFC1771: A border gateway protocol 4 (bgp-4). Network Working Group.

RILEY, G. F. 2003. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*. ACM Press, 5–12.

RILEY, G. F., FUJIMOTO, R. M., AND AMMAR, M. H. 1999. A Generic Framework for Parallelization of Network Simulations. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*.

WILSON, A. L. AND WEATHERLY, R. M. 1994. The aggregate level simulation protocool: An evolving system. In *Proceedings of the Winter Simulation Conference*. 781–787.

WU, H., FUJIMOTO, R., AND RILEY, G. 2001. Experiences parallelizing a commercial network simulator. In *Proceedings of the Winter Simulation Conference*.

XIAO, Z., UNGER, B., SIMMONDS, R., AND CLEARY, J. 1999. Scheduling critical channels in conservative parallel simulation. In *13th Workshop on Parallel and Distributed Simulation*.

ZENG, X., BAGRODIA, R., AND GERLA, M. 1998. GloMoSim: a library for parallel simulation of large-scale wireless networks. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*.

ZHOU, J., JI, Z., TAKAI, M., AND BAGRODIA, R. 2003. Maya: a multi-paradigm network modeling framework for emulating distributed applications. In *17th Workshop on Parallel and Distributed Simulation*. 163–170.

Recieved June 2003, accepted December 2003.