Distributed Network Simulations using the Dynamic Simulation Backplane[1]

George F. Riley
Mostafa H. Ammar
Richard Fujimoto
Kalyan Perumalla
Donghua Xu
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{riley,ammar,fujimoto,perumalla,xu}@cc.gatech.edu
(404) 894-6705, Fax: (404) 385-0332
Aug 28, 2000

We present an approach for creating distributed, component-based, simulations of communication networks by interconnecting models of sub-networks drawn from different network simulation packages. This approach supports rapid construction of simulations for large networks by reusing existing models and software, and fast execution using parallel discrete event simulation techniques. A *dynamic simulation backplane* is proposed that provides a common format and protocol for message exchange, and services for transmitting data and synchronizing heterogeneous network simulation engines. In order to achieve "plug-and-play" interoperability, the backplane uses existing network communication standards, and dynamically negotiates among the participant simulators to define a minimal subset of required information that each simulator must supply, as well as other optional information. The backplane then automatically creates a message format that can be understood by all participating simulators and dynamically creates the content of each message by using callbacks to the simulation engines. This paper describes our approach to interoperability as well as an implementation of the backplane. We present results that demonstrate the proper operation of the backplane by distributing a network simulation between two different simulation packages, ns2 developed at USC/ISI and GloMoSim developed at UCLA. We present performance results that show that the overhead for the creation of the dynamic messages is minimal. Although this work is specific to network simulations, we believe our methodology and approach can be used to achieve interoperability in other distributed computing applications as well.

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

1. Introduction

Distributed network simulations exchange information using event messages, which typically model the data packets flowing between the simulated network elements. When the processes composing the distributed simulation are homogeneous, then all can easily agree on the content and meaning of the event messages. However, when exchanging event messages between heterogeneous simulators, several interesting problems arise. How do the simulators agree in advance on the representation of a simulated data packet? How can a simulator insist that a particular protocol header must be present? How can a simulator specify the level of detail that is modeled for a particular protocol? What should a simulator do when presented with protocol information for which it has no internal representation?

To address these issues, we introduce the *Dynamic Simulation Backplane,* which provides a common event message-passing interface between distributed simulations. The backplane creates a dynamic format for network events 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 defines a common *API* for simulators to describe which network protocols are supported and which data elements within each protocol are required or available by that simulator. Finally, the backplane supports baggage data, which occurs when a given simulator must retain protocol information of interest only to another simulator.

1.1. Motivation

There are several commercially or publicly available network simulation packages, each of which has its strengths and weaknesses. The *ns[1]* network simulator has a rich set of end-to-end network protocols, and a variety of routing element queuing disciplines. The OpNet[2] simulator has a large database of network equipment models, including routers and switches from several network equipment vendors. The GloMoSim[3] simulation engine provides strong support for wireless networks with mobility. Our research studies the interoperability of these heterogeneous network simulators, thereby allowing the simulator modeler to describe and simulate each portion of a network with the simulator most well suited for that portion of the network. In an ideal world, a network modeler could use a different network simulator for different portions of the entire network model, selecting the best simulator for the simulation requirements of that portion of the model. For example, we might choose the *ns* simulator to model the behavior of the TCP endpoints, using one of the rich set of TCP models available in *ns*. Next we might choose *GloMoSim* to model a wireless local area network where the TCP endpoints are attached. Finally, we might choose *OpNet* to model a wide area wired network connecting the wireless LANs together, selecting the network routers from the large database of network equipment supported by *OpNet*.

As a second example, suppose that a network modeler has previously developed a detailed model of a local area network using the *OpNet* simulator. A second modeler has created a model of a wide area network using *ns*. Finally a third modeler has developed a good model of wireless local area network using GloMoSim. Each of the three models has been thoroughly tested and each modeler is confident of the correctness of the model. Should the three modelers want to combine the simulation into one large model, they are faced with two possibilities. One solution is to convert two of the three models into the same environment as the third (i.e. convert all models to the *ns* simulation environment). By doing this, the confidence in two of the three existing models is lost, due to the modifications to the model required by the conversion. A better solution would be to run each of the three simulations in their native environment, together with a method to allow event messages to be exchanged between the simulation engines.

For a third example, suppose that a given network model is too large to be defined and simulated within the physical memory constraints of a single workstation. With a good method for distributing the simulation on two or more workstations, the overall size of the network being simulated can grow almost linearly with the number of workstations.

1.2. Related Work

The distributed execution of a *single* network simulation, either on several workstations or on a tightly coupled SMP system, has been studied for some time. Cowie et al.[4, 5] describe the Scaleable Simulation Framework (SSF) as a method for parallel simulation of large scale networks. Nicol et al.[6] propose IDES, a Java based simulation engine designed specifically for distributed network simulations. Perumalla et al.[7, 8] created the Telecommunications Description Language (TED), which allows multithreaded network simulations on an SMP processor. Bagrodia et al.[3] developed the GloMoSim simulation environment, which is built on top of the Parsec[9] parallel simulation environment. Riley et al.[10] designed and implemented Parallel/Distributed *ns* (*pdns*), which allows a single *ns* simulation to be distributed on a network of workstations. All of the previous work has, however, been focused on a homogeneous simulation environment. All of the distributed processes are running the same simulation engine, and thus the semantics of event messages transferred between remote simulators is the same. Event messages can be transmitted between simulators as just a “Bag of Bits”, without regard to the internal representation of these events. Clearly, when exchanging messages between heterogeneous simulators, the bag of bits approach will not work.

The High Level Architecture (HLA) [11] provides a standardized API for simulation engines to register objects and request notification of object updates. While this approach does not limit the distributed simulation to a common simulation engine, it does require the simulations to agree on the format of the objects being exchanged. To contrast this with our work, we make no assumptions regarding representation of messages internally in the simulator.

1.3. The Dynamic Backplane Approach

In order for heterogeneous simulators to exchange meaningful event messages, there must be some common ground for the semantics and meaning of the information being exchanged. In the realm of network simulations, a good starting point is the published standards for network protocols. Any simulator that supports the simulation of data flows using the TCP protocol[12] must have some understanding of at least some subset of the data items specified in RFC793. While a complete implementation for all TCP variations and all TCP protocol fields may not be present, each simulation must at least have some notion of a *Sequence Number*. Similarly, if the simulator supports the routing of simulated packets using the IP protocol[13], then some parts of the data items specified in RFC791 must be known. Again, all of the items may not be supported, such as the fragmentation of packets, but at a minimum some notion of a *Destination Address* must be understood.

With this in mind, we designed the backplane using the concept of *Protocols* and *Data Items*. Each simulator registers with the backplane a complete list of the protocols that are known to that simulator. Within each protocol, the simulator registers which of the data items defined in that protocol are supported. However, with the understanding that network simulations are often used to promote experimental protocols or extensions to existing protocols, the backplane does not limit the registration only to standardized protocols or data items. A simulator may register any protocol, or any data item within a protocol. As long as one other simulator registers a protocol or item by the same name, those simulators can exchange meaningful information.

Once all protocols and items are registered, the backplane negotiates between the participants, using a global consensus protocol, to obtain a complete picture of the registered protocols and items. Using the information from the global consensus, the backplane can then create dynamic format messages (on a message-by-message basis) to exchange information between simulators. The details of the registration process and the global consensus are given later.

The remainder of this paper is organized as follows. Section 2 describes in detail the design and operation of the backplane. Section 3 gives a description of experiments we used for demonstrating the viability of the backplane and lists some performance results. Finally, Section 4 states some conclusions and gives the future direction of our research

2. The Dynamic Simulation Backplane

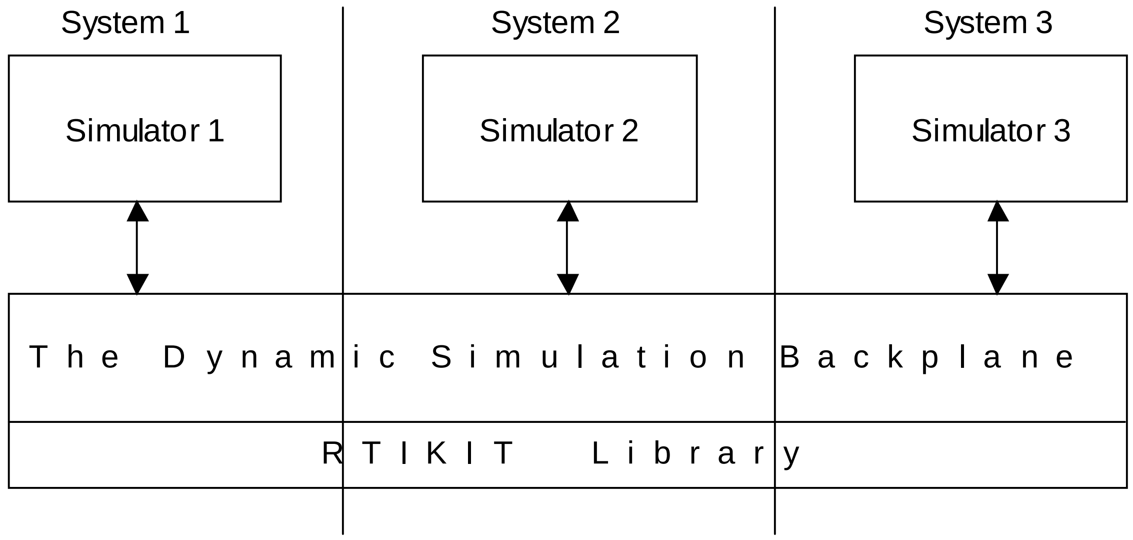

Figure 1, Dynamic Simulation Backplane Architecture

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

The backplane and RTIKIT services fall into five basic categories:

1. Protocol/Item Registration Services

2. Consensus Computation

3. Message Importing/Exporting Services,

4. Simulation Time Management Services, and

5. Event Distribution Services.

2.1. 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. We chose to use these standards 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 he *Export Query Callback*, 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 Export Query Callback, 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.

2.2. 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. There are several possibilities.

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.

2.3. Message Importing/Exporting Services

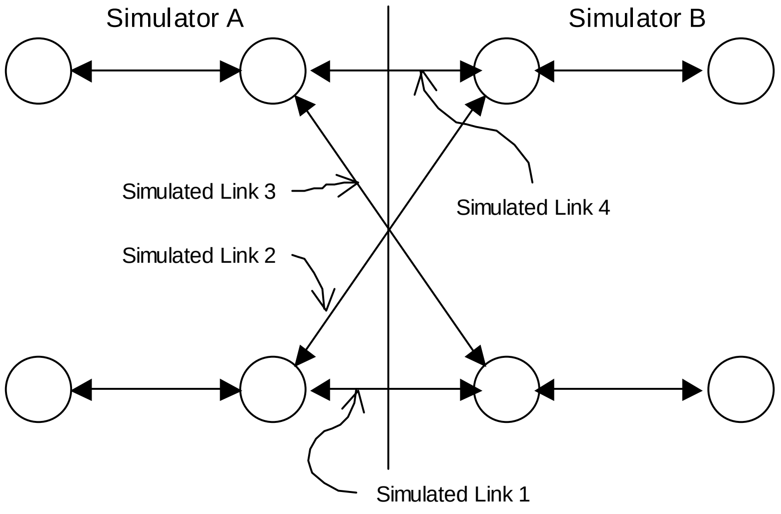

Figure 2, Simple Distributed Simulation

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 2. 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.

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.

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 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.

Baggage

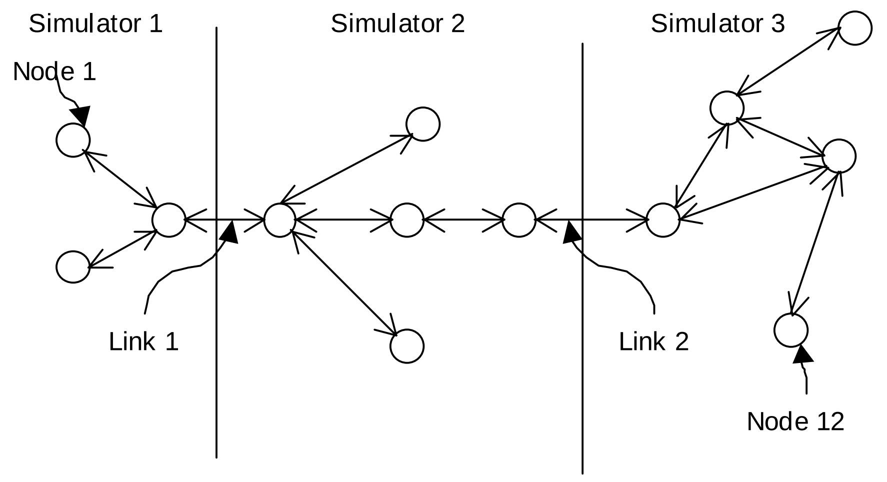

Figure 3, Baggage Example

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 3. 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 12. It is clear that when simulated packets arrive at node 12 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 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.

Importing/Exporting Example

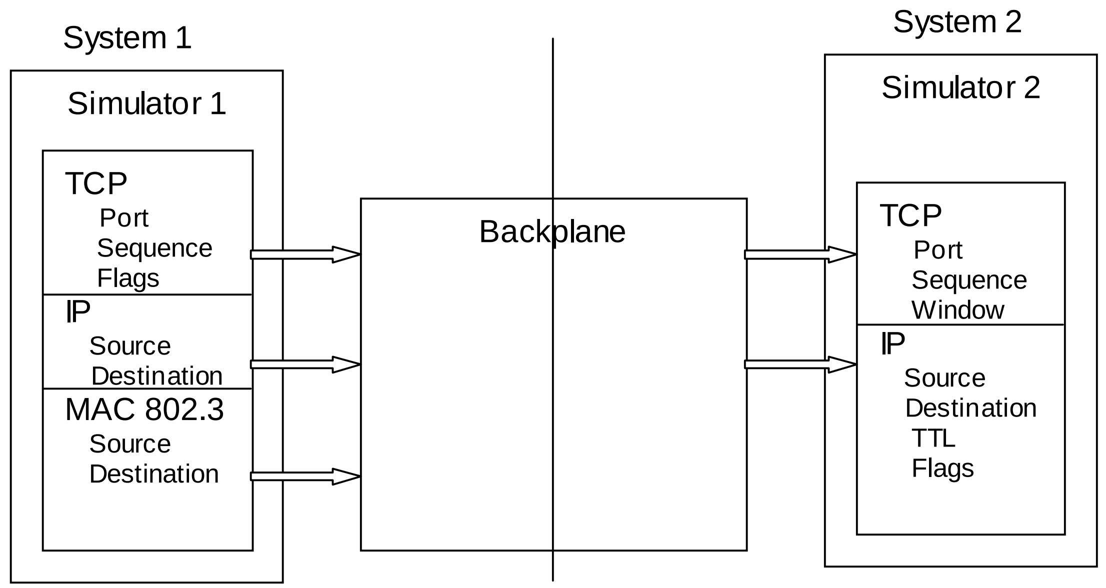

Figure 4, Exporting and Importing Example

Figure 4 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 has created 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 (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. By allowing simulators to calculate reasonable defaults for optional data items, and by abstracting away entire optional protocol layers, simulators can still interact and exchange messages, providing that all required protocols and items are present.

2.4. Simulation Time Management Services

An important requirement for any distributed discrete event simulation is the proper management of simulation time advancement. The participating simulators cannot just advance their own local view of the simulation time as fast as possible. Instead, they must insure that they will never receive an event message in the simulated past. To accomplish this constraint, the simulators must periodically participate in a global consensus computation to determine a lower bound on the timestamp of the smallest unprocessed event message, including event messages that are in transit from one simulator to another. This global minimum timestamp value is called the *Lower Bound Time Stamp* (LBTS). Once this LBTS value is determined, all simulators can use this value as an upper bound on the local simulation time advancement. If no simulator advances the local simulation time beyond the computed LBTS value, and if no simulator sends an event message in the simulated past, then it can be guaranteed that no simulator will receive events in their local view of the simulated past.

A number of approaches to computing the LBTS exist [14-18]. The backplane makes use of time management services provided by the RTIKIT [19], which uses a butterfly barrier technique first proposed by Brooks [20]. Using the butterfly barrier, simulators exchange local LBTS and message count information with each other in a series of rounds. After the completion of a fixed number of rounds, all processors have agreement on the global state of the unprocessed messages, and can thus compute an LBTS value.

The backplane provides the time management services by use of the *NextEventRequest* function. When calling this function, simulators specify the timestamp of the next unprocessed event in their local event queue. The backplane responds with a *TimeAdvanceGrant* callback, which reports the time to which this simulator may safely advance the local simulation time. The granted value is always less than or equal to the requested time in the *NextEventRequest* call.

2.5. Event Distribution Services.

Another requirement for all distributed simulations is *Data Distribution*. Often an event message is created at one simulator that in fact must be processed at some other simulator. Considering again the sample distributed simulation shown in Figure 2, simulator A must inform simulator B of receive packet events for any simulated packets sent on links 1, 2, 3, or 4. In this case it is not sufficient for A to inform B of a received packet. It must also advise B of which of the 4 links the packet is to be received on.

The backplane again makes use of the services provided by the RTIKIT. The RTIKIT defines links as *Object Instances*, and allows simulators to register *interest* in object instances. Simulator A would register each of the links as an instance of a link object. Simulator B would register interest in any actions affecting the link object. When generating a data packet to be sent on link 1, Simulator A calls the *UpdateAttributeValues* backplane function, specifying the link object being updated, the timestamp of the event, and the data associated with the event. When this function is called, any simulator previously registering interest in this object is notified that the event occurred.

3. Experimental Methodology and Results

To demonstrate the feasibility of the backplane approach, and to measure the overhead incurred by the conversion of messages to and from the dynamic format, we devised a series of three benchmarks. Those are:

1. A micro-benchmark, where we coded a simple wrapper around the backplane, registered a varying number of protocols and data items, and measured the CPU time required for the *ExportMessage* and *ImportMessage* functions.

2. A homogeneous, distributed network simulation using three processes all running the ns network simulator. This simulation was run with two different versions of pdns, first one without the backplane, and the second using the backplane for all event messages sent from one simulator to another.

3. A simple heterogeneous simulation, as depicted in Figure 5. For this experiment, we used the Telnet application that is part of the GloMoSim simulation engine as the data flow endpoints. The ns simulation engine was used to model a small wide area network connecting the two GloMoSim wireless LANs.

3.1. Item Exporting and Importing Overhead

The purpose of the micro-benchmark was simply to measure 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. A simple wrapper around the backplane 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 1 below. 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.

Number Protocols	Number Items / Proto	Total Items	Microseconds per Item
1	1	1	0.77
1	10	10	0.39
10	1	10	0.58
10	10	100	0.38
10	100	1000	0.38
100	1	100	0.58
100	10	1000	0.41
100	100	10000	0.42

Table 1, Micro-Benchmark

3.2. Backplane Overhead in Homogeneous Simulation

Next the parallel/distributed ns software 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 simulation modeled FTP data flow 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 2 below. Given the small overhead determined in the micro-benchmark, 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 somewhat smaller event messages than the standard ns. The standard ns uses rather large events, where the backplane exports and sends to peers only the used portion of any given event message.

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

Table 2, Homogeneous Simulation

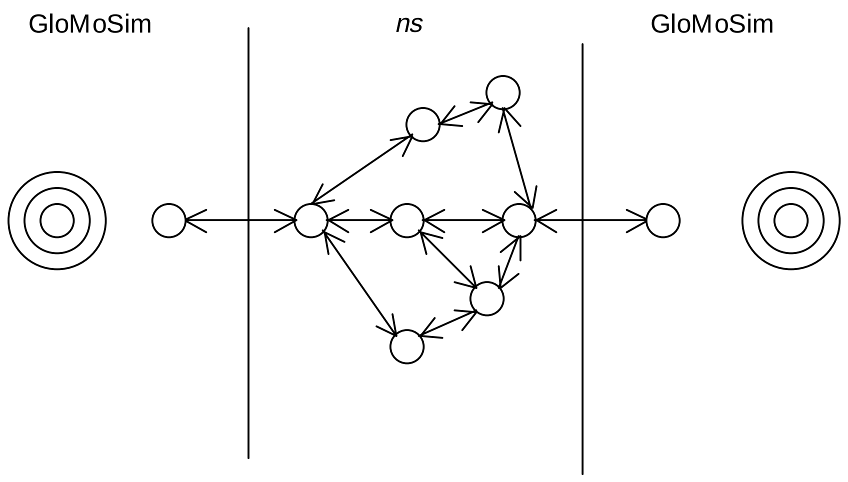

Figure 5, Experimental Heterogeneous Simulation Model

3.3. Heterogeneous Simulation Demonstration

Finally we used the backplane to implement the simple distributed simulation shown in Figure 5, consisting of two GloMoSim wireless nodes, connected via a small *ns* wired network. The GloMoSim endpoints modeled a Telnet connection and the ns network forwarded the simulated packets between the two wireless endpoints. All simulators registered the IP and TCP protocols. Each simulator registered the data items for those protocols specific to their unique implementation.

This simulation demonstrates the proper operation of baggage data items, since a number of GloMoSim specific data items are used which have no meaning in the *ns* environment. The flow of packets through the backplane was tracked using debug messages showing the contents of the packets and the number of packets processed. While this simulation, by design, modeled only a single data flow and a small number of packets, the overall operation of the backplane was verified. No performance numbers are shown here, since there is no easy way to determine any comparison data.

5 Conclusions and Future Work

We believe the Dynamic Simulation Backplane is a viable approach for interconnecting heterogeneous simulations of computer networks. The experimental results show that the overhead to convert messages to a dynamic format is small enough to be inconsequential; and in fact can give slightly better performance due to the selective exporting of data items.

For future work, we are planning on more experimentation with the GloMoSim to *ns* interfaces, using more protocols and more data items. We also are planning on integrating the OpNet network simulator into the backplane environment, although this effort is complicated by the lack of source code for OpNet.

5. Bibliography

1. McCanne, S. and S. Floyd, *The {LBNL} Network Simulator*. 1997.

2. Bertolotti, S. and L. Dunand, *Opnet 2.4: an environment for communication network modeling and simulation*, in *Proceedings of the European Simulation Symposium*. 1993.

3. Zeng, X., 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*. 1998.

4. Cowie, J., et al., *Towards Realistic Million-Node Internet Simulations*, in *International Conference on Parallel and Distributed Processing Techniques and Applications*. 1999.

5. Cowie, J.H., D.M. Nicol, and A.T. Ogielski, *Modeling the Global Internet.* Computing in Science and Engineering, 1999.

6. Nicol, D., et al., *IDES: A Java-based Distributed Simulation Engine*, in *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 1998.

7. Perumalla, K.S. and R.M. Fujimoto, *Efficient Large-Scale Process-Oriented Parallel Simulations*, in *Proceedings of the Winter Simulation Conference*. 1998.

8. Perumalla, K., R. Fujimoto, and A. Ogielski, *TeD - A Language for Modeling Telecommunications Networks.* Performance Evaluation Review, 1998. **25**(4).

9. Bagrodia, R., et al., *Parsec: A Parallel Simulation Environment for Complex Systems.* IEEE Computer, 1998. **31**(10): p. 77-85.

10. Riley, G.F., R.M. Fujimoto, and M.A. 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*. 1999.

11. Myjak, M.D., et al., *Implementing Object Transfer in the HLA*, in *Proceedings of the Spring Simulation Interoperability Workshop*. 1999.

12. Postel, J., *Internet RFC793: Tramsmission Control Protocol.* IETF Network Working Group, 1981.

13. Postel, J., *Internet RFC791 : Internet Protocol*, in *IETF Network Working Group*. 1981.

14. Mattern, F., *Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation*, in *Journal of Parallel and Distributed Computing*. 1993.

15. Chandy, K.M. and J. Misra, *Asynchronous Distributed Simulation via a Sequence of Parallel Computations.* Communications of the ACM, 1981. **24**(4): p. 198-205.

16. Riley, G.F., R.M. Fujimoto, and M.A. Ammar, *Network Aware Time Management and Event Distribution*. 2000.

17. Steinman, J., *SPEEDES: Synchronous Parallel Environment for Emulation and Discrete Event Simulation*, in *Advances in Parallel and Distributed Simulation*. 1991, SCS Simulation Series. p. 95-103.

18. Nicol, D.M., *The Cost of Conservative Synchronization in Parallel Discrete Event Simulations.* Journal of the Association for Computing Machinery, 1993. **40**(2): p. 304-333.

19. Fujimoto, M., Perumalla, Tacic. *Design of High Performance RTI Software*. in *Distributed Simulation and Real-Time Application*. 2000. SanFrancisco, CA.

20. Brooks, D.E., *The Butterfly Barrier.* The International Journal of Parallel Programming, 1986. **14**: p. 295-307.