Proceedings of the 1992 Winter Simulation Conference
ed. J. J. Swain, D. Goldsman, R. C. Crain, and J. R. Wilson

A DISTRIBUTED, OBJECT-ORIENTED COMMUNICATION NETWORK SIMULATION TESTBED

M. Scott Corson

Electrical Engineering Department
and Systems Research Center
University of Maryland, College Park

ABSTRACT

The Distributed Network Simulation Testbed is a research tool designed to aid users in the modeling and performance analysis of communication protocols. The testbed can be used to model communication networks consisting of mobile and/or immobile nodes communicating over broadcast and/or point-to-point channels. Its object-oriented design permits users to utilize previously developed modules and, when necessary, to derive new modules which are subsequently added to the testbed. The testbed, implemented in Sim++, can run either sequentially on a single processor or in parallel on multiple processors. An object-oriented, graphical user interface allows users to monitor simulation progress for both demonstration and debugging and permits users to graphically construct network simulations from existing testbed components.

1 INTRODUCTION

The Distributed Network Simulation Testbed is a research tool designed to aid users in the modeling and performance analysis of communication protocols. It serves as a reusable simulation platform upon which users develop individually tailored simulations. The testbed's primary design goal is to reduce time necessary for users to develop communication network simulations. This goal is accomplished through object-oriented software reuse. When using the testbed, users construct communication network simulations from existing testbed components and when necessary, develop new components for their simulation which are then added to the testbed for future use. The secondary design goal is to permit distributed, multi-processor simulation. Distributed execution permits simulation of much larger networks than would be possible on a single workstation (due to memory constraints) and, for some applications, gives parallel performance speed-up.

The testbed is written in the Sim++ Programming Language, a process-oriented discrete-event simulation language imbedded in the object-oriented language C++. The testbed's design is *object-oriented* and hence, very modular allowing it to be incrementally built and used. For those unfamiliar with object-oriented programming, a good introduction is given in Wiener and Pinson (1988). The testbed may be configured for two modes of execution, single processor or multi-processor. Given the same initial conditions, both modes produce the identical simulation runs and either may be advantageous, measured in terms of execution speed, depending on the network being simulated.

2 ABSTRACT SYSTEM MODEL

The system model stems from a top-down, object-oriented view of a network. The design approach is to combine all possible communication network models and, from this aggregation, extract those core components which are present in all models. These core components are the abstract data types or objects of the object-oriented model and form the simulation's abstract framework. Each abstract type heads an object hierarchy derived from the abstract type. The relationships between the abstract objects define the basic relationships between their respective derived non-abstract object hierarchies. Users are not permitted to create instances of abstract objects but must construct simulations from the more specialized objects derived from the abstract objects. The following outlines the simulation's major abstract data types, the name of each is italicized when introduced.

At its highest level, a communication network consists of a set of *nodes* which exchange messages with one another. The nodes may be mobile (cellular user, satellite, personal communications user, etc.) or fixed (satellite ground station, telephone switch,

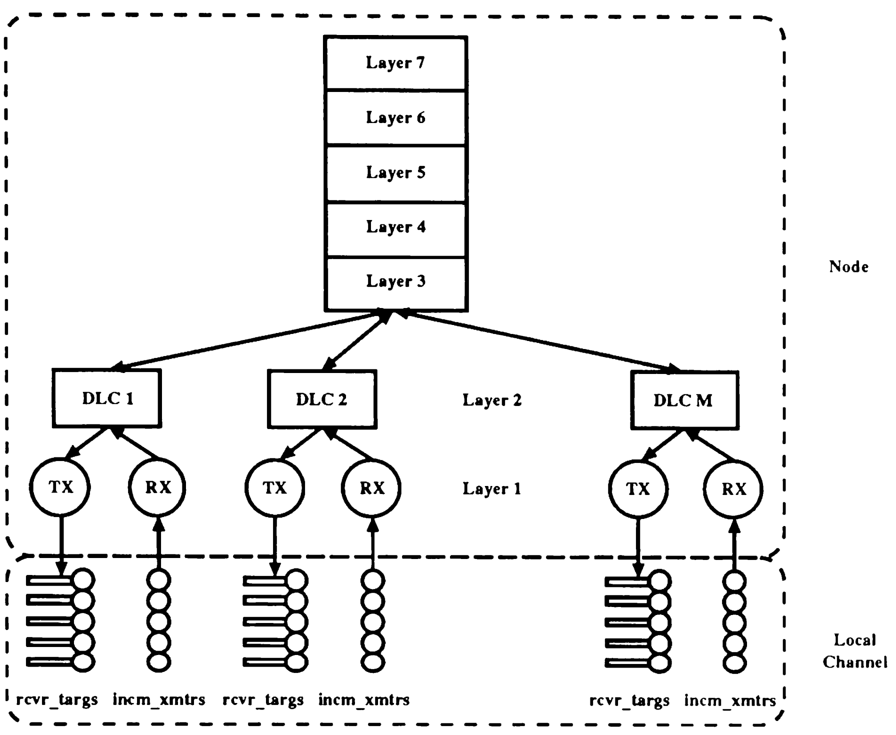

Figure 1: OSI 7-layer model

LAN node, etc.). Nodes are broken down into *layers* according to the 7-layer OSI model (see Fig. 1). Layers 2-7 model communication protocols and consist of protocol objects whereas layer 1 models the physical hardware layer and consists of *transmitters* and *receivers*. Both transmitters and receivers are further separated into two abstract types, *broadcast* and *point-to-point*; a distinction which will be made clear shortly.

The nodes communicate by sending messages over communication *links*. Links, which are also subdivided into the abstract varieties broadcast and point-to-point, connect a transmitter to a receiver. The implication is that a *broadcast link* type connects a *broadcast transmitter* to a *broadcast receiver* and that a *point-to-point link* type connects a *point-to-point transmitter* to a *point-to-point receiver*. A point-to-point link models a physically hardwired connection such as a fiber optic cable or telephone line. A broadcast link models a wireless, possibly temporary connection between a transmitter and a receiver within range of the transmitter. All links consist of *physics* objects and *noise* objects. Physics objects determine parameters such as propagation delay and signal attenuation. Noise objects determine received SNR and model the link's particular noise phenomena, possibly statistically corrupting the packet's content.

Communication *channels* are either broadcast or point-to-point and consist of a collection of appropriately typed links. Broadcast channels are assigned a portion of the spectrum, determined by center frequency and bandwidth, and transmissions on these frequencies occur over broadcast links assigned to those channels. Hence, link type and frequency determine the assignment of broadcast links to broadcast channels. For point-to-point channels, the assignment of links to channels is determined solely by link/channel type. For instance, a telecomm network may consist of both fiber and copper links, each group constituting a different logical point-to-point channel. The relationship between transmitters, receivers, and links can be seen for both channel types in Fig. 2. Note, the transmitter/receiver mapping is 1-1 for point-to-point channels and 1-N for broadcast channels.

In modern networks, messages are exchanged between nodes in the form of *packets*. Packets are composed of bits which are grouped into data *fields*, each of which has a fixed bit length and conveys a special meaning to the protocols.

Users construct simulations, not from the abstract data types mentioned above but, from subtypes derived from the abstract types. Yet, the basic relationships between the abstract objects is the inherited by the derived objects. For instance, a cellular simulation may consist of two types of nodes, cellu-

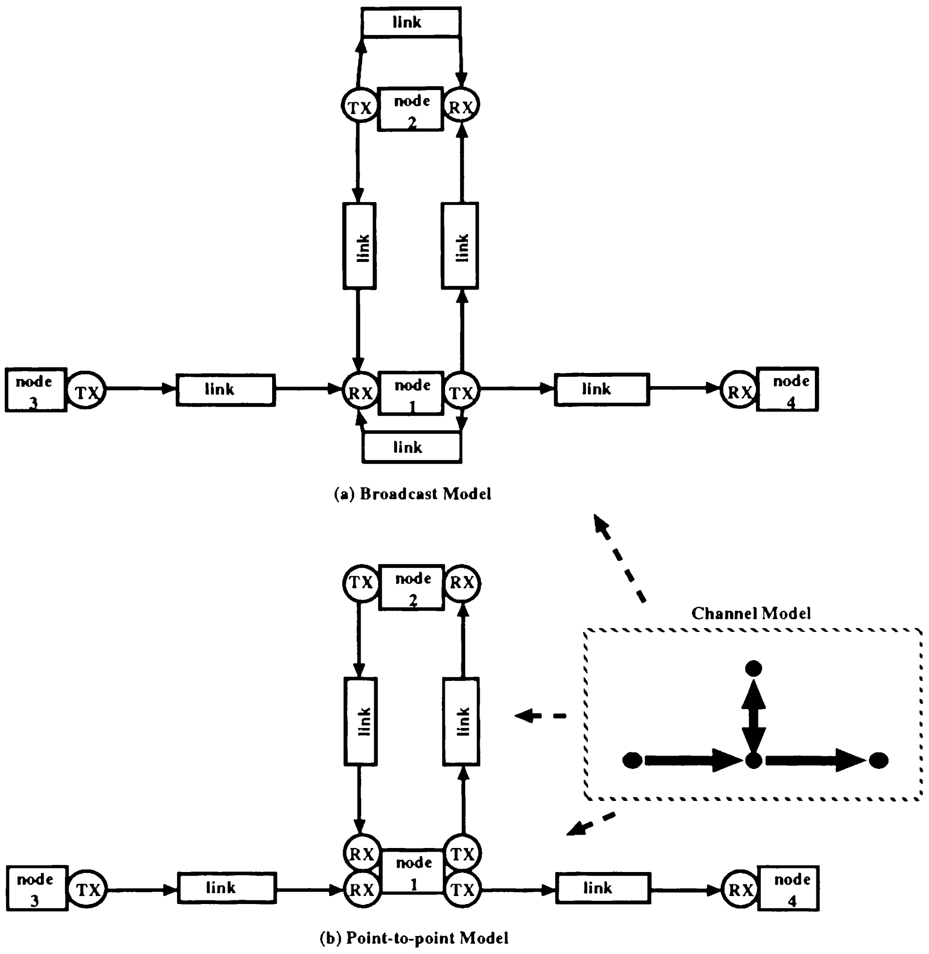

Figure 2: Transmitter/receiver link mapping

lar users and cell base stations, both of which are derived from the abstract type node. They communicate over cellular channels on cellular links using cellular transmitters and receivers, all of these derived from their corresponding broadcast abstract type. Alternatively, a satellite simulation may consist of satellites and ground stations with communication occurring using derived objects specific to satellite communications. In both cases, the network's basic operation remains the same. Namely, nodes transmit packets over broadcast channels which are received by other nodes. However, the way in which this occurs differs greatly between cellular and satellite networks.

Therein lies the key to the simulation's design. The commonality between cellular and satellite communications, namely that of broadcast communication, is identified and placed in a set of abstract "broadcast" objects. This information is then "inherited" by both cellular and satellite communications objects resulting in software reuse. Also, sets of operations particular to broadcast communications are identified in the broadcast abstract classes and are redefined in each of the derived classes to suit their particular requirements. These operations can then be applied uniformly to a heterogeneous list of broadcast objects and the correct operation is automatically applied to each object resulting in software "polymorphism."

3 SIM++ IMPLEMENTATION

In Sim++, a program is broken down into *entities*. Entities are loosely coupled, independently executing objects that interact by scheduling and receiving time-stamped events. These entities may either all reside on a single processor or be distributed over multiple processors. The interacting entities form a distributed, object-oriented, discrete-event simulation whose synchronization is maintained by mechanisms built into Sim++, thus relieving the programmer of the burden of process synchronization. In

Sim++ terminology, when all entities reside on a single processor, the simulation is termed *sequential* whereas a multi-processor simulation is termed *distributed*. Sim++ ensures that, given the same initial conditions, both sequential and distributed simulation runs produce the same results.

During sequential execution, a global simulation clock and event list is maintained and the discrete-event simulation proceeds forward in simulation time according to standard event list processing techniques. During distributed execution, a separate clock and event list is maintained at each processor. The global simulation time, defined to be the minimum of the local clocks, is not known to the entities. The entities execute events from the local event list optimistically under the assumption that their local clock contains the global simulation time. Sim++ saves the state of each entity at each event from the local time back to the global simulation time. Occasionally, an event is received at a processor which is scheduled to occur *before* the processor's local time. In these cases, the states of all entities at that processor must be rolled back to their states just prior to the scheduled time of the event and before local forward progress may be resumed. The rollback may require sending "anti-events" to cancel events which had been sent during the computation which was rolled back. This synchronization mechanism involving state saving, rollback, and anti-messages, is known as "Time-warp" and is transparent to the simulation. This transparency permits identical source code for both sequential and distributed execution modes.

Sim++ designates two types of abstract entities: *sim_entity* and *sim_interface_entity*. An entity derived from sim_entity may undergo rollback whereas an entity derived from sim_interface_entity always executes at the global simulation time and hence, never undergoes rollback.

Sim++ simulations consist of two phases: *initialization* and *execution*. During the initialization phase, entities are created and assigned to processors and global memory is initialized. Once completed, the execution phase begins. During the execution phase, global memory may *not* be modified and the mapping of entities to processors is frozen.

The testbed contains three types of abstract entities: *nodes*, *simulators*, and *monitors*. Both nodes and simulators are derived from sim_entity whereas monitors are derived from sim_interface_entity.

During the initialization phase, the testbed utilizes a YACC-based parser to read in a simulation file. This simulation file specifies either (i) an entire network simulation consisting of nodes, simulators, and (possibly) monitors and all the information required to perform a simulation or (ii) a monitor used to graphically build a network simulation.

3.1 Simulators

Simulators manage communication channels; each may manage one or more channels. There may be multiple simulators per simulation, each managing a different set of channels. A communication channel consists of links, all of which must be the same type. The simulator contains the global topological knowledge of each channel it's managing. This is essential for modeling broadcast channels or for modeling point-to-point channels that use centralized link control. Consequently, the simulator controls the status of each link in those channels.

All simulators must be derived from the abstract base class *com_sim*, itself having been derived from *sim_entity*. The class *com_sim* contains an array of pointers to *channels*, each non-NULL entry indicating a channel is present. The *com_sim* also holds an array containing topological information regarding those nodes that either (i) are permanently connected to a link in one of the simulator's channel (point-to-point link) or (ii) may possibly become connected to a link in such a channel (broadcast link). An example of the latter case would be a cellular user node which has a receiver tuned to a cellular channel being managed by a simulator, but is not currently in range of a transmitter. The simulator must track the position of the node to know when it comes in range of a transmitter.

Channels come in two abstract varieties: broadcast (*bcast_chan*) or point-to-point (*ptop_chan*), the major difference being the virtual operations defined for each channel. The type of channel controlled by the simulator depends on the simulator's type. For example, a cellular simulator would control cellular channels while a satellite simulator would manage satellite channels. Nevertheless, both the broadcast channels types are derived from *bcast_chan*, which is derived from *channel* and thus share the same underlying data structure shown in Fig. 3. The data structure for each *channel i* contains two lists, a *xmtr_id* list and a *rcvr_id* list. Each *xmtr_id* and *rcvr_id* identify a transmitter or receiver on a node which is associated with a link in channel *i*. These "ids" are indexes into the simulator's node topological information array shown in Fig. 4.. Each node contains a *xmtr* array and a *rcvr* array. Each *xmtr* associates with it a list of *rcvr_targs*, each of which represents a receiver on a node also tuned to channel *i* which is in range of the *xmtr*. Each *rcvr_targ* contains a pointer to a *link* object over which the receiver may be reached. Sim-

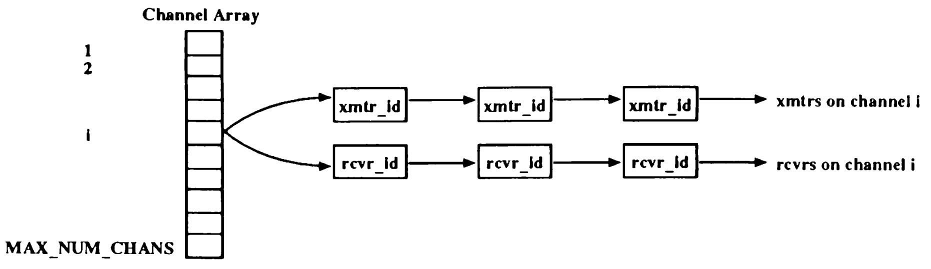

Figure 3: Channel data structure

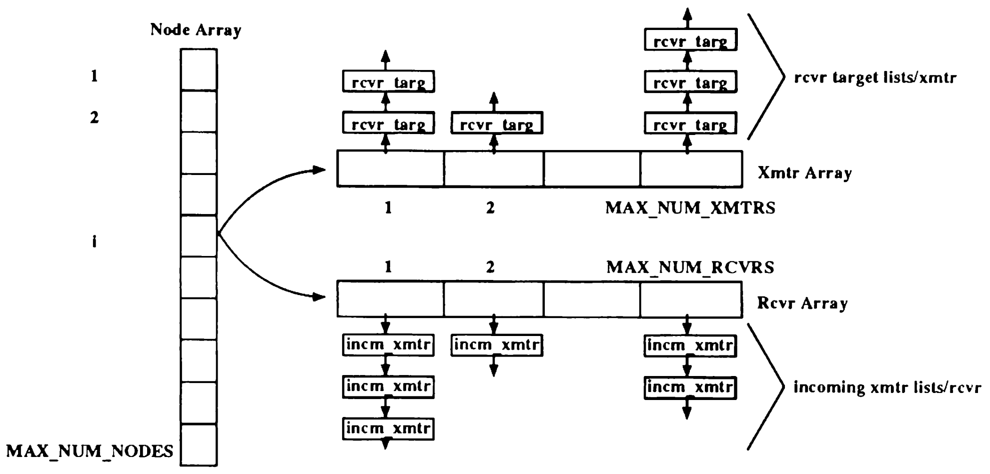

Figure 4: Node topology information data structure

ilarly, a *rcvr* identifies a receiver on a node which is tuned to channel *i* and associates with it a list of *incm_xmtrs*, each of which represents a transmitter on a node also tuned to channel *i* which is in range of the *rcvr*. In this case, however, the *incm_xmtr* contains only the ID of the incoming transmitter and not a link pointer. This data structure is generic and can be used for all channels. Note, the characteristics of each different channel determine its connectivity and the link type pointed to by each *rcvr_targ*. Portions of the data structure containing broadcast channel information are dynamic and change as the broadcast channel's topology changes. Portions containing point-to-point channel information retain the connectivity fixed initially. However, link parameters and operation may change dynamically if necessary.

3.2 Nodes

Nodes represent the communication points of the network. They do not possess global topological knowledge and must rely on the simulators for accurate topology information (more on this later). Each node must be derived from the abstract base class *osi_node*, itself having been derived from the abstract classes *node* and *sim_entity*. The abstract base class *node* contains information regarding the node's ID, location, and a pointer to a *motion_model* which determines its movement (if mobile).

Each *osi_node* is broken down into the 7-layer OSI architecture of Fig. 1. Each block in the figure represents a pointer to an abstract base class which must be assigned to an object derived from the base class. The particular set of protocols and hardware objects attached to these pointers depends on the type of node and determines how the node functions. Protocol layers may be swapped easily by reassigning pointer values, provided the adjacent layers' interfaces are compatible with the new protocol layer.

Each layer 2 protocol object (derived from *layer_2*) is, by default, associated with a different transmitter/receiver pair. However, the mapping of *layer_2*s to hardware objects may be reconfigured as desired. All transmitters and receivers must be derived from the abstract objects *xmtr* and *rcvr* respectively.

Network traffic is generated in the *layer_7* object by calling the virtual function *gen_msg()*. C++ implements polymorphism through the use of virtual functions. For those unfamiliar with C++, refer to

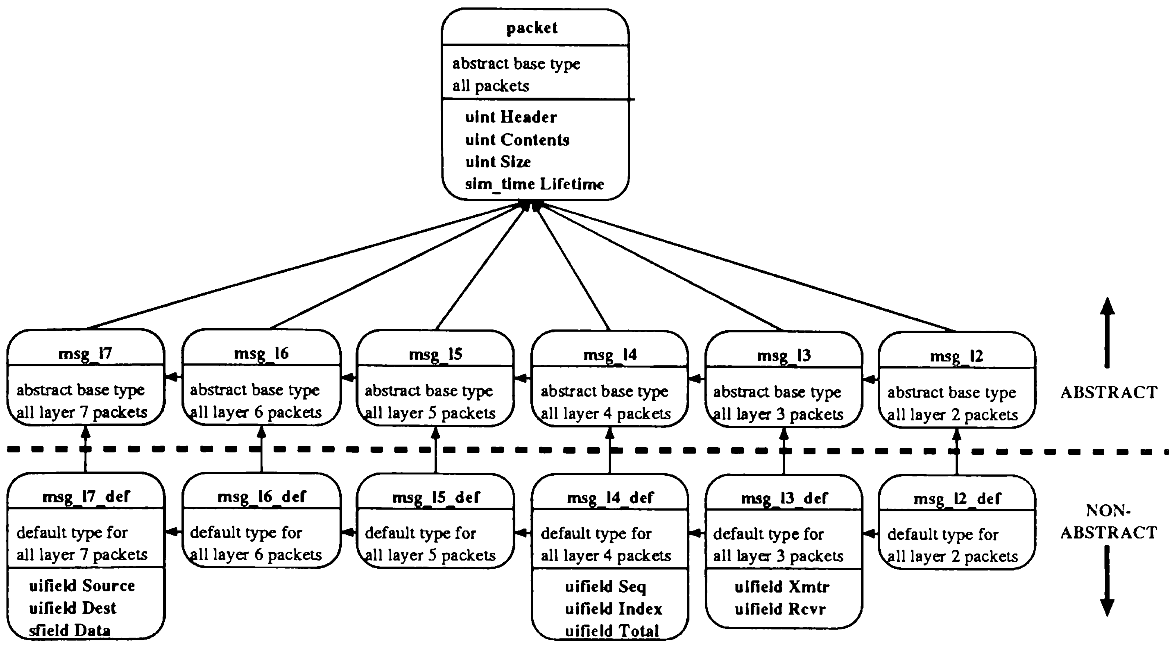
packet
abstract base type all packets
uint Header
uint Contents
uint Size
sim_time Lifetime
msg_l7
abstract base type all layer 7 packets
msg_l6
abstract base type all layer 6 packets
msg_l5
abstract base type all layer 5 packets
msg_l4
abstract base type all layer 4 packets
msg_l3
abstract base type all layer 3 packets
msg_l2
abstract base type all layer 2 packets
ABSTRACT
msg_l7_def
default type for all layer 7 packets
uifield Source
uifield Dest
sfield Data
msg_l6_def
default type for all layer 6 packets
msg_l5_def
default type for all layer 5 packets
msg_l4_def
default type for all layer 4 packets
uifield Seq
uifield Index
uifield Total
msg_l3_def
default type for all layer 3 packets
uifield Xmtr
uifield Rcvr
msg_l2_def
default type for all layer 2 packets
NON-ABSTRACT

Figure 5: Default packet hierarchy

the book by Ellis and Stroustrup (1990). The function *gen_msg()* returns a packet derived from *msg_l7*. If the default layer 7 protocol *layer_7_def* is present, the packet returned would be of type *msg_l7_def*. The default packet hierarchy used by the default layer protocols is shown in Fig. 5. The *msg_l7* is passed to the *layer_6* through the virtual function *pkt_tx(msg_l7*)*. All *layer_6*s must have such a function which is redefined for the each protocol. In a similar fashion, the packet continues down the layers, each layer expecting a packet type derived from the previous layer's abstract packet type, until it reaches a *layer_2* object and is queued for transmission.

Eventually, the packet is transmitted by a *xmtr* towards one or more *rcvrs*. Separate copies of the packet are given to each *link* corresponding to each *rcvr*. The *link* object operates on the packet with its physics and noise objects, each derived from *phys_obj* and *noise_obj* respectively, and delivers the packet to the *rcvr*. At the receiving node, the packet is passed up through the layers. If the node was the intended final destination of the packet, it is delivered to *layer_7*. Otherwise, it is intercepted by the *layer_3* protocol, rerouted, and sent back down for transmission.

3.2.1 Design for Parallel Implementation

Up to now, we have described the major software objects without mention of sequential versus parallel implementation issues or performance. It was stated previously that Sim++ does not require different versions of the source code for each execution mode. However, that does *not* mean that one may ignore parallel processing issues if one hopes to achieve speed-up in a distributed run. *A simulation written without regard for distributed execution will likely run more slowly distributedly than sequentially.*

Earlier, it was stated that the *links* reside in the *channels* in the simulator entities. Therefore, whenever a *xmtr*, which resides on a *node* entity, transmits a *packet*, the *node* must "send" (schedule one or more events) the *packet* to the proper simulator which, in turn gives the *packet* to the *channel* and finally to the *link*. Similarly, after the *link* operates on the *packet*, the simulator must send it to the proper *node* which gives it the the intended *rcvr*.

This model is true to real world situation. Each transmitter blindly transmits packets into the communication channel and they magically reappear at the receiver. However, this implementation requires that every packet transmitted by a node entity is sent through a simulator entity before reaching the intended node entity. This design is not too bad for sequential execution as there is only one event list and each event is processed in the order scheduled. However, packet transmission constitutes the lion's share of the work in a communication network and occurs frequently. Also, a typical network model consists of relatively few channels but many nodes. Therefore, during distributed execution, the simulators would

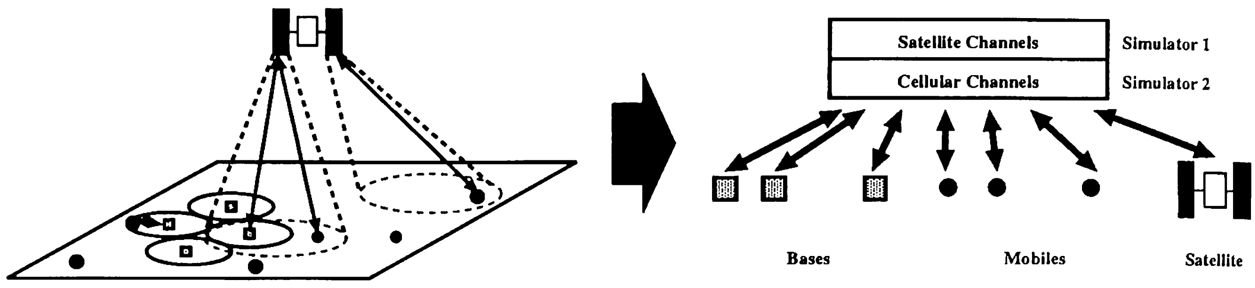

Figure 6: Combined satellite/cellular system

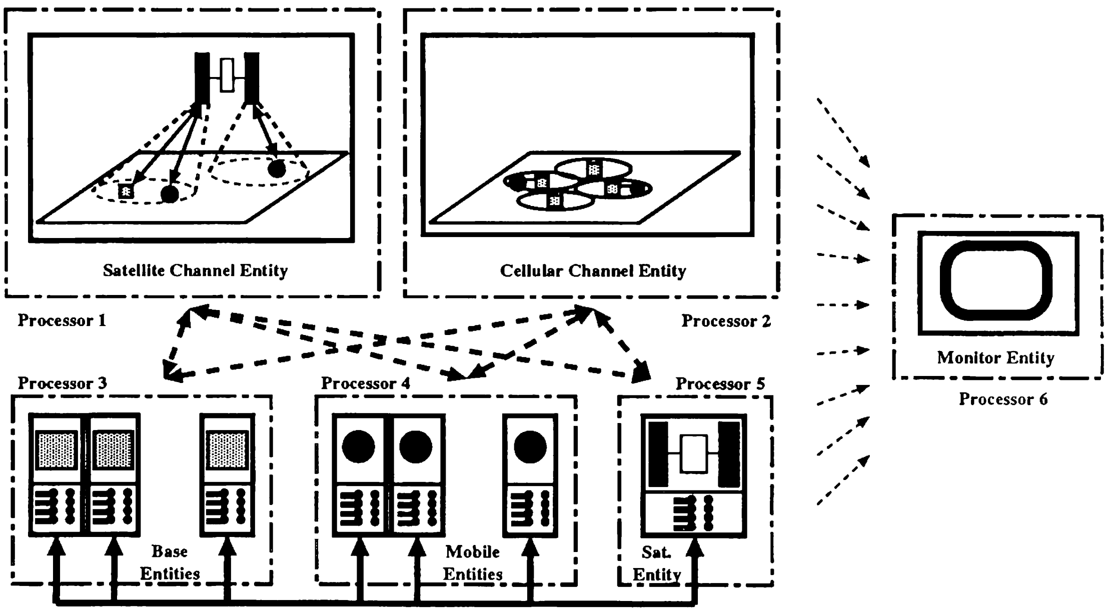

Figure 7: Combined satellite/cellular processor mapping

become bottlenecks, each processing a large number of events, effectively removing a portion of the system's inherent parallelism and lowering achievable speed-up.

The implementation is modified for parallel execution by having each node utilize its own "local channel" or *local_chan*, (recall Fig. 1). The *local_chan* contains the node's local topological knowledge. Associated with each *xmtr* and *rcvr* is a *rcvr_targ* list and *incm_xmtr* list, respectively. These lists are managed by the simulators and kept consistent with those in the *channels*. Whenever an event occurs which changes a link's status, the change is reflected in both the *channel* and *local_chan* in which the link resides.

Hence, when transmitting, the *xmtr* gives the packet to the link in the *local_chan* (via a function call) instead of its copy in the *channel* (via events). Once the link has operated on the packet, it sends (via events) the packet to the proper node entity. By using *local_chans*, we remove half the event scheduling and eliminate the bottleneck problem. The price paid for the performance increase is partly in memory (duplicate *rcvr_targ* and *incm_xmtr* lists) and partly in CPU time (the events necessary to keep the lists consistent). The assumption is that many more events must be sent to transmit packets than to keep the lists consistent due to topological changes. This assumption holds for point-to-point networks and for broadcast networks which don't have rapidly changing topologys. Simulators are not involved in packet transmission and are only concerned with channel topology management.

3.3 Monitors (under development)

Monitors present a 3-dimensional, animated image of network operation for demonstration or protocol debugging. They also provide a graphic, object-oriented, menu-driven user interface from which the user may graphically construct networks from testbed components.

Monitors may execute in two modes: *passive* and *interactive*. In passive mode, user-selected graphical events are received from the nodes and simulators, placed in a graphics event buffer, and displayed as quickly as possible. Nodes and simulators are permitted to execute freely, provided the graphics buffer does not overflow. If this occurs, the nodes and simulators are halted while the display catches up with the simulation. In interactive mode, the nodes and simulators are halted after *each* graphic event thus permitting "single-step" operation essential for protocol debugging. Users are able to change simulation parameters at each step during a run. This ability precludes the use of global data (due to the Sim++ constraint) for many items which are global (such as the simulation run's duration) and requires that a separate copy be kept at each entity.

There may be multiple monitors active at one time, each viewing a different set of network objects and object interactions. Only one monitor may be "interactive" at a given time. Monitors run under X-windows utilize the XGL graphics library.

3.4 Example

Consider the example shown in Fig. 6. A group of mobiles needs to communicate. With cellular coverage, they communicate through the base stations over cellular channels while outside the coverage, they communicate through the satellite and base stations over combined satellite/cellular channels. The mobiles, bases, and satellite are node entities and two simulators are used for channel management.

A possible processor mapping is seen in Fig. 7. The solid communication indicates packet transmission events between the nodes via the local channels. The dashed communication to the channels are topological update events. The dashed events to the monitor (if present) indicate graphical events.

4 SUMMARY

The testbed can be used to model communication networks consisting of mobile and/or immobile nodes communicating over broadcast and/or point-to-point channels. Its object-oriented design permits users to utilize previously developed modules and, when necessary, to derive new modules which are then added to the testbed.

The testbed, implemented in Sim++, and can run either sequentially on a single processor or in parallel on multiple processors. The source code is identical for both modes and is tailored for parallel execution.

A graphical user interface allows users to monitor simulation progress for both demonstration and debugging and permits users to graphically construct network simulations from existing testbed components.

REFERENCES

Ellis, M. and B. Stroustrup. 1990. *The Annotated C++ Reference Manual* Addison-Wesley

Wiener, R. S. and L. J. Pinson. 1988. *An Introduction to Object-Oriented Programming and C++* Addison-Wesley

AUTHOR BIOGRAPHY

M. SCOTT CORSON is a Faculty Research Assistant in the NSF Systems Research Center at the University of Maryland, College Park. He is also a Ph.D. candidate in Electrical Engineering and will join the faculty at the University of Illinois-Chicago in the Fall. His research focuses on distributed algorithms for multiple access and routing in mobile radio networks. While at the Center, he has directed the testbed's development in support of network research efforts underway in the Communications and Signal Processing Laboratory.