Modeling and Simulation of Active Networks*

Dhananjai M. Rao and Philip A. Wilsey
Experimental Computing Laboratory
Dept. of ECECS, PO Box 210030, Cincinnati, OH 45221–0030.
dmadhava@ececs.uc.edu, philip.wilsey@uc.edu

Abstract

Active networking techniques embed computational capabilities into conventional networks thereby massively increasing the complexity and customization of the computations that are performed with a network. In depth studies of these large and complex networks that are still in their nascent stages cannot be effectively performed using analytical methods. Hence, discrete event simulation techniques are the only viable means to study and analyze active networking architectures. Furthermore, customized and flexible tools are required to for the analysis of active networks using simulation. This paper describes an integrated environment for the modeling and parallel simulation of active networks called Active Networks Simulation Environment *(or* ANSE*). ANSE utilizes the Time Warp synchronized kernel of* WARPED *(a general purpose discrete event simulation kernel) to enable parallel simulation of active network models.* ANSE *also includes complete support for the modeling and simulation of active networks based on PLAN (Packet Language for Active Networks). This paper presents the issues involved in the design and development of* ANSE*. The Application Programming Interface (API) of* ANSE *is presented along with the issues involved in utilizing it to develop support for PLAN based active networks. The paper also presents some results obtained from the several experiments conducted to evaluate the effectiveness of* ANSE*. Our studies indicate that* ANSE *provides an effective environment for modeling and simulation of large scale active networks.*

1 Introduction

Techniques to effectively utilize the computational and communication infrastructure of modern networks has lead investigators to develop active networking architectures. In an active network, the nodes constituting the network are capable of performing customizable general purpose processing (or *services*) on the datagrams flowing through them [1, 7]. Active networking techniques enable a massive increase in the complexity and customization of networking services. Active networking techniques also encompass conventional networking architectures and protocols [1]. Unfortunately, traditional analytical methods cannot be effectively used for studying active networks [8]. Hence, empirical methods must be employed. Simulation, discrete event simulation in particular, has proven to be an effective tool to study conventional networks and is also the only method available for analyzing active networks [8, 7, 10].

Model development, verification, and validation plays a critical role in simulation studies. Without sufficient verification and validation (V&V) little confidence can be placed in the results obtained from a simulation [9]. The network models should reflect the size and complexity of actual networks in order to ensure that crucial scalability issues do not dominate during validation of simulation results. Modeling and simulation of large networks can involve extremely long run times on sequential machines. Consequently, sophisticated parallel simulation must be employed to simulate large models in reasonable time frames [7, 10]. To assist in modeling and simulation of active networks, we have developed a tool called the Active Networks Simulation Environment (or ANSE). ANSE provides a hierarchical modeling language that can be used to develop network models represented as a set of interconnected *nodes* (or networking components) developed using ANSE's Application Program Interface (API). The API fully insulates the application modules from the underlying simulation kernel. The API has been used to develop a library of components that can be used to model active networks based on PLAN, a Packet Language for Active Networks. The simulation infrastructure of ANSE utilizes the Time Warp synchronized kernel of WARPED (a general purpose discrete event simulation kernel [4]) to enable parallel simulation of active network models.

This paper presents the issues involved in the design and development of ANSE. Section 2 presents brief description

*Support for this work was provided in part by the Advanced Research Projects Agency under contract DABT63–96–C–0055.

0-7695-1092-2/01 $10.00 © 2001 IEEE

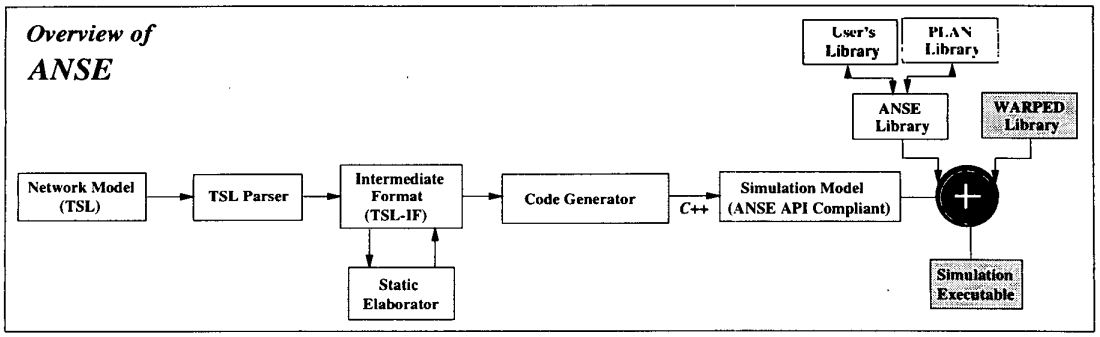


Figure 1. Overview of ANSE

of WARPED, the simulation kernel utilized by ANSE. In Section 3 an overview of ANSE and its various components are presented. The unified modeling front-end and ANSE's API are also presented in this section. The issues involved in the design and development of support for modeling and simulation of active networks based on PLAN is presented in Section 3.5. Section 4 presents results from several experiments conducted using ANSE. Finally, concluding remarks and pointers to future work are presented in Section 5.

2 Background

The parallel simulation capabilities of ANSE have been enabled by developing the framework around a general purpose discrete event simulation engine. Object oriented (OO) techniques have been employed to isolate the various modules of ANSE from the underlying simulation kernel. This design not only provides a desired level of "separation of concerns" but also enables the use of different simulation kernels without modification to the application modules. The current implementation of ANSE utilizes the WARPED simulation kernel [4] to enable sequential and parallel simulation of active network models. WARPED is an Application Program Interface (API) for a general purpose discrete event simulation kernel with different implementations [4]. ANSE utilizes the sequential kernel and the Time Warp based parallel simulation kernel of WARPED.

The parallel simulation implementation in WARPED uses the Time Warp optimistic synchronization strategy [3]. A Time Warp synchronized parallel simulation is organized as a set of asynchronously communicating logical processes (LPs). The LPs communicate between each other by exchanging *virtual time*-stamped event messages [3]. Each LP processes its events maintaining a local virtual time (LVT), changing its state, and generating new events without synchronization concerns to other LPs. Although each LP processes local events in their correct time-stamp order, events are not globally ordered. Causal violations may occur due to the optimistic nature of Time Warp. Causality violations are detected by a LP when it receives an event with time-stamps lower than its LVT (called a *straggler* event). On receiving a straggler, a *rollback* mechanism [3] is invoked to recover from the causality error. The rollback process recovers the LP's state prior to the causal violation, canceling the erroneous output events generated by sending out anti-messages, and re-processing the events in their correct causal order [3]. Each LP maintains a list of state transitions along with lists of input and output events corresponding to each state to enable the recovery process. A periodic garbage collection technique based on Global Virtual Time (GVT) [3] is used to prune the queues by discarding history items that are no longer needed.

The WARPED kernel presents an interface to build logical processes based on Jefferson's original definition [3] of Time Warp [4]. The kernel provides an API to build different LPs with unique definitions of state [4]. The basic functionality for sending and receiving events between LPs using a message passing system is supported by the kernel. In WARPED, LPs are placed into groups called "clusters". LPs on the same cluster communicate with each other without the intervention of the message passing system, which is *faster* than communication through the message system [4]. Although LPs are grouped together into clusters they are not coerced into synchronizing with each other. Control is exchanged between the application and the simulation kernel through cooperative use of function calls.

3 The Active Networks Simulation Environment (ANSE)

ANSE was developed to ease modeling and simulation of active networks. An overview of ANSE is presented in

Figure 1. As shown in Figure 1, the primary input is the topology of the network model to be simulated. The syntax and semantics of the input topology is defined by the Topology Specification Language (TSL). TSL provides simple, yet robust, hierarchical modeling techniques for representing a network as a set of interconnected components/nodes. The components used in the TSL description are developed using ANSE's API. As illustrated in Figure 1, the input network model is parsed into an object-oriented in-memory intermediate format (TSL-IF). An elaboration module is used to elaborate or "flatten" a hierarchical network model. Elaboration is performed to ease further processing of the network model. TSL-IF is used to represent the elaborated network model. As shown in Figure 1, a back-end code-generator utilizes the elaborated network model to generate ANSE API compliant (C++) code. The generated code is compiled and linked with necessary libraries such as the WARPED library, ANSE library, PLAN library, and other user-defined libraries (as the case maybe) to obtain the final executable. The final executable performs the actual simulation when it is run. A detailed description of the various modules constituting ANSE is presented in the following subsections.

3.1 Topology Specification Language (TSL)

The primary input to ANSE (as shown in Figure 1) is the topology of the network to be simulated is provided to the environment in Topology Specification Language (TSL) [5] syntax. The Backus Normal Form (BNF) of TSL grammar is shown in Figure 2. As specified by the grammar, a TSL specification consists of a set of interconnected topology specifications. Each topology specification consists of three main sections, namely; (i) the *object definition section* that contains the details of the modules that need to be used to simulate the topology; (ii) the *object instantiation section* that specifies the various nodes constituting the topology; and (iii) the *netlist section* that defines the interconnectivity between the various instantiated nodes. Figure 3.1 presents a network model along with the corresponding TSL description. An optional *label* may be associated with each topology. The label may be used as an object definition in subsequent topology specifications to nest a topology within another. In other words, the labels, when used to instantiate an object, result in the complete topology associated with the label to be embedded within the instantiating topology. Figure 3.1 presents the TSL source code to model a larger network using hierarchical constructs. As illustrated by the figure, the model of the network is specified by interconnecting three instances of the network model shown in Figure 3.1. Using this technique, a simple sub-network consisting of merely ten nodes can be recursively used to construct a network with six levels of hierarchy to specify a network

```
design_file ::= include_list tsl_design_topology |
    tsl_design_topology
include_list ::= include_clause | include_clause include_list
include_clause ::= include " file_name ";
file_name ::= identifier | identifier . identifier
tsl_network ::= tsl_topology | label tsl_topology |
    tsl_topology tsl_network | label tsl_topology tsl_network
tsl_topology ::= { object_definition_section }
    { object_instantiation_section } { net_list_section }
label ::= identifier
object_definition_section ::= object_definition |
    object_definition object_definition_section
object_definition ::= object_name : url optional_parameter
object_name ::= identifier
url ::= host_name : port_number . factory
optional_parameter ::= parameter ; | ;
parameter ::= " string " | ""
factory ::= identifier | identifier . factory
port_number ::= number
object_instantiation_section ::= object_instantiation |
    object_instantiation object_instantiation_section
object_instantiation ::=
    object_instance : object_name optional_parameter |
    object_instance : object_name number optional_parameter
    | object_instance : label
object_instance ::= identifier
net_list_section ::= net_list | net_list net_list_section
net_list ::= object_instance : instance_list ;
instance_list := object_instance | object_instance instance_list
identifier ::= start_char any_char
start_char ::= [a - z, A - Z]
any_char ::= [a - z, A - Z, 0 - 9, _ ]
string ::= string_char | string_char string
string_char ::= [ ]
number ::= [0 - 9]
```

Figure 2. BNF of TSL grammar

with million (10^6) nodes [7, 6]. ANSE also include a tool for translating Georgia Tech Internet Topology Models (GT-ITM) [11] models into equivalent TSL descriptions. The translation tool and GT-ITM can be used to automatically generate network topologies in TSL.

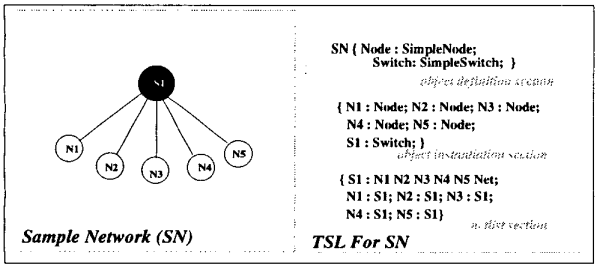
3.2 TSL Parser

The input topology configuration is parsed using a parser into an OO TSL Intermediate Format (TSL-IF). The TSL parser is generated using the Purdue Compiler Construction Tool Set (PCCTS) [7]. TSL-IF forms the primary input to the other modules of ANSE. TSL-IF is designed to provide efficient access to related data from the various TSL sections [6]. In conjunction with the parser, TSL-IF

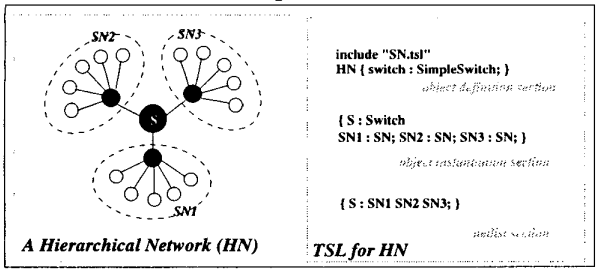
is also implemented in C++. The IF consists of a set of cross-referenced classes, each class representing a particular grammar entity. The IF is composed by filling in the references in the various C++ classes generated by the parser with appropriate values. Since composition is achieved via base class references, each node can refer to another node or even a sub-network. This provides an efficient data structure for representing and analyzing hierarchical networks [7, 6].

3.2.1 Static Elaborator

Hierarchical constructs provide convenient techniques to specific large networks by reusing the specification for smaller sub-networks [7, 6]. However, the hierarchical constructs have to be elaborated or "flattened" prior to simulation [7, 6]. Elaboration is the process in which each hierarchical level is broken down to its constituting components. The basic steps involved in elaborating a hierarchical specification are shown in Figure 4. As illustrated in the figure, the elaborator starts with an user-specified topology and recursively traverses the various sub-topologies in the model and creates new instances of the sub-topologies and the objects. Elaboration of sub-topologies is done before they are imploded into the enclosing topology. Imploding hierarchies involves inclusion of all necessary object definitions, object instantiations, and corresponding data structures. Elaboration may be done *statically* or at *runtime*. Static elaboration occurs prior to code-generation while runtime elaboration occurs just before simulation commences, when



(1) Simple network



(2) Hierarchal network

Figure 3. Example Network models and TSL

step 1: Initialize elaborator
1. Initialize new symbol table and IF
2. Search input IF and locate node corresponding to user specified top level topology
3. Call elaboration (**step 2**) with new topology

step 2: Elaboration subroutine (*parameter* `topology`)
1. Process the list of netlists specified in the `topology`. If the node is an object instantiation perform **step 3**. If the node is a topology label perform **step 4**.

step 3: Elaborate object instantiation
1. Create new instance of the object instantiation with mangled labels.
2. Create new object definition for the new instance with mangled labels and add to new symbol table, if necessary.
3. Add new object instantiation to the new topology and update netlist entry.

step 4: Elaborate `sub-topology`
1. Instantiate temporary symbol table and IF
2. Recursively call elaboration with the `sub-topology`
3. Implode new IF to the new topology

Figure 4. Phases in elaborating a TSL design

the generated code is executed. In ANSE, static elaboration is performed with the TSL-IF generated by the parser and the elaborated topology is also represented in TSL-IF (as shown in Figure 1).

3.3 Code Generator

As shown in Figure 1, the back-end code-generator utilizes the elaborated TSL-IF to generate a simulatable model from the given TSL description. The generated code in in C++ in concordance with all the other components of ANSE. The OO nature of TSL-IF has been exploited in the development of the code-generator. The generated code is compliant with the API of ANSE. A model developer can directly develop the network model (compliant with ANSE's API) and bypass these stages. However, the complexity involved in model development would be considerably higher. The back-end code-generator can be replaced with a different code-generator in order to re-target the generated code for different frameworks.

3.4 ANSE API and Library

ANSE presents an interface to the application developer for modeling a network as set of communicating logical processes (LPs). The LPs are modeled as entities which send and receive events to and from each other, and act

on these events by applying them to their internal state. Figure 5 shows the Universal Markup Language (UML) diagram for the core classes that constitute the API. As illustrated in the figure, the `NetworkNode` class forms the parent class for all the networking components in the system. The `ActiveNode` and `PLANNode` are derived from this class. The `NetworkNode` class is used to model conventional networking components while the `ActiveNode` is used to model active components. The `NetworkNode` also provides methods for accessing routing tables and supports primitive domain name services (DNS). The state (`NetworkNodeState` and `ActiveNodeState`) and packet (`Packet` and `ActivePacket`) classes corresponding to the LP hierarchy are also shown in Figure 5. The state classes are used to encapsulate the state information associated with each node/component. The state information is used by WARPED (the underlying simulation kernel) to enable *rollbacks* [3] and recover from causal violations that could occur in Time Warp based simulations [4]. The `Packet` (and derived) class is used for all communications between the nodes constituting a network model. The packets in turn represent the discrete events in the simulation. The API has been developed in C++ and the object oriented features of the language have been exploited to ensure it is simple and yet robust. The API plays a critical role in insulating the model from the underlying simulation kernel. The interface has been carefully designed to provide sufficient flexibility to the application developer and enable optimal system performance. Further details on the API are available in the literature [7].

3.5 PLAN library

As illustrated in Figure 5, ANSE's API has been used to develop a library for modeling and simulating active networks based on PLAN, a Packet Language for Active Networks [2]. PLAN is a simple, functional programming language based on a subset of ML with some added primitives to express remote evaluation [2]. In a PLAN based active network, the active packets can contain a PLAN program

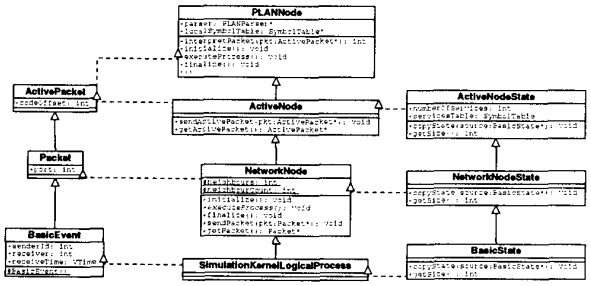


Figure 5. Core classes of ANSE API

that can be used to customize the active network to provide different networking services. The PLAN library provides a `PLANNode` that is capable of parsing and interpreting PLAN packets. A PLAN parser, constructed using PCCTS, is used to parse incoming PLAN packets into an OO intermediate format (IF). The IF is fed to a PLAN interpreter that executes the program contained in the packet. The interpreter supports all PLAN constructs including recursive function calls, exceptions, and forwarding of any PLAN packets generated during interpretation. The PLAN library also contains a `PacketInjector` component that can be used to inject PLAN packets into the simulated network. The `PacketInjector` can be used to inject a PLAN program from a given file or obtain the PLAN program interactively from the user. The `PacketInjector` can be driven using a variety of traffic generators based on random number generators available as a part of the ANSE library. The random number generators generate traffic based on mathematical distributions (such as normal or constant delay distributions, Poisson distribution, and Pareto distributions). The library also contained components for modeling different types of communication links with different parameters (such as transmission delay and packet loss ratio).

The runtime structure of a typical PLAN based active network is shown in Figure 6. As illustrated in the figure, a single instance of the PLAN parser and interpreter are shared by the different `PLANNodes`. The design helps to minimize the overall resource requirements (memory in particular) of the simulations; thereby enabling simulation of larger networks using available hardware resources. However, in parallel simulations, a single instance of the PLAN parser and interpreter are used in each cluster. This approach is a tradeoff between the overall memory requirements of the simulation versus simulation overheads (such as communication and concurrency). It must be noted that concurrent demands/usage of the parser and interpreter never arises because execution of events on a cluster proceeds in a serial order. In other words, although the WARPED clusters and the LPs operate asynchronously with each other, the events on a given cluster are executed serially. Hence, in a given cluster, only one event can be active at a time and the PLAN parser and interpreter are assigned (or reserved) for use by that event. Since parsing of PLAN packets their interpretation are two distinct and independent stages, they can be cascaded or pipelined (*i.e.,* when a previous packet is being interpreted the next packet can be parsed) to improve performance. Such a design would be of considerable benefit in shared memory multiprocessor (SMP) platforms. Any dependencies or inconsistencies that could arise due to asynchronous pipelining can be resolved by directly utilizing the optimistic simulation infrastructure. In other words, if inconsistencies arise then the simulation would get *rolledback* and the events would get reprocessed

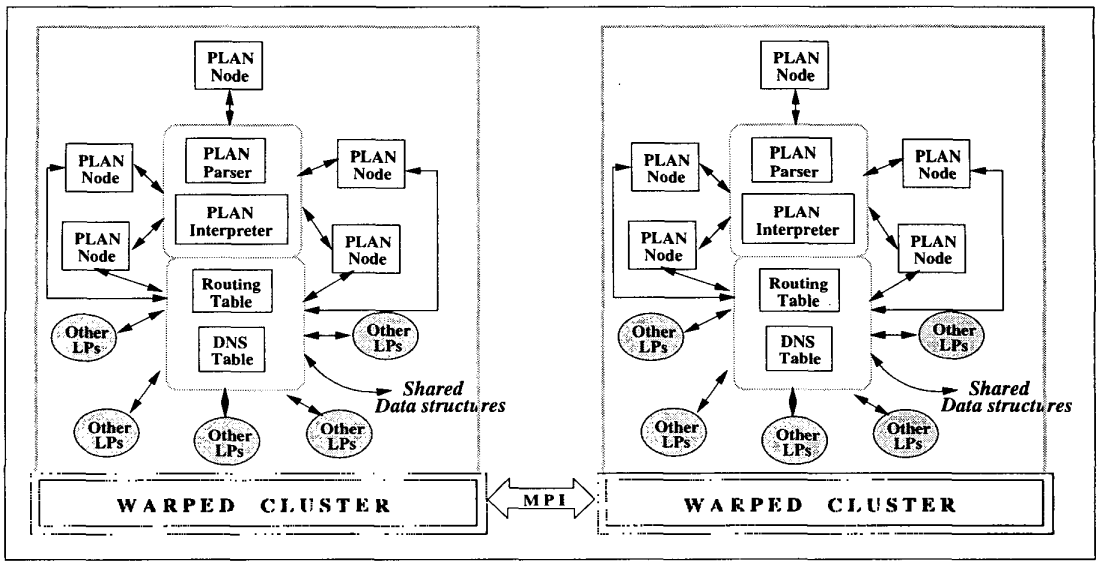


Figure 6. Runtime structure

Model	Hierarchies	Number of Processes			
		PLAN	Packet Injectors	Others	Total
Model1	1	9	9	41	59
Model2	2	19	19	127	165
Model3	2	26	26	193	245
Model4	3	55	55	375	485
Model5	3	110	100	781	991

Table 1. Characteristics of the models

in the correct causal order. However, such a design was not adopted in the current implementation because other simulation techniques (such as sequential simulation and conservative simulations) may not be capable of supporting such a design.

As shown in Figure 6, the routing tables and DNS tables (built and maintained by `NetworkNodes`) are also shared between the various nodes constituting the simulation. This design also helps to reduce the overall memory requirements of the simulations. As explained earlier, concurrent access to these data structures does not arise. Hence, complex locking mechanisms/semaphores are not necessary to ensure their consistency/coherence. The routing and DNS tables are replicated at each cluster (in a parallel simulation) to minimize simulation overheads. The runtime modules of ANSE (present in the ANSE library) assist in constructing the tables by providing necessary information about the network model being simulated. Although, the LPs are partitioned onto different clusters, the necessary information (such as name of the nodes along with the interconnectivity data) related to all the nodes are extracted and filled into the tables. In the current implementation of ANSE, the LPs are equally divided amongst the clusters used in a simulation *i.e.,* each cluster has (almost) an equal number of LPs. ANSE's API also includes interfaces for implementing other partitioning algorithms. It must be noted that, partitioning, assignment of nodes to clusters, and parallel simulation is completely transparent to the application modules. The applications (and generated code) does not change based on the number of clusters or the underlying synchronization technique used in the simulations.

4 Experiments

The experiments conducted to evaluate the performance of ANSE and the results obtained from the experiments are presented in this section. Table 1 tabulates the characteristics of the models used to conduct the experiments. The network models were described in TSL and utilized the various components available in the PLAN and ANSE libraries. The network models consisted of a set of interconnected PLAN nodes. The larger models such as `Model3`, `Model4`, `Model5` were constructed using the hierarchical modeling constructs supported by TSL. The number of PLAN nodes in each model is shown in Table 1. The other components of the model such as traffic generators, packet injectors, and links are grouped together and tabulated in Table 1 (under the "Others" column). A route tracing PLAN program [2] was run on the simulated network model. The route tracing PLAN packets hop from one node to another (as they get interpreted by each PLAN node in the

simulated network) and at each node they generate two new PLAN packets. One packet carries the information about the current hop back to the source node (*i.e.,* the node at which the route tracing for that particular packet began). The other packet proceeds forward to trace the route until the destination node is reached. The destination node on each packet was randomly chosen from the set of nodes participating in the simulation. The `PacketInjector` (described earlier) was used to inject the PLAN packets into the simulated network. Each `PacketInjector` was programmed to generate 500 requests (*i.e.,* trace route to 500 randomly chosen PLAN nodes). The links interconnecting the nodes were configured (through suitable parameters in the TSL description) to have zero losses *i.e.,* no packets get lost (in other words, a basic TCP/IP type of connectivity was assumed).

The graphs in Figure 7 present the time taken for performing the different phases of model generation such as parsing, elaboration, code-generation, and compiling the generated code. The experiments were conducted on a Linux workstation consisting of dual Pentium II (300 Mhz) processors with 128 MB of main memory. The timings were obtained using the standard Unix `time` command. The times plotted in the graphs are the average values computed from 10 simulation runs. As illustrated by the graphs shown in Figure 7 the time overall time for generating a network model scales almost linearly with respect to the total number of objects (or LPs) constituting a network model. This data suggests illustrates the scalability of the modeling and simulation infrastructure supported by ANSE. It also indicates that ANSE will be capable of generating large network models in reasonable time frames.

The parallel simulation experiments were conducted using a network of workstations. Each workstation consisting of two Pentium II (300 MHz) processors in shared memory configuration. Each workstation had 128 MB of main memory (RAM) and were running Linux. The workstations were interconnected by fast Ethernet. The parallel simulations were conducted using 1 to 16 WARPED clusters. The results obtained from the various experiments are presented by the graphs in Figure 8. The timing information for the various simulations were obtained using the standard Unix `time` command. The simulation times plotted in the graphs are the average values computed from 10 simulation runs. The timings obtained from the simulations conducted using the sequential kernel available with WARPED are also plotted in the graphs. The sequential simulator was configured for its most optimal configuration (including using a splay tree data structure for maintaining event lists).

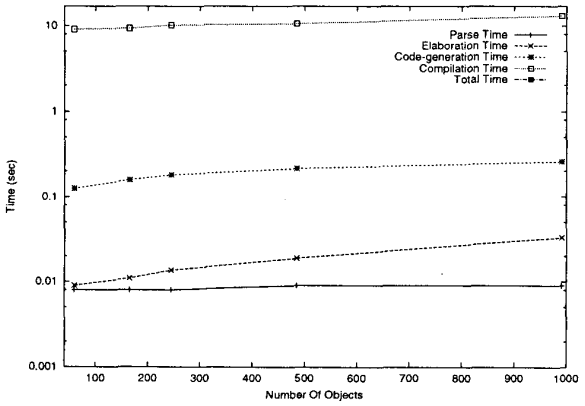
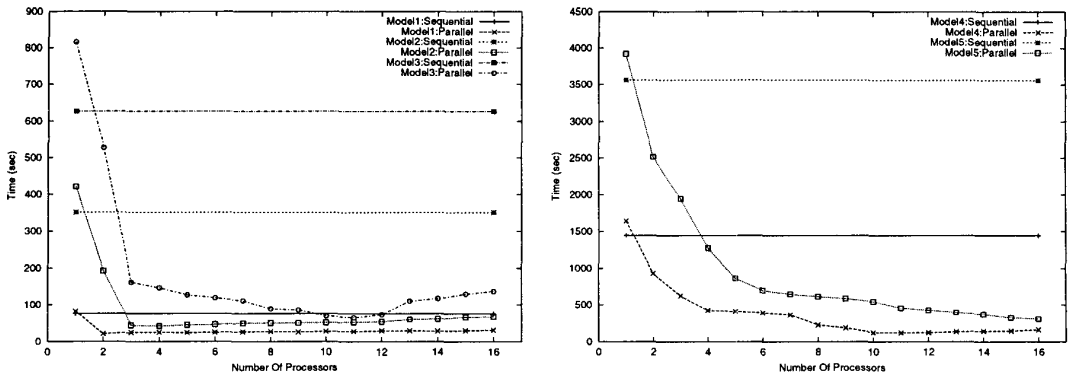


Figure 7. Time for model generation

As illustrated by the graphs in Figure 8, parallel simulation provides considerable improvements in performance for even medium sized network models. For example, parallel simulation using 16 processors provides an order of magnitude improvement in performance when compared to a sequential simulation. The primary factor for the pronounced improvement in performance is the high event granularity of the active packets which need to be parsed and interpreted. As the number of processors used in the simulation are increased, the computational load gets distributed across the parallel processors which in turn reduces the overall simulation time. However, as illustrated by Figure 8(a), for small models, the performance deteriorates as the number of processors are increased. This is because the smaller models do not have sufficient concurrency and load to utilize all the parallel processors. Hence, the overheads of parallel simulations out weigh the gains accrued by increasing the number of processors. The results also demonstrate the scalability of the parallel simulation framework. The experiments highlight that considerable improvements in performance of the simulations can be achieved by employing parallel simulation techniques. The experiments also illustrate the overall effectiveness of ANSE for modeling and simulation of active networks.

5 Conclusions

The issues involved in the design and implementation of an Active Networks Simulation Environment (ANSE) were presented in this paper. The experiences gained during the development of ANSE also highlight a number of issues on different aspects of active network modeling and simulation. Our experiences indicate that it is better to have to have a simple, yet flexible, language such as TSL, for modeling network topologies. It is useful to have a clear delineation between the languages for developing the software modules for networking components and network modeling languages. For example, TSL and ANSE can also be used to enable simulation of conventional networks. The flexibility and general purpose design of ANSE can be utilized to en-



(a) Small Models **(b) Large Models**

Figure 8. Comparison between sequential and parallel simulation times

able inter-operability between different type of models and even different simulators. An experimental evaluation of ANSE was presented in the paper. The experiments demonstrate that considerable improvements in performance of the active network simulations can be achieved by employing parallel simulation techniques. The experiments in conjunction with the diverse set of issues addressed by ANSE highlight the effectiveness of the active networks simulation environment provided by ANSE.

References

[1] DAVID L. TENNENHOUSE, J. M. S., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A survey of active network research. *IEEE Communications Magazine 35*, 1 (Jan. 1997), 80–86.

[2] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTHER, C. A., AND NETTLES, S. PLAN: A Packet Language for Active Networks. In *In Proceedings of the Third ACM International Conference on Functional Programming Languages (SIGPLAN'98)* (May 1998), pp. 86–93.

[3] JEFFERSON, D. Virtual time. *ACM Transactions on Programming Languages and Systems 7*, 3 (July 1985), 405–425.

[4] RADHAKRISHNAN, R., MARTIN, D. E., CHETLUR, M., RAO, D. M., AND WILSEY, P. A. An Object-Oriented Time Warp Simulation Kernel. In *Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, D. Caromel, R. R. Oldehoeft, and M. Tholburn, Eds., vol. LNCS 1505. Springer-Verlag, Dec. 1998, pp. 13–23.

[5] RAO, D. M., RADHAKRISHNAN, R., AND WILSEY, P. A. FWNS: A Framework for Web-based Network Simulation. In *1999 International Conference On Web-Based Modelling & Simulation (WebSim'99)* (Jan. 1999), A. G. Bruzzone, A. Uhrmacher, and E. H. Page, Eds., vol. 31, Society for Computer Simulation, pp. 9–14.

[6] RAO, D. M., AND WILSEY, P. A. An object-oriented framework for parallel simulation of ultra-large communication networks. In *Proceedings of the the Third International Symposium on Computi ng in Object-Oriented Parallel Environments* (Nov. 1999).

[7] RAO, D. M., AND WILSEY, P. A. Simulation of ultra-large communication networks. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Oct. 1999), pp. 112–119.

[8] RILEY, G. F., FUJIMOTO, R. M., AND AMMAR, M. H. A generic framework for parallelization of network simulations. In *Proceedings of the Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (Oct. 1999), pp. 128–135.

[9] ROBINSON, S. Simulation model verficiation and validation: Increasing the users' confidence. In *Proceedings of the 1997 Winter Simulation Conference* (Dec. 1997).

[10] SWAMINATHAN, K., RADHAKRISHNAN, R., WILSEY, P. A., AND ALEXANDER, P. Large scale active networks simulation. In *International Workshop on Applied Parallel Computing (PARA98)*, B. Kagstrom, J. Dongarra, E. Elmroth, and J. Wasniewski, Eds., vol. LNCS 1541. Springer-Verlag, June 1998, pp. 537–542.

[11] ZEGURA, E., CALVERT, K., AND BHATTACHARJEE, S. How to model an internetwork. In *Proceedings of IEEE INFOCOM* (Apr. 1996), pp. 594–602.