PARALLEL SIMULATION OF TCP/IP USING TeD

Brian J. Premore
David M. Nicol

Department of Computer Science
6211 Sudikoff Laboratory
Dartmouth College
Hanover, New Hampshire 03755-3510, U.S.A.

ABSTRACT

We are developing a parallel simulation framework for large-scale network simulation. An important component of this work is the development of a TCP/IP model library. TCP/IP experts at LBNL have already developed the *ns* simulator for investigation of protocol variants. *ns* is a flexible and easy-to-use tool, but its construction thwarts high performance simulation of large scale networks. In order to model TCP/IP as accurately as does *ns* but still provide high performance, we are transforming *ns* source code into the Telecommunications Description Language (TeD), a tool that brings automated parallelization to network simulation. This paper describes the issues that arose in the course of this transformation.

1 INTRODUCTION

The explosion of network use and interest in networks creates a need to simulate very large scale networks. Applications include the study of more efficient ways to handle the ever-increasing loads of the Internet, multi-media over the Internet, multi-casting and routing algorithms, etc. With these demands, so arises a need for faster simulators to handle the larger loads. Towards this end, the Telecommunications Description language (TeD) (Perumalla, Ogielski, and Fujimoto 1996; Perumalla and Fujimoto 1996a) is a system under development at Georgia Tech to provide a simulation framework for large-scale network simulation. TeD is modular and object-oriented. Its design reflects an overriding goal that TeD submodels be reusable, that they support a library approach to building up large complex systems, and models be extensible. TeD models are run in parallel, automatically, by being transformed into functionally equivalent GTW (Georgia Tech Time Warp) (Das et al 1994; Perumalla and Fujimoto 1996b) code.

We are part of an NSF-funded project to develop technology to support large-scale network simulation. A key ingredient to many networks is the TCP/IP protocol family. Our goal is to develop accurate TCP/IP models and make them available as TeD libraries so that higher level protocol simulations might be built modularly, drawing upon the TCP/IP model code. Clearly then it is important to use an accepted and accurate model for TCP/IP behavior. To ensure this we turned to the *ns* simulator, developed at Lawrence Berkeley National Lab by noted TCP/IP experts. This paper describes the issues that arose in the course of transforming *ns* source code into functionally equivalent TeD code. The transformation is of some interest, because in its focus on automated parallelization, TeD imposes modeling constraints that affect how one can build TeD models.

ns is built using a combination of Tcl and C++. The user's interface to *ns* is through a Tcl script that defines network topology, protocol variants, traffic type, traffic source and sinks, etc. Behind Tcl, C++ objects implement the network behavior and execute the discrete-event simulation. An *ns* user essentially describes an experiment or set of experiments. If one wished to implement a new variant on TCP/IP, one would need a certain facility with defining new objects "under-the-covers" and knowledge of how to integrate them into the *ns* framework. The heart of *ns* is event-oriented simulation (as opposed to process-oriented simulation).

TeD is best viewed as a combination of a special TeD "meta-language" that provides simulation specific constructs, and C++. The meta-language is the framework that describes the simulation topology and the experimental framework. Ordinary C++ code—some of calling meta-language macros—is used to describe object behavior. Given a sufficient set of library submodels, one can build a TeD model at basically the same level of abstraction as one builds an *ns* model. In TeD's case it is mostly a matter of configuration and selection of pre-defined behaviors for pre-defined network objects. Because TeD, like *ns*, is object-oriented, converting from *ns* into TeD is a reasonable way to approach the goal of fast, accurate simulations. There is a fundamental difference though. While the *ns* engine is event-oriented, the TeD world-view is process-oriented (with restrictions). This difference and the restrictions TeD places on model expression create the key technical challenges of this transformation. Nevertheless, we believe that TeD TCP/IP models can be built which behave identically (statistically) to equivalent *ns* models, but, because of TeD's parallel execution, achieve significant speedups.

2- AN OVERVIEW OF SIMULATION USING *ns*

ns extends Tcl with simulation specific objects and functions. A network topology is defined by using `nodes` and `links` as building blocks. `nodes` are connected by `links` and may contain `agents`, which are responsible for sending and receiving packets. In the process of creating these structures, the user can describe many of their characteristics. For example, two characteristics of `links` are 'bandwidth' and 'packet drop policy.' The user can also schedule changes to occur at specified times during the simulation. For instance, one might simulate a node in the network going down at time 100.0. Finally, the user can set up customized traces to gather information about the simulation. See Figure 1 for an example of a typical *ns* script.

3- AN OVERVIEW OF MODELING IN TeD

A model in TeD consists of *entities* which communicate using *events* and *channels*. Each entity object represents some physical network object, and events are messages between entities that encapsulate information and stimulate simulation behavior. Messages are passed through channels. Both the sending and receiving entity declare a channel variable (which may have different names) and a configuration statement executed at initialization *maps* them together, creating a link between those entities. Channels of this type are called *external*. The entity itself in TeD just specifies the interface of that object to the rest of the world—the instantiation of behavior that uses that interface is separate, being contained in an *architecture*. We will illustrate these concepts with a TCP model. Top level entities in this model resemble *ns* primitives, e.g., NODEs and LINKs. NODEs may have any number of channels, while LINKs have exactly two. These channels are mapped to define the (static) network topology. The object-oriented structure of TeD also allows for entities to contain components which are themselves entities. For example, NODE entities might have an array of AGENT subentities which do the actual implementation of TCP.

```
# create two nodes
set n0 [ns node]
set n1 [ns node]

# connect them with a 1.5Mb link with a transmission
# delay of 10ms using FIFO drop-tail queueing
set link0 [ns link $n0 $n1 drop-tail]
$link0 set bandwidth 1500000
$link0 set delay 0.100

# links in ns are unidirectional, so do both directions
set link1 [ns link $n1 $n0 drop-tail]
$link1 set bandwidth 1500000
$link1 set delay 0.100

# set up BSD Tahoe TCP connection in one direction only
set src1 [ns agent tcp $n0]
set snk1 [ns agent tcp-sink $n1]
ns_connect $src1 $snk1
$src1 set class 0

# Create an ftp source at the source node
set ftp1 [$src1 source ftp]

# Start up the ftp at the time 0
ns at 0.0 "$ftp1 start"

# run the simulation for 10 simulated seconds
ns at 10.0 "exit 0"
ns run
```

Figure 1: Sample *ns* Code Defining a Network with Two Connected Nodes, with an Established FTP Connection Between Them

TeD follows the VHDL (Bhasker 1996) language in that for each entity, at least one architecture must be defined which specifies that entity's behavioral model. The architecture defines the *state* of the entity (variables to be used during the simulation), and its *behavior* (which includes how it will respond to events that it receives). In the TCP example, an architecture for a NODE might contain a routing table (state), and a process which forwards packets when they are received (behavior). In fact, *processes* are the primary means by which the behavior of an entity is described. One entity can have multiple processes, and they can pass information and synchronize with each other if they are connected by *internal* channels. Such channels reside within the entity, but behave otherwise identically to external channels.

Functions can also be used to describe the behavior of an entity, so long as the behavior described does

```
event Packet {
  int seq_number;                       // packet sequence number
  . . .                                       // lots of other fields
}
channel packetChannel { Packet }   // channel type declaration

entity Node( int numLinks ) {             // takes one parameter
  channels {
    // a channel array to be mapped to the connecting links
    inout packetChannel nodeChannel[numLinks];
  }
}
architecture NodeArch(int IPaddress) of Node( int NumLinks ) {
  state {
    RtrTable r_table;   // routing table for forwarding packets
  }
  channels {
    packetChannel forwardChannel;        // an internal channel
  }
  behavior {
    // will contain any agent subentities at a node
    component theAgents;
    process #1 handle( nodeChannel[0 to numLinks-1] );
    process #2 forward( forwardChannel );
  }
}

entity Link {
  channels {
    // external channels, mapped to the two connected nodes
    inout packetChannel linkChannel[2];
  }
}
architecture LinkArch of Link {
  state {
    PacketQueue q;                    // to hold packets in transit
  }
  channels {
    packetChannel sendChannel;             // an internal channel
  }
  behavior {
    process #1 send( sendChannel );
    process #2 handle( linkChannel[0 to 1] );
    function update_stats(int packet_class);
  }
}
```

Figure 2: Simplified Pseudo-TeD Code Defining a Packet Event with Node and Link Entities

not involve synchronization or use of message channels. It is advantageous to use functions when possible, as TeD processes have additional overhead. However, as we will see, our transformation frequently forced us to employ processes owing to the TeD constraints on functions and synchronization.

Figure 2 presents an example of typical TeD code, and Figure 3 presents a diagram of what a modeled network might look like.

4- CONVERTING *ns* TO TeD

Because both *ns* and TeD are based heavily on C++, the framework of the conversion involved a direct translation from pure C++ in *ns* to TeD "encapsulated" C++ structures. However, because TeD is, in a sense, a narrower version of C++—not just any C++ code can be transplanted into a TeD model—there were many constraints which hindered a completely straightforward translation. In building the TeD model, we sought to ensure consistent behavior with *ns*, and intuitive (and efficient) implementation. Consistent behavior with *ns* will make verification of the transformation straightforward, while an intuitive implementation makes for a clean and easily extensible model.

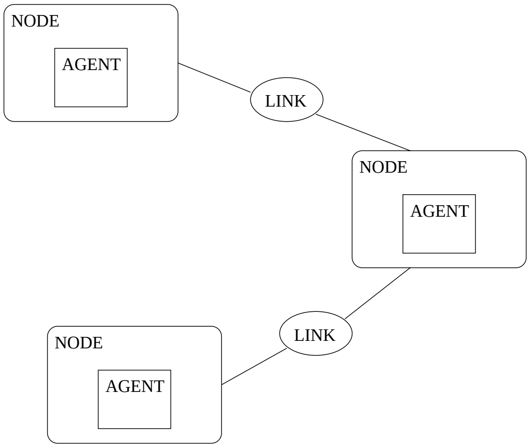


Figure 3: A Simple Network as it Might be Constructed in Either TeD or *ns*

TeD constraints make the achievement of both goals non-trivial. First we shall briefly summarize the general transformation approach. Then we'll examine the restrictions which had the most impact on the structuring of the TeD code, and the difficulties they presented. Finally, we discuss how we worked around these difficulties.

4.1- The General Approach

To make the translation as simple as possible, we mapped *ns* constructs directly into analogous TeD constructs whenever possible.- Specifically, C++ classes in *ns* became entities in TeD, with the exception that the *ns Packet* class was more naturally implemented as a TeD event. *ns* class methods became either functions or processes of the corresponding TeD entity. Typically, if an *ns* method passed a *Packet* object as one of its parameters, that method was translated into a TeD process (event-driven) which is invoked whenever a packet arrives on one of its channels. Other *ns* object methods are converted into TeD functions with identical parameters. (Section 4.1.1 discusses this further.)

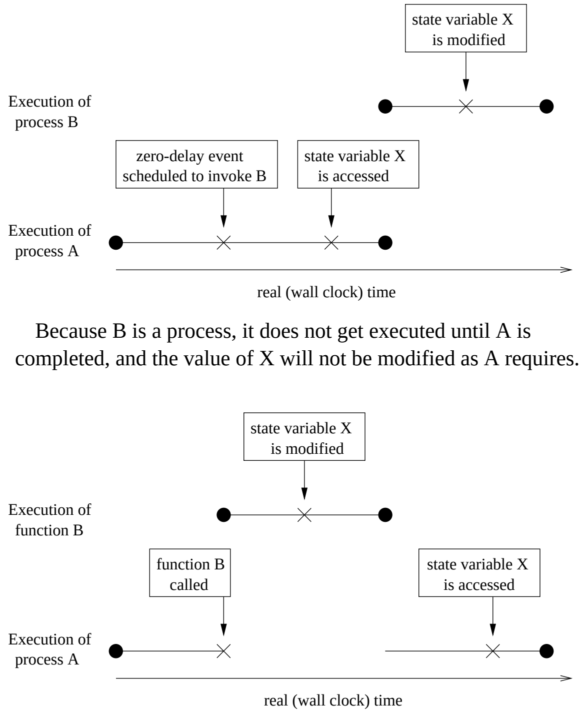
A consequence of this transformation is that *ns* methods that pass *Packets* as arguments to other *ns* methods must be transformed into TeD processes, and must establish TeD channels between TeD processes corresponding to *ns* caller and callee methods. If caller and callee are member functions of the same object, the corresponding TeD channel is internal; it is otherwise external. For example, a NODE passing a PACKET to a connecting LINK uses an external channel in the transformed code, while one process in an AGENT passing a PACKET along to another process within that same AGENT uses an internal channel.

The primary *ns* building blocks, NODE, LINK, and AGENT, all map directly into corresponding TeD entities. The latter provides an excellent example of how TeD separates interface specification from implementation. *ns* provides a number of different agent types, e.g., TCP agents, sink agents, ftp agents. In the *ns* internals a specific agent type is created by defining a class derived from the AGENT base class, specializing methods and additional data structures as needed. Similar variations are possible with the *ns* DATA_SOURCE class, and its corresponding architectural variants in TeD. In TeD, the entity definition corresponds to the base class (think of it as a virtual base class) and the architectures correspond to derived classes. The point of TeD's approach this way is to emphasize the distinction between interface definition and implementation.

4.1.1- Processes and Functions

TeD provides a process-oriented view of behavior, as TeD model behavior is expressed though the interaction of TeD processes. TeD processes interact by sending events to each other through TeD channels; TeD processes synchronize through blocking "wait" statements that cause the process to suspend for some specified duration of simulation time or until some event appears upon a channel.

TeD provides both functions and processes that can read from and write to model state variables. Functions are the usual sort one finds in imperative programming languages, and they may call a limited number of TeD macros that reference the simulation clock, and manipulate state variables. In particular, a function *cannot* refer directly to a TeD event data structure which means it cannot send or receive events on a TeD channel. Nor can it call any TeD synchronizing statement (a wait).


state variable X is modified
Execution of process B
zero-delay event scheduled to invoke B
state variable X is accessed
Execution of process A
real (wall clock) time
state variable X is modified
Execution of function B
function B called
state variable X is accessed
Execution of process A
real (wall clock) time

Because B is a process, it does not get executed until A is completed, and the value of X will not be modified as A requires.

Since B is a function, there are no synchronization problems to worry about.

Figure 4: Examples of Process and Function Interaction

TeD processes have time and space overhead, as process representation persists in the TeD internals. Furthermore, the flow of control between processes is different than that between ordinary functions. Therefore TeD, we prefer to convert *ns* methods to TeD functions. However, this is impossible if the method schedules events, or is a method that is called first to process an *ns* event occurrence. An *ns* method is readily converted to a TeD function if it only reads and writes architecture state variables. It can be converted to a TeD function if it reads values from an event state, but only with a bit of hands-on modification. One needs first to identify which fields of the event are read. A TeD function cannot read these directly, but it *can* read new variables passed as extra parameters. The code calling the function uses TeD macros to extract field values from TeD events, and passes those values along to the function. While this sort of transformation has its attractions, it will greatly complicate any future attempt we might make to fully automate the transformation of *ns* code to TeD.

Figure 4 shows example time-lines of executing simulations with interrelated processes and functions.

An additional reason for preferring functions to processes is that flow-of-control is more apparent. A function call behaves exactly as one expects. However, if one TeD process sends another an event through a channel, the recipient process will not execute before the sending processes has suspended, even if the channel is internal and has zero time delay associated with it. Thus it can be challenging to correctly transform an *ns* calling sequence where method

A calls method B, B then schedules an event and returns to method A. If the post-call code in A depends on state-modifications done by B, we cannot mindlessly turn A and B into processes that share a channel, because the state changes anticipated by A in the original *ns* code will not have taken place. Depending on A and B there are some approaches one can consider. If A is the only caller of B one may be able to introduce an additional channel between processes representing A and B for the purpose of B reporting its completion and A blocking on that report. A's call to B is transformed into a process representing A sending an event down one channel, and waiting for a response on the new channel. B's process, of course, must then send an event down the new channel when it has completed its task. If A's call to B is nested inside of an **if** conditional, **switch**, or a loop, then A's process will have to be rewritten to hoist the new call to **wait** out of the condition—TeD permits calls to wait statements only at the lowest level of nesting in a process's code (Premore, Nicol, and Liu 1996). Such a transformation is dependent on the specifics of the process code.

The introduction of an acknowledgment channel will not work if method B is called by more than one method, because the recipient of the acknowledgment must be unique. One can finesse this difficulty at the introducing a channel for each unique caller (and deal with the issue of caller identity so that the acknowledgment can be sent through the correct channel).

4.2- Exceptions to the General Approach

Even though TeD has many features which aid general network simulation, we found *ns* using constructs for which there was no immediate replacement in TeD. These include event cancellation, dynamic memory allocation, and process synchronization.

4.2.1- Event Cancellation

Some *ns* agents send out packets of data (one such agent is a TCP_AGENT) and simultaneously send a (future) time-out message to another agent. The *ns* mechanism is to pass a future event to the *ns* scheduler, for, if an acknowledgment is not received by the time-out event, special action is taken. Typically though the acknowledgment is received in time, and the processing of that acknowledgment directly cancels the time-out event. Event cancellation of this type is not supported in TeD.

We can accomplish the same effect in TeD by having ack message processing mark the sent packet as acknowledged, and by having a time-out event check whether its packet was acknowledged. TCP rules for acknowledgment allow us to minimize the additional saved state to only one word (the earliest sent package not yet received). However, event cancellation would still be useful and would reduce—potentially significantly—the overall memory use in the simulator (by eliminating time-out events whose packets have arrived) and the time spent manipulating the event list, and the time spent firing up a process that will only turn around and suspend itself.

4.2.2- Dynamic Memory Allocation

Dynamic memory allocation is a well-known problem area for optimistically synchronized parallel simulators; the difficulty stems from the need to maintain value *histories* in the variables using such space. On the other hand, C and C++ programs use dynamic memory constructs freely, and *ns* is no exception.

ns dynamically creates and destroys packets. Happily, since these corresponded to events in TeD, and the TeD language provides a structure for manipulating them (creating, sending, and receiving), no dynamic memory issues arose—it was built into TeD, and no memory needed to be allocated explicitly in the modeling process. On the other hand, *ns* is rife with dynamic data structures such as linked lists that are implemented naturally using dynamic memory. Our sole recourse is to use statically allocated arrays, sized maximally. Determining maximal array sizes is a dark art given the variant behavior one expects of discrete-event simulations.

4.3- Process Synchronization

As described in Section 4.1.1, we prefer transforming *ns* methods into TeD functions rather than TeD processes. However, there can be times when TeD processes cannot be avoided, and this leads to synchronization problems. Consider the following situation: a method for the *Agent* class, called `send`, has a loop which each pass calls one of two other methods, `output` or `sched`, for which the order of invocation matters. Restrictions discussed earlier force these latter methods to be transformed into TeD processes; `send` invokes them by sending them events along internal channels, with no delay. However, this means that all of those events will "arrive" at exactly the same time. There is an undocumented default order TeD uses to evaluate these "same-time" events, and being static it cannot be adapted to the particular sequence called for by the equivalent *ns* code. In this case we could have included acknowledge channels (as described earlier), or finesse the problem through the simulation clock. Choosing the latter method we enforced the desired sequence by adding insignificant but mathematically present delays to the invocation times.

Simultaneous events are a common source of problems for simulators of all kinds. A more esthetically pleasing solution in this case would be possible if TeD provided semaphores shared by processes bound to the same entity.

5- PERFORMANCE ISSUES

The strategy we have outlined for transforming *ns* source code into TeD has the advantage of relative simplicity, and (we hope) provides a set of readable TeD objects. Since we track the *ns* logic as closely as we can, validation of the transformed code is a matter of ensuring that *ns* and TeD versions of the same problems yield statistically identical output (we would have to modify *ns* in order to synchronize random number streams and have the two versions produce deterministically the same sample path from the same random number seeds).

The downside of this approach is that it generates events in TeD where in *ns* there are function calls. The performance implications are potentially serious. Still, our initial goal is to get an accurate TCP/IP simulator up in TeD first, and then work at performance optimizations. Even with the existing approach, the potential exists for larger simulations and (perhaps) faster simulations by exploiting parallelism.

Another potential problem area is the size of state memory. Large arrays in TeD must be declared where *ns* uses dynamic lists. These must be “state” arrays, saved by the optimistic simulator, at potentially high cost. TeD does provide a “large state” mechanism for incremental state-saving. Nevertheless, the cost of state-saving is generally acknowledged as the most serious among Time Warp overheads, and is one we will have to examine carefully.

6 CONCLUSIONS

We are developing TCP/IP model libraries for the TeD parallel simulation language. To ensure the accuracy of our models, we are transforming the logic of the *ns* simulator into TeD. This paper reports on the approach we are using, the difficulties we encounter and their solution, and the tradeoffs of this approach. We find that while TeD lacks constructs and imposes restrictions that complicate an otherwise straightforward translation, essentially we are able to map the *ns* logic into TeD without destroying its logical flow.

After validating the TeD models so created, we will evaluate the performance of TeD versus *ns* to better assess the performance price we pay for the direct translation approach, and then of course evaluate the parallel performance of the TeD models. Finally, anticipating much room for performance optimization, we will revisit the overall structure with an eye towards limiting the number of extraneous events. Ultimately we aim to provide the TeD user community with a library of modular high performance TCP/IP models.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant and CCR-9625894 and DARPA Contract N66001-96-C-8530.

REFERENCES

Bhasker, J. 1996. *A VHDL Primer*. Prentice Hall.

Das, S., R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. 1994. GTW: A Time Warp system for shared memory multiprocessors. In *Proceedings of the 1994 Winter Simulation Conference*, ed. J. D. Tew, S. Manivannan, D. A. Sadowski, A. F. Seila, 1332–1339.

Perumalla, K., A. Ogielski, and R. Fujimoto. 1996. MetaTeD—A meta language for modeling telecommunication networks. GIT-CC-96-32, College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

Perumalla, K., and R. Fujimoto. 1996. A C++ instance of TeD. College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

Perumalla, K., and R. Fujimoto. 1996. GTW++—An object-oriented interface in C++ to the Georgia Tech Time Warp system. GIT-CC-96-09, College of Computing, Georgia Institute of Technology, Atlanta, Georgia.

Premore, B., D. Nicol, and X. Liu. 1996. A critique of the Telecommunications Description language (TeD). PCS-TR96-299, Department of Computer Science, Dartmouth College, Hanover, Hew Hampshire.

AUTHOR BIOGRAPHIES

BRIAN J. PREMORE received the B.S. degree in Mathematics and Computer Science from Clarkson University in 1995. Since then he has been a Ph.D.

student in Computer Science at Dartmouth College, where he is working on parallel simulation.

DAVID M. NICOL received the Ph.D. in Computer Science from the University of Virginia in 1985 and is presently an Associate Professor of Computer Science at Dartmouth College. He is an area editor for the ACM's *Transactions on Modeling and Computer Simulation* and an associate editor for the *INFORMS Journal on Computing.* He has served as the 1989 Program Chairman and the 1990 General Chairman of the Workshop on Parallel and Distributed Simulation (PADS), has served on the PADS Steering Committee, and in 1996 was the Program Chairman for the ACM Sigmetrics Conference. His interests are in parallel simulation, performance analysis, and algorithms for mapping parallel workload.