Realizing Large-Scale Interactive Network Simulation via Model Splitting

Nathanael Van Vorst and Jason Liu
School of Computing and Information Sciences
Florida International University
Emails: \{nvanv001, liux\}@cis.fiu.edu

Abstract—This paper presents the model splitting method for large-scale interactive network simulation, which addresses the separation of concerns between network researchers, who focus on developing complex network models and conducting large-scale network experiments, and simulator developers, who are concerned with developing efficient simulation engines to achieve the best performance on parallel platforms. Modeling splitting divides the system into an interactive model to support user interaction, and an execution model to facilitate parallel processing. We describe techniques to maintain consistency and real-time synchronization between the two models. We also provide solutions to reduce the memory complexity of large network models and to ensure data persistency and access efficiency for out-of-core processing.

Keywords—network simulation, interactive simulation, network experiment, online control and visualization

I. INTRODUCTION

Simulation is useful for studying large complex networks, in particular, the emergent behaviors and large-scale phenomena, which are difficult to scale down due to the interaction and inter-dependencies between components at different layers of the system that cannot be easily abstracted away. And yet, large-scale network simulation is not used commonly today. This can be attributed to the lack of easy-to-use large-scale network simulators and realistic network models that can withstand the test of time.

There is an important distinction between developing a large-scale network simulator and a large-scale simulation experiment. The former copes with the computational issue of running parallel simulation on high-end computing platforms, while the latter deals with the challenges of creating topologies and traffic conditions resembling the target network. In between, different players assume different roles.

A parallel simulation engine provides a method for decomposing large-scale models into an interconnected graph of logical processes, capable of processing simulation events concurrently on today’s parallel and distributed platforms. Large-scale network simulation produces a huge number of simulation events, thus providing ample opportunities for parallelism. A network simulation framework extends the parallel simulation engine with domain-specific models, including common network components (such as subnetworks, routers, hosts, links, and network interfaces), and basic programming constructs (such as sending and receiving messages and scheduling timers). Based on these network components and constructs, network models can then be developed for specific network protocols, services, and applications. Most existing network simulators offer a good selection of common protocols at different layers of the network stack, such as Ethernet, TCP/IP, BGP, OSPF, DNS, etc. Network experiments consist of a detailed specification of a network topology and traffic pattern for specific network scenarios for testing applications. Depending on the goal of the simulation, a network experiment typically needs to reproduce the network conditions observed on real networks at some level.

To date, several large-scale network simulators have demonstrated good performance and scalability with various degrees of success. They are confirmed cases of good parallel simulation practice. For example, TeD [1] adopts a time warp simulation engine for running large-scale network models. PDNS [2] uses a federated approach to parallelize a popular sequential network simulator. SSFNet [3] features a programming interface designed for large complex models using conservative synchronization protocols. GTNetS [4] is also a parallel network simulator with a good set of network protocols. ROSSNet [5] uses the technique of reverse computation for compact, light-weight optimistic parallel simulation of large network models.

All the above large-scale network simulators focus primarily on the performance aspect of the simulators, and all of them failed, to some degree, to capture the attention of the network research community at large, which nevertheless is in great need of verifiable large-scale network simulation tools. The failure can be partially attributed to the difficulties in using a large-scale network simulator to conduct experiments. We believe this can be attributed to a poor separation of concerns. A network researcher is more interested in the development and use of realistic, verifiable large-scale network models and network experiments, while a simulator developer is more concerned with methods for developing the parallel simulation engine and the network simulation framework, which can effectively project large-scale network models onto parallel platforms to achieve good parallel performance. This separation of concerns inevitably brings complexities both in the description of the network models and the execution of the models on parallel platforms.

Our work begins with the notion that a large-scale network simulator shall be both easy to use and able to efficiently execute on parallel platforms. In particular, the simulator must
provide a user-friendly interface which allows users to easily
focus on developing complex network models and conducting
large-scale network experiments with little concern about
model execution issues. Such user-friendly interfaces need to
provide a complete end-to-end workflow for network exper-
imentation, from the initial development and maintenance of
network models, through the configuration and execution of
the network experiments, to the visualization and analyses of
the experiment results.

We introduce the method of model splitting, which naturally
divides the system into an interactive model and an execution
model. The interactive model is an iconic, succinct represen-
tation of the target network configurations; it is intended to
be small so that it can fit in a client machine and possibly
work in conjunction with a well-designed user interface. The
interactive model is used to address the user facing aspects of
the system: it allows the user to easily specify the network
models, visualize the network configurations, scrutinize the
model parameters, inspect or modify the simulation state,
and allow for online monitoring and steering of the network
experiments. In contrast, the execution model is a full-fledged
representation of the large-scale network models for the target
high-end computing platforms. The execution model needs to
include partitioning and mapping information for the network
models to be deployed and run on the parallel platforms. The
execution model also needs to capture the detailed evolution of
the simulation state. In addition, certain simulation state needs
to be tracked or recorded during the experiment for online or
postmortem data collection and analysis.

To maintain consistency between the interactive model and
the execution model, we introduce a process called network
scripting, which allows the system to automatically create
the interactive model and the execution model from a single
specification of the network configuration. To facilitate real-time
high-capacity interaction between the interactive model and
the execution model, we introduce a streamlined monitoring
and control mechanism, that can modify the runtime state
of the simulation and filter state updates between the execution
model and the interactive model in support of interactive
experiments. To reduce the model size, we introduce the technique of model replication, which
allows sharing data for representing the network topologies
with identical structures, the state of network entities and the
behavior of network protocols with identical functions. To
ensure efficient in-memory processing of the interactive model,
we present a model caching scheme on top of a database
management system supporting data persistence of large-scale
network models that reside out-of-core.

We implemented the model splitting approach in PRIMEX,
which is a large-scale network simulator with a real-time
extension that allows parallel and distributed simulation of
large-scale networks in real time so that the simulator can
interact with real network applications and real network traf-

Fig. 1. A schematic view of model splitting.

interface, through an interactive python console, or using
scripts written in XML, Java, or Python. The interactive model
is persisted to a database so that the users can later reuse the
model, or simply a part of it, to construct other models. The
execution model, equipped with the information of the network
configuration and the target execution environment, is run at
the parallel machines. During the experiment, the user can
use the graphical user interface to visualize the state of the
network and interactively control the network experiment by
reconfiguring the network model.

The remainder of this paper is organized as follows. In
section II we describe the model splitting technique and iden-
tify major issues which arise from the method. In subsequent
sections (sections III to VI), we present our solutions to
address these issues of model splitting in detail. We also
describe our implementation and show preliminary experiment
results in these sections. Finally, we present the conclusion and
outline our future directions in section VII.

II. Model Splitting

In this section we present the overall architecture of model
splitting and describe the associated problems and solutions.

A. Architecture

The basic premise behind model splitting is that a user’s
interaction with a model should be explicitly separate from
how a model is executed. Fig. 1 shows a schematic view of
a simulation system that employs the model splitting
 technique. With two separate models—the interactive model
that runs on the user workstation and the execution model
that runs on the parallel platform—we are able to address the
separation of concerns. The interactive model should focus
on model configuration, experiment specification, interaction
and visualization, by supporting user facing tools such as the
scripting languages, the graphical user interface, and interac-
tive consoles. In comparison, the execution model should focus
on execution issues, such as parallelization, synchronization,
communication, and data collection on parallel platforms.

Both models are derived from the network models imple-
mented for various network protocols and applications, and the
experiment specification with detailed network topology and
traffic description. However, they intend to address different
concerns. More specifically, the interactive model has four
important design considerations. First, the interactive model
shall be constructed and configured easily, through a graphical user interface or via a scripting language, or both. Second, the interactive model needs to be persistent across different experiments, for repeatability, for debugging, and for model reuse. Third, the interactive model needs to maintain a small memory footprint as it is expected to interact with the user on a user workstation (the front end) with fairly limited memory. Finally, the interactive model can be geographically separated from the execution model: the user configures and visualizes network experiments at the front end on a user workstation remotely connected to the back end, which is the parallel computing cluster running the simulation.

In contrast, there are five key design considerations for the execution model. First, the execution model shall be able to capture the network operations at sufficient levels of detail that satisfy the fidelity requirements of the simulation study. Second, the execution model must be able to run efficiently at the back end on the parallel platform. The model needs to be partitioned and mapped onto the parallel machines to minimize the synchronization overhead. Third, the execution model needs to conserve memory; this is especially important when dealing with extremely large-scale network models. Fourth, the execution model needs to interoperate with the interactive model running at the front end (at the user workstation), for network monitoring and visualization, user interaction and dynamic reconfiguration. Finally, the execution model may generate a huge volume of simulation results, which needs to be collected efficiently for postmortem analysis.

B. Problems and Solutions

Splitting the network model and experiment specification into the interactive and execution models can provide the desired separation of concerns; however, it also brings several potential problems. The first problem is that we need to maintain consistency between the two models. On the one hand, the interactive model and the execution model should be based on the same implementation of the network models. That is, they should contain the same definition of network entities and protocols with the same state variables (with the same data types and ranges). Inconsistencies often arise at the development stage when the network models are constantly being changed. On the other hand, the two models should be derived from the same experiment specification. They should follow the same network configuration, with the same number of instances of network entities and protocols, and the same configuration parameters. Inconsistencies may happen when the user interactively creates and configures the network experiment.

To solve this problem, we introduce the method of network scripting. Network scripting is a technique of automatically generating the interactive and execution models using the annotations embedded in the implementation of the network models. These annotations provide the meta-information of the state variables and the hierarchical structure of the network model. For example, the annotations can describe whether the state variables are configurable by the user, and if so what are their data types, what are the ranges of values they can take, and what are the default values. The annotations can also describe, for example, the minimum and maximum number of sessions that can be instantiated for a protocol on a host or router. The meta-information is used to generate the schema for the scripting language, the interactive console, and the graphical user interface, which are then used to create and interact with network experiments. The interactive model also has an embedded database management system to deal with large-scale models that must be handled out-of-core. The meta-information is used by the execution model to prepare for the full-fledged network model and enable the mechanism for interoperating with the interactive model (and the user) during the simulation.

The second potential problem associated with model splitting is that the two models need to be synchronized during runtime. That is, the state update information needs to be exchanged between the two models: modification to the state of the interactive model needs to be sent to the execution model in a timely fashion, and vice versa. For large-scale networks, there can be considerable amount of information exchanged between the two models causing significant overheads. Furthermore, since the interactive model and the execution model can be geographically distributed with relatively constrained connections in-between (i.e., with large latencies and low bandwidths), the update information may be delivered with significant delay. To overcome this problem, we propose a real-time monitoring and control framework to automatically and efficiently synchronize the states of the interactive and execution models. This is achieved by reducing the volume of update information and dynamically adjusting the synchronization intervals in response to user input.

The third problem with model splitting is about the memory consumption. The execution model is engineered to run on parallel machines to cope with the computational demands of large-scale network experiments. The interactive model, however, runs on a user workstation, which does not have sufficient resources to handle large-scale experiments in the same way as the execution model. We need to conserve memory for both models, especially for the interactive model. We propose a two-pronged approach to solve this problem. First, we use model replication to minimize the memory footprint of large-scale network models. Model replication is a technique for constructing large network topologies using light-weight copies of identical sub-structures. This method is especially effective at conserving the memory required to conduct routing on the simulated network. Second, for the interactive model, we use an efficient model caching algorithm on top of a database management system for handling experiments that cannot be contained in the main memory. We extend network scripting to embed functionalities within the interactive model which allow for transparent paging of the interactive model to and from the database.

In the subsequent sections, we describe each of the aforementioned solutions in more detail.
III. NETWORK SCRIPTING

Network scripting is a technique of embedding simple annotations within the implementation of network models to help create a consistent representation of the target network, both in the form of the interactive model (for users to create, configure, visualize, and control network experiments), and the execution model (for efficient execution of the network experiments on parallel platforms). Using the annotations, network scripting also automatically creates the mechanisms to synchronize the runtime state of the models during the experiment execution.

A. Related Work

Existing network simulators have partially addressed the complexities of configuring network models. A common approach is to use a description language to describe the configuration of the network, e.g., the Domain Modeling Language (DML) used by SSFNet [3], and the Networ Description language (NED) used by OMNeT++ and INET [7]. Using a separate language can simplify the specification of network experiments, including the structure of the networks and the parameters for individual components. However, in doing so, one must manually ensure the consistency between the description language and the model implementation.

The Telecommunication Description Language (TeD) [1], as one of the earliest efforts of parallel network simulation, takes a different approach. Similar to VHDL, TeD provides a modular description of the hierarchical structure of the network model together with the definition of detailed behaviors of the components. The problem with this approach is that the user is forced to deal with the complexities associated with the detailed model implementation in the same model description language [8].

Another method is to adopt a split programming approach, where the simulation code is developed in a compiled language and the logic of building and configuring the network models is written in a scripting language (such as Tcl). This is the approach employed by ns-2 [9] and GTNeTS [4]. The split programming approach certainly simplifies the specification of the network and provides automatic consistency between the scripting language and the model implementation. However, the two need to be on the same machine—the simulator cannot handle situations where the specification, configuration, and visualization is done interactively on a machine separate from where the model is run.

B. Annotations

Fig. 2 illustrates the procedure and the functioning components of network scripting. The developer annotates the implementation of network models. All basic network elements, such as subnetworks, routers, hosts, links, interfaces, and protocols, are represented as model nodes, each containing the state of the corresponding network element and defining the logic and behavior of the network element. A network model is organized as a hierarchy of model nodes: a network may contain subnetworks, routers, hosts, and links; a router or host may contain one or more network interfaces and a stack of protocols.

With network scripting, the model nodes shall be embedded with annotations to indicate the relationship of the model nodes within the hierarchy and the access types of the state variables. More specifically, the model nodes shall contain structure annotations, which are used to ensure the model nodes can be instantiated properly in relation to one another. For example, a network can have routers as its children, but not the other way around. In this case, a child-type annotation is used to specify whether a model node can be composed of specific types of model nodes as its children. The annotations can also specify the minimum and maximum number of instances allowed for a specific type of model node. For example, a router requires that it contain at least one network interface, but at most one IP protocol.

There are four basic types of state annotations: configurable, controllable, mutable, and sharable. A configurable state annotation is used to mark a variable that can be set by the interactive model at the configuration stage, in which case the value needs to be propagated to the execution model before setting up the experiment. Each configurable state variable can also be defined with a default value: if a configurable state variable is not specified by the user, the simulator will use the default value. A controllable state annotation is used to mark a variable that can be modified by the interactive model during the experiment and the value needs to be propagated to the execution model in real time. Both configurable and controllable variables can also specify their minimum and maximum values. A mutable state annotation indicates that when the variable is modified by the execution model during the experiment, its value needs be propagated to the interactive model in real time. The mutable and controllable variables are essentially used for online monitoring and steering of network experiments. A sharable state annotation indicates that the variable can be shared among the replicated instances of the model node, in which case only one copy of the state variable needs to be maintained in the simulation. In section V we explain in more detail the model replication technique, designed to reduce the memory consumption of large network models.

The annotated network models are parsed to generate the model schema, which contains the meta-information about the state variables of the model nodes and the structural relationship of the the model nodes. The model schema is
then used to generate the interactive model and the execution model.

The interactive model maintains a lightweight representation of network model instances, which we call the model graph. The model schema is used to generate three major components of the interactive model. It generates an application programming interface (API), which contains the definition of the interactive model nodes. An interactive model node is a memory-efficient representation of the corresponding model node defined in the annotated network model, which contains only the configurable, controllable and mutable state variables. The API provides a common framework for all external tools, such as the graphical user interface, the interactive console, and the scripting language, which one uses to configure and interact with the execution model visually or programmatically.

The model schema generates the database schema, which is used to marshall model nodes to and from an external database. The network model is persisted to the database so that one can repeat the network experiment or reuse any element of the network model. Persisting to a database also allows us to handle large-scale network models out-of-core.

The model schema also generates the synchronization mechanism. After the user finishes the configuration of the network model and the specification of the target runtime environment, this information needs to be “compiled” and sent to the execution model. Also, preprocessing needs to be done at this stage, such as the automatic IP address assignment for the network interfaces, the pre-calculation of static routing tables, and the partitioning of the network model for the target parallel platform. During the execution of the experiment, the interactive model can update the controllable variables, which need to be propagated to the execution model in real time; similarly, the mutable variables can be updated by the execution model and need to be propagated to the interactive model in real time. In section IV we describe the real-time interactions between the two models in more detail.

The execution model maintains a full-fledged representation of network model instances, which we call the partitioned model graph. The execution model includes both annotated and unannotated states of all model nodes already partitioned to run on the parallel machines. For the annotated states, the execution model uses the model schema to create a variable map so that each state can be individually retrieved and modified. The execution model also provides a special treatment for sharable state variables: it groups all sharable states within a model node together so that the entire collection can be shared among replicated node nodes. We discuss model replication in more detail in section V.

The model schema generates the synchronization mechanism for the execution model, so that it can import information about the network configuration and the preprocessed results from the interactive model. The execution model also needs to receive real-time updates of controllable variables from the interactive model, and send real-time updates of both controllable and mutable variables to the interactive model. The partitioned model is executed by the parallel simulator on the parallel platform, which generates simulation results and trace data to be stored in a distributed database or a parallel file system.

C. Implementation

We implemented network scripting in the PRIMEX simulator. We annotated the network model implemented in C++. Fig. 3(a) shows an example of an annotated model node, which represents a network interface. In the annotations block, we defined three configurable variables: bitrate is the bandwidth of the network interface, maxqlen is the maximum queue length, and dropprob is the packet drop probability. Two of them are also marked as shared. We also defined two mutable variables (inbytes and outbytes) for collecting the statistics on the number of bytes received and sent by the network interface during simulation. A network

```java
class NIC extends ModelNode {
    NIC(ModelNode parent) {
        super(parent);
    }
    void setInitBitrate(long v) {
        long getInitBitrate() {
    }
    void setInitMaxqlen(long v) {
        long getInitMaxqlen() {
    }
    void setInitDropprob(double v) {
        double getInitDropprob() {
    }
    void setInitBitrete(long v) {
        long getInitBitrate() {
    }

    NICQueue addREDQueue() {
        NICQueue getQueue() {
    }
    NICQueue addDropTailQueue() {
        NICQueue getQueue() {
    }
}
```
interface is required to have one and only one child of type NICQueue—either a drop-tail queue or a RED queue depending on the configuration. There are also unannotated variables defined within the model node. Here we show only sendtimer, which is used for scheduling packet departure events.

The annotated network models are parsed to generate the interactive model and the execution model. Fig. 3(b) shows the corresponding interactive model node, which is implemented in Java. There is also an XML schema (not shown) generated from the annotated network model. For each annotated variable in a model node, the interactive model provides the corresponding getter and setter methods. If the variable is configurable (such as bitrate, maxqlen, and dropprob), we define getInit and setInit methods so that the user can check and modify the variable during model configuration. If the variable is controllable (such as dropprob), we define both getter and setter methods so that the user can query and modify the value of the variable during runtime. If the variable is mutable (such as inbytes and outbytes), we only need to provide the getter method, since the execution model will update this variable in real time.

In the example, the network interface has one child type, NICQueue; we define a getQueues method to obtain a list of network queues (although only one is allowed according to the model schema). For each model node type derived from NICQueue, we create a separate add method. The model schema identified only two derived model node types: DroptailQueue and REDQueue; that’s why there are two add methods.

We implemented a graphical user interface (using Prefuse [10]) for the users to visualize and interact with the network model. We also implemented an interactive console in Python, which uses a python interpreter written in Java [11]. PRIMEX allows the user to specify the network model in XML, Python, or Java. All these tools are developed using the above API generated from the model schema.

Fig. 3(c) shows the definition of the corresponding execution model node for the network interface in C++. All sharable variables (bitrate and maxqlen) are put together and defined in an inner class, called shared. The default constructor of the model node creates the shared region when a new model node is created; however, if the new model instance is “replicated”, the second constructor is used so that the region is shared among the replicated instances. The configurable, controllable, and mutable variables are all wrapped using templates (Configurable, Controllable, and Mutable) for a uniform treatment of all data types when synchronizing with the interactive model. In addition, for controllable and mutable variables, the templates also track modifications so that the values can be propagated to the interactive model during runtime.

IV. REAL-TIME INTERACTIONS

To support real-time interaction with network experiments, the interactive and execution models must synchronize their runtime state by exchanging state updates. In this section, we describe the design and implementation of an online monitoring and control framework for this purpose.

A. Design

The monitoring and control framework, shown in Fig. 4, consists of: 1) a local controller, collocated with the interactive model at the user workstation; 2) a master controller, chosen to run at one of the parallel machines; and 3) the slave controllers, each running together with the corresponding parallel simulator instance on the parallel platform. The master and slave controllers are collocated with the execution model. When the execution model modifies the controllable or mutable states, the simulator instance generates state updates and sends them to the slave controller collocated on the same parallel machine. The slave controller then forwards the state updates to the master controller, which sends them to the local controller at the user workstation. The local controller eventually updates the interactive model. In the opposite direction, when controllable states are modified in the interactive model, state updates are generated and forwarded to the master controller. The master controller, with the help of the partitioning information, sends the state updates to the corresponding slave controller, which modifies the execution model accordingly.

We implemented two mechanisms for collecting data from the execution model. One is a monitor, which is associated with a state variable. A monitor can output data, periodically (with a predefined interval), on-demand (whenever the state changes), or both (by exporting the data at most once for a given period if it is changed). The other mechanism is an aggregator, which is associated with a collection of state variables. An aggregator is used to produce the aggregate statistics from a collection of state variables, and can thus reduce the amount of data that needs to be collected. Note that, in addition to supporting real-time interaction, these mechanisms can also be used to support data collection for postmortem analysis, by storing the data in a collocated database or using a parallel file system.

The interval at which the execution and interactive models exchange state updates determines the responsiveness and timeliness of the interaction. Ideally, the interval should be kept small. However, the connection between the interactive and execution models may be constrained, making it infeasible to send updates at a high frequency. We propose two techniques to reduce the volume of state updates. The first method defines an area of interest, which is the set of interactive model nodes that a user is currently viewing or interacting with. A
user (or program) cannot access the runtime state of a model node beyond the area of interest. The interactive model keeps track of the set of model nodes that are currently viewed by the user and informs the execution model to only send the state updates within the area of interest.

The second method uses a rate limiting mechanism, which dynamically adjusts the interval at which state updates are sent. As a user interacts with an experiment, the area of interest may grow to be very large, and in extreme circumstances it may include the entire execution model. This could cause a significant backlog of state updates to accumulate on the path from the execution model to the interactive model. To prevent the backlog we can dynamically adjust the interval of the state updates so that the rate stays below the capacity of the system. This would ensure that the updates can be delivered in a timely manner, however, at a lower rate. We implemented the rate limiting mechanism in the simulator to control the time interval between sending batches of updates for all mode nodes within the area of interest.

### B. Evaluation

To evaluate the effectiveness of using the area of interest and the rate limiting scheme, we conducted an experiment using a synthetic network model with 12K model nodes running on a Linux cluster. We placed the local controller, the master controller, and the slave controller on three separate machines connected by a gigabit switch and measured the throughput at the simulator and the three controllers. First, we fixed the area of interest to encompass 75% of the model and changed the update frequency to produce different loads. Fig. 5 shows the throughput at different measurement locations, averaged over ten separate trials (the confidence intervals are too small to be seen on the figure). Although the simulator can output updates at a high rate, the throughput peaks at around 74K updates per second at the slave controller, and around 68K updates per second at the master and local controllers.

This experiment shows the throughput limitations of the monitoring and control framework implementation. If the offered load is higher than the achievable throughput, the system may become congested, which would cause further delays for the updates to reach the interactive model. We conducted another experiment, this time by changing the area of interest at the interactive model to oscillate between 25% and 75% of the model every 5 seconds. We also enabled the rate limiting mechanism to control the rate at the simulator to stay below a given threshold. We varied the threshold in the experiment and measured the mean jitter, calculated as the difference in the arrival time between the successive batches of updates. Fig. 6 shows the results. We observe that the jitter increases dramatically between 30K and 50K updates per second, which is before the peak throughput. This could be attributed to the lack of rate limiting within the batch. The experiment nevertheless demonstrates that, with the proper threshold, the rate limiting mechanism can deliver the updates with less jitter.

### V. Model Replication

Both the interactive and execution models need to conserve memory; this is especially important when handling extremely large network models. In this section, we present the model replication technique for reducing the memory complexity.

#### A. Technique

We classify the simulation state into three categories: structural, instance, and ephemeral. Structural state is used to describe the network topology of a network model: Which hosts belong to the networks? How many interfaces are attached to the hosts? How are they connected to one another? Instance state includes state variables belonging to the model node instances, such as the link bandwidth, and the TCP congestion window size; they are essential for representing the behavior of the specific network entities. Ephemeral state represents network traffic during the execution of the experiment. For packet-oriented simulation, it consists of packets stored at the network queues. Traffic modeling is beyond the scope of this paper. Here we focus only on conserving memory for the structural and the instance state.

Model replication is a technique of constructing network topologies using light-weight replicas of the model nodes with identical sub-structures. These replicas need only to maintain a single copy of the sharable state variables (identified by network scripting as shown in section III), and therefore can
reduce memory consumption. The idea of model replication is based on the observation that realistic network models can be built with similar structures. Leskovec found that one can build large networks recursively from a small network topology and the resulting network topology can preserve many network characteristics similar to those observed on the Internet, such as the heavy-tail degree distribution [12]. In our previous work [13], we explored the effects of composing network topologies with replicated sub-structures using a modified version of BRITE [14], a network generator. We found that networks generated with a large proportion of identical sub-structures still exhibit similar key graph metrics, such as the node degree distribution and the path length distribution, to those that contain no identical sub-structures.

Model replication is designed to allow users to create memory-efficient network models by sharing both structural and instance state among the model nodes that are replicated from the same base structure. The topology of the network model is represented as an augmented tree that encodes the replications. Fig. 7 shows an example of a small topology on the left and its corresponding internal representation on the right. The top-level network contains two identical sub-networks, Net1 and Net2, each also containing two identical subnetworks. In the internal representation, Net2 makes a reference to Net1, which indicates that both share the same sub-structure and use the same set of sharable variables. Similarly, Net4 makes a reference to Net3, and H2, H3 and H4 all make a reference to H1. At the preprocessing step, the tree will be expanded; however, all replicated model nodes still maintain only one copy of sharable variables. The example network contains six subnetworks, six routers, twenty links, and sixteen hosts in the two subnetworks at the top level. Eventually, only one set of sharable variables is needed for each of the two subnetworks (Net1 and Net3), the two routers (R1 and R2), the two links (L4 and L6), and the one host (H1).

Fig. 7. Internal representation of an example network topology.

B. Evaluation

To evaluate the effectiveness of model replication, we conducted an experiment using a synthetic network. We varied the proportion of replicated sub-structures in the model and measured the memory needed to instantiate the execution model. At the base level, we used the campus network, which consists of 508 hosts and 30 routers in 17 nested subnetworks [15]. We examined three networks with 216, 512, and 1,000 campuses. We varied the proportion of replicated campuses from 25% to 99%. The results are shown in Fig. 8.

The three network models performed consistently. The memory savings due to model replication reached as much as 33%. A detailed analysis of the model’s memory usage reveals that the amount of sharable state stays around 38%. The 5% difference is due to the overhead of maintaining the replications, which include the memory for the virtual function tables, pointers to the shared state, and some auxiliary data structures to allow the model nodes to be shared among replications.

The most significant savings come from static routing. In our previous study [13], we presented a method for pre-calculating static forwarding information and sharing the information among subnetworks with identical structures in the network model. We demonstrated that this method can reduce memory consumption by as much as three orders of magnitude for large-scale network models.

Model replication can save memory for both interactive and execution models. For the interactive model, it can also improve the preprocessing time. This can be attributed to three factors. First, model replication reduces memory for storing the interactive model and therefore demands less I/O when the interactive model is persisted to the database (we discuss out-of-core processing of the interactive model in the next section). Second, putting the sharable variables together improves locality, which can be exploited by an intelligent caching technique (again, we discuss the model caching technique in the next section). Finally, preprocessing is needed by the interactive model to compile and send the information about the network configuration and the runtime environment to the execution model. The preprocessing time is reduced for replicated models because it can reuse the results from replicated model nodes.

We conducted another experiment using the same network models and measured the reduction in the preprocessing time of the interactive model. Fig. 9 shows the results. We observe
significant reduction in the preprocessing time for as much as 83% with 99% replications.

VI. MODEL CACHING

The interactive model needs to operate on large network models potentially too big to fit in the main memory. We present a model caching technique for out-of-core processing, which transparently marshals the interactive model nodes to and from a database as they are created and modified.

A. Design

Fig. 10 shows a basic design of the model caching mechanism, which we explain below. Network scripting automatically creates an interactive model node for each annotated network model node. All interactive model nodes extend a base class that includes the functionalities for out-of-core processing. The user runs the external tools, such as the graphical user interface, in separate threads to create, modify or delete the model nodes. The interactive model registers these events with the model cache, which is also run as a separate thread (step 1 in the figure).

The model cache maintains a working set of model nodes that have been recently accessed or modified. The model nodes that are not in the working set must be persisted to the database. To do this, we use an advanced Java referencing scheme, in which a model node refers to its parent and children using soft references, as opposed to strong references typically used in Java. The main difference between strong and soft reference is that softly referenced objects can be reclaimed by the Java Virtual Machine (JVM) during garbage collection. The model cache maintains a strong reference to all nodes within the working set to ensure that they cannot be reclaimed by the JVM (step 2). When new model nodes are added to the working set, and if the cache is full, the model cache needs to evict existing model nodes by removing the strong references to them and by sending requests to the queue to have the model nodes written back to the database (step 3). When the interactive model accesses a model node, it checks to ensure the node is in memory, and if it is not, the model cache will send a request to load the node from the database and wait for the completion of the requested operation (also, step 3). We designate a separate thread, called the DB thread, to handle the requests from the model cache and perform the corresponding database operations (step 4). When the DB thread finishes loading a model node, it will unblock the waiting model cache thread (step 5).

B. Caching Policy

An important aspect in the design of the model caching mechanism is the choice of the cache eviction policy. The eviction policy needs to take advantage of the access locality. We study the typical access pattern of the interactive model via an experiment. We used the same campus network model as in the previous section. We chose the model with 216 campuses, which contains a total of 425,369 models nodes for all the hosts, routers, network interfaces, links, and subnetworks.

Fig. 11 shows the results. The y-axis is the id of the model nodes being accessed. The diagonal lines indicate the model nodes are accessed in sequence at different preprocessing stages. For example, the first diagonal line shows that the model nodes (425,369 of them) are created one by one. The network topology has multiple levels and replications are applied at each level. At the bottom level are the campus networks, each with around 2K model nodes (denoted as X in the figure); the next to the bottom level has around 12K model nodes (Y); and the level above it has around 72K model nodes (Z). The figure shows that lower level model nodes are more frequently accessed because they are replicated by the other model nodes and they contain the sharable state variables.

The campus network topology and its replications account for nearly all of the locality seen in our models, as was the case in this example. Since the model nodes are accessed mostly sequentially, we choose to use a modified FIFO eviction policy. We select the victim to be the earliest created model node that is not a network and has not been replicated. This allows the model cache to give priority to networks and model nodes that have been replicated in memory.

C. Database

Another important aspect in the design of model caching is the choice of the database and the supporting framework for persisting the interactive network model. We first implemented the interactive model using the Java persistence architecture (JPA), which is a standard programming framework for managing relational data in applications. We found, however, that the JPA incurs significant overhead when dealing with large datasets; as a result, it was unable to handle network experiments with more than a few hundred thousand model nodes. We went on implementing a custom persistence framework so that we could have full control over the overhead involved in persisting and retrieving objects to and from the database.
VII. SUMMARY AND FUTURE WORK

Model splitting divides a network model into an interactive model and an execution model in order to separate the user-facing concerns, such as visualization and interaction, from the concerns for efficient execution of network models. The consistency problem between the two models is addressed using network scripting, which allows us to extract the model schema and automatically generate the interactive and execution models using simple annotations. The interoperation between the interactive and execution models is accomplished by a real-time monitoring and control framework with data filtering and rate limiting mechanisms to cope with a constrained connection between the two models. To support large network models, we present a model replication technique for reducing the memory complexity, and a model caching technique on top of a database management system for persistence.

We are investigating more robust congestion control and transport mechanisms to better support real-time interaction. We will also extend our custom database to improve its read performance. Our PRIMEX simulator has been used for conducting interactive large-scale network experiments with limited test scenarios. We would like to expose our simulator to more general network applications.

ACKNOWLEDGMENT

This research is supported in part by the National Science Foundation award CNS-0836408, and a subcontract from the GENI Project Office at Raytheon BBN Technologies.

REFERENCES


