An Evaluation of the Network Simulators in Large-Scale Distributed Simulations

Selim Ciraci
Pacific Northwest National Laboratory
Richland, WA, USA
selim.ciraci@pnnl.gov

Bora Akyol
Pacific Northwest National Laboratory
Richland, WA, USA
bora@pnnl.gov

ABSTRACT

Networks for the smart grids are characterized by millions of sensor nodes exchanging information about the status of the grid. This information exchange must be realized reliably and efficiently due to the mission critical nature of the power grid. Hence, the applications and the network protocols developed for the smart grid need go through rigorous testing and analysis before deployment. Developers usually do not have access to such a large-scale network that can be used as a controlled test-bed; therefore, network simulation becomes an essential tool for testing.

Network simulation is a well studied problem in the literature and there are various widely used network simulators. These simulators can be adopted for testing applications and protocols of the smart grid. Due to the scale of these networks, parallel/distributed simulations need to be conducted. Even though most network simulators support distributed simulations, generating a large-scale network model to simulate can still be a cumbersome task. In this survey, we describe a selection of commonly used network simulators and evaluate them with respect to the following features that can aid users in distributed simulations of large-scale networks: transparency of setting up distributed simulation, automated topology generation, information hiding, lightweight routing protocols, network error simulation, evaluation of the network model during simulation and trace analysis tools. As a complementary result, we identify two issues with network simulators that can be addressed with runtime steering methods.

Categories and Subject Descriptors

J.2 [**Computer Applications in Physical Sciences and Engineering**]: Engineering; A.1 [**Introductory and Survey**]

General Terms

Design, Performance

Keywords

network simulator, large-scale network simulation, distributed simulation

Copyright 2011 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
HiPCNA-PG'11 November 13, 2011, Seattle, Washington, USA.
Copyright 2011 ACM 978-1-4503-1061-1/11/11 ...$10.00.

1. INTRODUCTION

Networks in smart grids are characterized by millions of sensor nodes that exchange information about power usage. The smart grid applications rely on this exchange to match the power supply to the demand [8] [6]. Because power grids are mission critical, the protocols and the applications used in the smart grids have to be reliable and efficient. As such, these protocols and applications need to be rigorously tested and analyzed before the development phase. However, developers and researchers, usually, do not have access to a large-scale network that can be used as a controlled test-bed. Therefore, network simulation becomes an essential tool for testing.

Network simulation has been well studied in the literature. These studies contributed to the development of widely adopted network simulators. A network simulator is a library containing a rich set of components representing the network hardware and software. These components can be used in the simulation to generate close to real-life execution scenarios. Due to this capability, network simulators have received a lot of attention for testing the stability, performance and effectiveness of new protocols. Recently, network simulators gained support for distributed simulations of large-scale network models [12]. Here, the network model is divided among different hosts to reduce the execution time and to combine the available memory in all the hosts to store the model.

With distributed simulation support, network simulators can be used for testing the applications and the protocols of the smart grid. The scale of the network models, however, poses challenges such as long source files for setting up network models, huge trace files generated by the simulators, long execution times, and lack of lightweight protocols. These challenges, when not addressed appropriately, might increase the effort needed to use the simulator and negate the benefits of the simulation.

In order to address these challenges, new features for network simulators are required. To this aim, we have identified the following features that can ease the development and simulation of large-scale networks with distributed network simulators: *transparency of setting up distributed simulations*, *automated topology generators*, *information hiding*, *support for lightweight routing protocols*, *network error simulation*, *evaluation of the network model during simulation*, and *trace analysis*. In this survey, we report how the network simulators support these features. Our survey covers network simulators ns-2 [4] with distributed simulation extension [12], GTNetS [1], INET for OMNET++ [2], and ns-3 [5], which are generic (i.e. not designed for specific networking scenarios), open source, and support a wide range of link layers, protocols and applications. Compared to existing surveys of network simulators [15] [13] [9], our contribution is the identification of the features that aid the simulation of large-scale networks and a com-

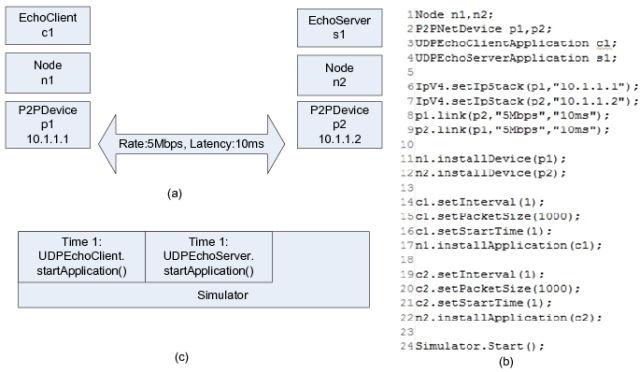

Figure 1: An example network topology (a), the program to simulate this topology (b), and the initial state of the simulator (c)

parison of the network simulators with respect to these features. As a complementary result, we also report two issues we identified with the network simulators and propose a solution approach based on runtime steering methods.

This paper is organized as follows: Section 2 provides a background on network simulation. An overview of the current network simulators are presented in Section 3. The comparison of network simulators with respect to the features aiding the development/simulation of large-scale networks is presented in Section 4. Section 5 discusses the open issues we identified with network simulators, and describes our solution approach. Finally, Section 6 presents our conclusions.

2. BACKGROUND ON NETWORK SIMULATORS

Network simulators are discrete event simulators that allow close to real-life simulation of communication networks. As the simulation mimics real-life network appliances, the network simulators follow a similar design. In this section, we briefly introduce this design and describe how simulations are programmed.

The network simulators provide the models for the following network appliances: the nodes representing the computers/routers, the network "enabled" applications running on the nodes, the network devices of the nodes, the protocol stack nodes used for communicating with each other and the packets. Each of these network appliances is implemented in an abstract class and they provide the methods for simulating the functionality of the network layer the appliance corresponds. The abstract classes are specialized to implement a specific network appliance. For example, the base class representing the network devices has methods for sending and receiving a packet. The class representing an Ethernet device overrides these methods and implements how an Ethernet card sends a packet.

Network simulators are class libraries that are imported by the users to program the simulation. Programming a simulation involves creating instances of the classes representing the network appliances and building the encapsulations between them. Figure 1-(a) presents an example network that we wish to simulate; the simulation program corresponding to this network is shown in Figure 1-(b). Here, the program starts by creating two instances of the class Node and two instances of the class P2PNetworkDevice. The class P2PNetworkDevice represents the network interface for a basic peer-to-peer connection between two computers like a laplink connection. At lines 6 - 9, the properties like link speed and latencies of the instances of this network device are set. Then, these are added to the instances of the class Node, just like adding network hardware to a computer.

The network simulation is executed by the simulator, which is implemented in a class providing the methods to schedule events, to start and to stop the simulation. The methods for scheduling events are called from the classes representing network appliances and the start/stop methods are called from the user programs. Here, events represent an occurrence related to the network, like a packet reaching its destination. An event consists of a time and a reference to the instance of the network appliance that will react to it. The simulator is a discrete event loop; it dispatches the event with the smallest time, the event receiver in turn might schedule new events to be dispatched at a later time. This continues until the stop method is called or until there are no more events left to dispatch.

In the example shown in Figure 1-(b), the call at line 24 starts the discrete event loop. Figure 1-(c) presents the scheduled events before this call. These events are scheduled from the methods setStartTime(). Once they are dispatched, they are received by the method startApplication() of the instances of the UDPEchoClient and UDPEchoServer classes. This method in both classes is implemented such that it schedules an event with a later time representing the network device receiving a packet sent by the application.

The execution time and the memory requirements of the simulator greatly depends on the size of the network model. As such, a simulation with a fairly large network model can take a long time to complete [15]. To address this problem, distributed simulations are proposed [12]. In a distributed simulation, the input model is divided across multiple simulator processes. A simulator process p executes the events from it's portion of the network model m and the events originated by other processes that are destined to a node in m. A middleware synchronizes the simulated time between the simulator processes: it ensures that when a simulator processes an event at time t, that simulator has received and processes all events (originating from other simulator processes) with timestamps less than t.

3. AN OVERVIEW OF NETWORK SIMULATORS

When programming simulations with large-topologies, it is important to reuse existing components as much as possible to reduce simulation development time. Therefore, the types of networking hardware/software that can be simulated become an important factor in the choice of a network simulator. The network simulators INET, ns-2, GTNetS and ns-3 have a wide repository of components that the users can import and use in their simulations. As these simulators are open-source, they can easily be reached and, hence, they are widely accepted/used simulators in the computer networking literature. Our survey includes these four simulators due to reasons mentioned above. This section provides an overview on these network simulators.

ns-2: This is implemented in C++ and the simulator exposes a front-end API to the users. This front-end API is in oTCL language; therefore, the network model and the simulation control is implemented in this language. As oTCL is an interpreted language, this approach significantly reduces compilation time of the network models. However, it also reduces the performance and resource requirements of the simulator hampering the scalability of ns-2. ns-2's development was started in 1996 and, since then, it has been widely adopted by the computer networking community.

GTNetS: Georgia Tech network simulator (GTNetS) is devel-

oped to address the scalability issues of ns-2. In addition to this, GTNetS provides a better representation of computer networks by implementing a clear distinction between protocol layers. For example, the nodes in GTNetS are simple containers where the networking interfaces, protocols and applications are "installed." The server applications are bound to ports and the client applications connect to the servers using their address/port pairs. As these representations mimic actual computer networks, the users can implement network models easily. GTNetS is implemented in C++ and does not provide a front-end language for users: the simulator is a class library, and the users import this library to implement the network model.

INET: INET framework is the computer network simulation toolkit for the OMNET++ simulator. In OMNET++, the simulated objects are represented as *modules*. A module can be *simple*, which is an atomic object, or *compound*, which is the combination of other modules. The simple modules are, generally, implemented in C++. However, compound modules are implemented in the custom Network Description Language (NED). In a NED file of a compound module the external interfaces (e.g., its parameters), the submodules (i.e., other modules included) and the connections between the submodules are defined. Simple modules also include a NED file, which describes the interfaces the module exports to the NED environment.

The INET framework follows the OMNET++ architecture, where the simple modules implement the queues/protocols, and compound modules represent nodes/routers. The network model is also a compound module that contains nodes, routers and the connections between them. Hence, the user implements the network model in the NED language. The modules in INET are packaged according to the network layer they correspond to; when programming the network model, the user simply imports the protocols/applications from the package corresponding to a network layer.

ns-3: Ns-3 is developed to address the scalability, memory leak and extendibility problems of the previous network simulators. ns-3 is programmed in C++; the network models can be programmed in Python and C++. To reduce the footprint of the library, it is possible to turn off Python bridge completely. Ns-3 borrows most of its architecture from GTNetS, but improves it by adding managed memory and helper methods. To provide scalability, most of the design decisions of ns-2 are not carried to ns-3. Therefore, ns-3 is not compatible with ns-2 and models in one have to be manually ported to the other. Ns-3 development has started in 2005 and, since then, most of components of ns-2 have been ported to the ns-3. Hence, ns-3 also provides a rich set of networking interfaces, protocols and softwares that can be used in simulations.

4. EVALUATION OF NETWORK SIMULATORS FOR LARGE-SCALE NETWORK SIMULATIONS

This section evaluates the network simulators with respect to the features that can aid the users in the development of large-scale network simulations. We selected these features based on our experiences in developing distributed network simulations, extending the simulators and the requirements of the smart grid applications.

4.1 Transparency of Distributed Simulations

The setup of distributed simulations should be transparent to the user; that is, the implementation of the network model for distributed simulations should not differ from the those of single-process simulations. Otherwise, there might be a significant overhead in porting a large-scale network model from single process simulation to distributed simulation. In addition to this, factors such as the distribution of the network topology and the middleware used to handle the communication between the processes also affect the time needed to prepare the distributed simulation. Table 1 summarizes the comparison of network simulators with respect to how distributed simulations are programmed and represented.

Table 1: Distributed simulation support in network simulators

Network Simulator	Difference in simulation setup	Network topology in each node	Middleware
ns-3	•	Full	MPI
GTNetS	•	Full	LibSynk
INET	••	Partial	MPI
ns-2	••	Partial	LibSynk

• nodes are assigned to processes •• nodes and links are assigned to processes

GTNetS and ns-3 are designed to support the distributed simulation with remote links, which are point-to-point links for handling communication between nodes at different processes. To distribute a network model, the user has to assign the nodes and applications to the simulator processes. The initialization of the links and the routing protocols, however, are the same for both distributed and single process simulations. The full network topology is created at each process in distributed simulations. Memory wise this is not optimal, however, the simulator can calculate the routes to nodes at different processes without user intervention in this way.

Parallel Distributed NS [12] (PDNS) is an extension to ns-2 that adds the support for distributed simulations. Because ns-2 is not designed for such simulations, the network model implementation for distributed simulations differs significantly from single process simulations. For example, the user has to create remote links to connect nodes at different processes, assign IP addresses to remote links and add routing entries for remote nodes. These aspects are not required for single process simulations.

Although OMNET++ has extensive support for distributed simulations, INET framework was not designed for them. The user has to ensure that modules, such as flat routing, are not used when setting up a distributed simulation. In certain cases, this can lead to users assigning IP addresses and routing tables, which might increase the effort spent in implementing the network model.

OMNET++ and ns-3 use MPI [3] to handle the communication between simulator processes. As MPI is a common platform for clusters, setting up the distributed simulation environment for these simulators can be a straight forward process. GTNetS and ns-2 (with PDNS) depend on the Real Time Infrastructure library *libsynk* [10]. Although this library is specifically designed for distributed discrete event simulations, it is not commonly used. As such, setting up the distributed simulation environment for GTNetS and ns-2 might involve a learning curve.

In all simulators, connections between nodes at different processes has to be point-to-point links and the user has to manually partition the network model into processes. The partitioning of the network model, if not studied well, might lead to "unbalanced" simulations increasing the execution time of the simulator. For example, if the partition is realized over a link where there is too much packet exchange, the simulator processes might spent most of their time sending/receiving data. On the other hand, if most of the load is pushed to one of the processes, that process has to execute more events while others idling the resources.

4.2 Information Hiding

When programming a simulation, it is important that the programmer to be only concerned with networking concepts. Hence, the simulator should aid the user by hiding its implementation de-

Table 2: Information hiding in network simulators.

Network Simulator	Information hiding level	Front-end language
ns-3	●	C++/Python
GTNetS	○	C++
INET	●●	Ned
ns-2	●●	oTCL

○ no, ● partial, and ●● full information hiding

Table 3: The types of topology creators support the network simulators.

Simulator	Grid	Star	Dumbbell	Tree	Internet
ns-3	√	√	√	×	⊙
GTNetS	√	√	√	√	⊙
INET	×	×	×	×	⊙
ns-2	×	×	×	×	⊙

√ supported by the network simulator, × not supported by the simulator and ⊙ support through external tools.

tails from the user. Exposing unrelated API to the user might cause the users to program the simulations in an undesired way, which can increase the possibility of errors. The comparison of network simulators with respect to this feature is summarized in Table 2.

In GTNetS, the full API of the simulator is exposed to the user, as it does not provide a front-end language. Even though C++ allows information hiding, some accessible methods (e.g., public methods) can still be unrelated to the users. Thus, the user has to discover the correct way of initializing and using the classes representing the network appliances, which can introduce a significant learning curve. GTNetS comes with example network models that provide a good starting point. The user still has to refer to the API specification for complicated network models, which can be an overwhelming task due to the complexity of the simulator.

In both ns-2 and INET the network models are programmed in a front-end language. ns-2's front-end language is an API for the scripting language oTcl. The developers of the simulator explicitly states which C++ class methods and/or attributes can be accessed from oTCL scripts. This allows the developers to expose a simplified API to the ns-2 user, reducing the learning curve involved in developing simulations. The disadvantage of using a full-fledged scripting language is that the interpreter of the language is computationally expensive, which reduces the performance of the simulator.

As INET framework is based on OMNET++, it uses the NED language for programming the network models. NED is a domain specific language designed to express the connections and the compositions of the modules. Since it is not a scripting language, it does not suffer from the performance degradation as ns-2. In addition to this, OMNET++ provides a configuration file for setting up parameters specific to a simulation. The user can setup general parameters like the location of the nodes for animation in the NED file. However, parameters such as application transfer rate can be set in the configuration file. This way, the same network model code can be used with multiple configurations.

ns-3 provides two separate APIs: 1) The "front-end" API, which includes the *helper* classes that ease the initialization and the usage of the classes representing the network appliances. 2) The "back-end" API includes all the details of the simulator. In ns-3, the *helper* classes can only be called from user programs; that is, using them at the back-end is not permitted, and yields to compilation errors. However, the users are not limited to using the helper classes; they can use the back-end API for initialization, which can lead to long simulation model codes. The benefit of providing two APIs is that the network models can be implemented in the language the simulator is programmed with. Thus, the simulator does not need to provide a front-end language and suffer from the performance problems associated with the interpretation. Note that the front-end API of ns-3 has bindings for Python programming language, which allows the user to program simulations with Python. However, the aim of these bindings is not to hide implementation details (because C++ and Python use the same API) but to provide a platform for users not familiar with the C++ programming language.

4.3 Topology Generators

Programming the topology of a large-scale network can be a very time consuming and error prone task. The network simulators can assist the users in this task by providing topology generators. Topology generators are subroutines/tools that automate the creation of certain network topologies (e.g., n-by-m grid topology). Table 3 lists the topology generators supported by the network simulators.

ns-2 does not support generators for simple topologies (i.e., Grid, Star and Dumbbell); thus, most of the time the user has to manually program the desired topology. The support for more complicated internet-like topologies, on the other hand, are provided by external tools. These tools convert the output of the internet-like topology generator tools, like GT-ITM [14], to oTCL scripts. These tools, however, are not part of the simulator (i.e., not distributed with the simulator) and, hence, they might involve a learning curve.

INET also does not provide generators for simple topologies. There are external tools that generate NED files for more complicated topologies. Similar to ns-2, these tools are not part of the framework and involve a *steep* learning curve.

Both ns-3 and GTNetS have methods for generating simple topologies. In addition to this, they also provide methods for importing the output of the topology generator tools and internet traces. With these, the users can create topologies of their network models with one simple method invocation. Other topologies still need to be manually implemented.

For all simulators, it is possible to modify the generated topology to fit the needs of the network model at hand.

4.4 Lightweight Routing Protocols

For large simulations, the user will end up creating many nodes with multiple connections. However, managing the routing tables of these nodes with a progressive routing protocol can consume too many resources: the frequent route discovery messages slow down the simulator and generate too many traces. If the routing algorithm is not of importance to the network model, the user can greatly benefit if simulator provides a lightweight routing protocol. Table 4 shows the lightweight routing protocol(s) that can be used for large-scale simulations.

ns-2 provides two routing settings that can greatly reduce the overhead caused by routing protocols. The first setting is manually filling the routing tables. Although this method eliminates the time spent in route calculations, manually adding entries to routing tables can be a complicated task for large-scale simulations. The second setting is using *global* routing algorithms: *static* and *session*. The static routing algorithm calculates the routing tables for the nodes before the simulation, and during the simulation these tables are not updated. Since the whole network topology is known in a simulator, Dijkstra's shortest path algorithm is used for calculating the routes. The session routing algorithm is identical to the static routing but it recalculates the routes when the topology changes (due to link/node failures).

INET framework provides modules for all the routing protocols

Table 4: Lightweight routing protocols for large-scale network simulations

Network Simulator	Routing Protocol
ns-3	NIX-Vector
GTNetS	NIX-Vector
INET	link state, flat
ns-2	static, session

in the internet. From these, algorithms based on link state (e.g., *OSPF*) can be used in large scale simulations to reduce the overhead involved in the routing. In addition to this, the framework provides the *flat* routing algorithm, which is very similar to ns-2's static routing. The users can also fill the routing tables manually.

GTNetS supports all the routing settings supported by ns-2. GTNetS also supports the lightweight routing protocol NIX-Vector routing [11]. NIX-Vector routing algorithm resolves the issues related to routing in large-scale simulations by: 1) calculating the routes on demand 2) caching the routing entries.

In ns-3, NIX-Vector routing algorithm, global routing algorithm and manually filling the routing tables can be used to reduce the routing protocol overhead. As discussed before, manually adding routing entries can be complicated task. Therefore, using global routing or NIX-Vector routing algorithm are preferable. The global routing algorithm calculates the routes once before the simulation, similar to ns-2's static routing. It can also be configured to recalculate routes when a link/node state change occurs. This behavior is similar to session routing in ns-2.

4.5 Network Error Models

To assess the efficiency and the robustness of protocols/applications, it is important to test their behavior with respect the network failures. Hence, the type of failures the simulator supports is an important aspect.

ns-2, ns-3 and GTNetS support a wide variety of error models. Here, the users can schedule link/node failures, packet loss/errors at specific simulated times or based on random variables. Certain packets can also be marked to be lost at the beginning of the simulation. For large-scale simulations, this feature can reduce the effort in the analysis phase as the user can focus only on the behavior of the protocols/applications after/before the marked packets.

Contrary to other simulators, INET framework separates error simulation from the network model. The framework provides a *scenario* module, which reads an XML file. In this file, the user can schedule events that change parameters of the network model. Although the scenarios allow the user to control a variety of parameters during the simulation, the user has to learn how various errors can be simulated by changing these parameters. Moreover, the simulator does not provide random variables. The user has to generate the times based on the desired random distribution manually and schedule events at these times, which can be time consuming.

4.6 Network Model Evaluation

For a large-scale network simulation, it is important that the network model reaches a desired goal, such as bytes sent from a node. Otherwise, the results from the simulation might be inconclusive. The network simulator can help the user by providing facilities to monitor the status of the simulation, the simulated networked hardware and software. In this way, the user can evaluate the network model and change certain simulation parameters during the simulation.

ns-2 and ns-3 do not provide such monitors; hence, the user has the wait for the end of the simulation to analyze the trace and verify that the network model was correct (i.e., reached desired goals).

Table 5: Options for tracing the network models. For all simulators, the trace output analysis is realized by external tools.

Network Simulator	Trace Options	Animation support	Statistics gathering
ns-3	• ○ ⋆	Offline	√
GTNetS	•⋆	Realtime	√
INET	•	Realtime	√
ns-2	•	Offline	×

• packets de-/queued at the nodes, ○ packet capture for each node and ⋆ user defined traces

If the network model has failed to reach the goals, then the user has to alter the parameters of the network model and restart the simulation. This is an impractical process as large-scale network simulations tend to take a long time.

OMNET++, which INET framework is based on, and GTNetS allows the user to display the network model before the simulation. This can be used for verifying the connections of the network model. However, these simulators also do not provide means for the user to change the parameters according to the status of the network model.

4.7 Trace Analysis

The output from a network simulator should provide detailed information about what has happened during the simulation. Thus, the users can analyze and asses the performance of the protocols and the applications. Most network simulators provide such a detailed output with simulation traces and logging. For large network simulations, however, these can be too long and, as such, extracting information can be an overwhelming process. Here, the network simulator can greatly help the user if it allows the user to control the nodes and links that are to be traced. In addition to this, animation of the packets and statistics collected on packet delivery can also provide an overview on the interesting points about the simulation. Table 5 lists the features provided by the network simulators to aid the trace analysis.

In ns-2, the traces detail the packets de-/queued at the nodes. The user can trace all the queues of all nodes or create specific traces for the input/output queues at the ends of a link. These traces are saved to a custom trace file, a text file with special syntax. ns-2 does not provide tools for analyzing the trace files. However, standard regular expression tools (like *grep*) can be used. These trace files can also be animated offline (i.e. after the simulation completes) with the NAM (network animator) tool that is part of the simulator. The down side of animation is that it "pollutes" the network model code with animation setup code such as setting up node coordinates and color. This is especially problematic with large-scale network models, where the model code can be too long. Adding the animation setup code to the network model makes it even longer which, in turn, hampers the maintenance.

INET framework benefits from OMNET++'s user interface tools. These tools provide the user the ability to control the simulation (execute, pause, step), view the trace as the simulation executes, visualize the network, visualize the status of the simulated objects and animate the whole network or a part of it in realtime (i.e. while the simulation is executing). The animation setup is, usually, realized while programming the network model. Because the network model code is separated from the simulation parameters in OMNET++, the "pollution" problem described above is not present in INET framework. In addition to these, the simulator creates a trace file that contains information similar to the trace files of ns-2. The framework comes with modules to save the trace files in ns-2's trace

file format. Hence, tools developed for ns-2's trace analysis can be used with INET framework.

ns-3 and GTNetS provide an extensible tracing mechanism that uses a callback mechanism. The user can attach trace objects to various attributes at nodes and links. GTNetS includes an animator where the users can see the network trace in realtime. The animator can be programmed to only view the topology, start showing the trace a specific time in the simulation and/or selectively enable/disable the animation on the nodes. ns-3 provides an animation interface where the users can build their own animation frameworks. ns-3 comes with NetAnim tool, which is a prototype tool for this interface. NetAnim is a limited offline animation tool in that it can only animate networks with p2p links. In both simulations, the animation setup is scattered over the network model code which can complicate the model code significantly.

In ns-3, it is also possible to generate packet traces for each node in the packet capture format (*PCAP*). *PCAP* is a very popular packet capture format and many tools are available for analysis of these files. These files provide an expressive querying language. These traces are generated for all nodes, which might be hard to analyze for large network simulations.

INET framework, GTNetS and ns-3 have built-in statistics gathering functionality, which can generate histograms and cumulative distributions about incoming/outgoing packets. This statics gathering functionality is designed to be extendable, where the users can easily integrate other statistics gathering methods to the simulator. The statistics gathering is also realized in memory and the results are written to a file at the end of the simulation. This increases the performance of the simulator while increasing the memory usage. In large network simulations, statics gathering can ease the analysis process as they can aid in pointing out anomalies.

5. OPEN ISSUES ON NETWORK SIMULATION

From the evaluation in the previous section, we have identified the following open issues about network simulators that can aid in large-scale network simulation:

Lack of Network Model Verification: The network simulators do not provide means for altering the parameters of the simulation once the it has started. Hence, the user has to wait for the simulation to complete, verify manually that the network model reached the desired goals, and restart the simulation with altered parameters if the model does not reach the goal. This is an impractical and error prone task.

Lack of Load Balancing: The reduction in the execution time of the simulator with distributed simulation depends on the user's ability to partition the network model before the simulation. However, it is very hard to anticipate the loads at the links and the nodes. Therefore, the user might not be able to partition the network model efficiently, which causes an increase in the execution time of the simulator.

To illustrate the effects of unbalanced distribution, we created 10-by-10, 20-by-10, 40-by-10 grid models with 12, 14, 16 and 20 constant bit rate applications, and simulated these networks in ns-3 with 2 processes. Equal number of nodes is assigned to each process, where process 0 gets first n nodes and process 1 gets the nodes between n and $2n$. A node in each process executed the sink application; constant bit rate applications assigned to a process sent packets to the sink application of that process. The assignment of the applications are used for generating *balanced* and *unbalanced* networks. For example, with 12 applications, the unbalanced network is achieved by assigning 7 applications to process 0 and 5 applications to process 1. The balanced network is achieved by assigning 6 application to each process. Each constant bit rate application is directed to send UDP packets of 1000bytes every 0.4 seconds for 1000seconds to the sink node (NIX-Vector routing is used in the simulations).

Table 6 presents the execution time (in seconds) of the simulator for the balanced and unbalanced network models. Here, we can see that the balanced executions take much less time than the unbalanced ones. With 12 applications, the unbalanced 10-by-10 network executes 3.7 seconds more than the balanced version, whereas the unbalanced 40-by-10 network executes 5.7 seconds more. In the 40-by-10 network with 20 applications, the execution time of the simulator with the unbalanced network increases by 25.8 seconds. These results show that how balanced the distribution increases the execution time of the simulator. This increase gets worse as the network model gets larger. Hence, we conclude that the distribution of the network model plays a crucial role in the simulation of large-scale networks.

The problems described above can be addressed by introducing runtime steering to the network simulator. Runtime steering has been proposed as a way to monitor a program's execution and alter the execution if it does not reach the desired goals [7]. In summary, a runtime steering framework consists of monitors and steering rules. The monitors are attached to the program and they extract information about the execution. The steering rules, on other hand, contain goals over the monitored information and actions. These rules are, usually, expressed in a custom build language. If a goal is violated during runtime, the action corresponding to it is executed. This action alters the program such that monitored information falls within the boundaries of the goals.

Runtime steering for network simulators would allow the user to alter the network model when it does reach the desired goals. However, altering the simulator might lead to an erroneous simulation. For example, a packet might reach a destination before its predecessor when an alteration increases the speed of a link. To prevent such situations, the framework can be coupled with a model checker. The changes in the parameters of the network model that might lead to inconsistencies can be identified. These changes can be expressed as constraints in the language of the model checker. Hence, before the execution of the steering rules, the model checker evaluates whether the current status of the network model violates the constraints. If so, the model checker stops the steering.

6. CONCLUSION

In this survey, we evaluated the state-of-the-art network simulators, ns-2, INET, GTNetS and ns-3, with respect to the features that can aid the users in large-scale network simulations. Based on our experience in large-scale network simulations and the requirements from the smart grid applications, we identified *automated topology generators*, *information hiding*, *support for lightweight routing protocols*, *network error simulation*, *transparency of distributed simulations*, *evaluation of the network model during simulation* and *trace analysis* as features that can aid the users during the development, execution and analysis phases of the simulation.

From our evaluation, we conclude that there is room for improvement in network simulators for the simulation of large-scale network models. Especially, the simulators included in this survey have limited support for transparency of distributed simulations, information hiding and trace analysis: For distributed simulations, the user has to adapt the network model for distributed simulations: the nodes at different processes need to be connected with point-to-point links, and the user needs to ensure that network model is balanced. The lack of information hiding leaves the users with mul-

Table 6: Effects of unbalanced distributions to execution time

	10-by-10 grid		20-by-10 grid		40-by-10 grid	
	Execution Time (s)		Execution Time (s)		Execution Time (s)	
# of Applications	Balanced Model	Unbalanced Model	Balanced Model	Unbalanced Model	Balanced Model	Unbalanced Model
12	24.6	28.3	31	36.2	55.5	61.2
14	27	31.4	37.9	43.4	66	76.3
16	29.3	35.1	44	51.5	70.4	89.1
20	38.2	43.8	50.1	62.9	89.2	115

tiple ways of programming the network model; thus, the user might initialize the model improperly leading to errors during simulation. Too much information hiding, on the other hand, requires the user to learn a new language and/or hamper the performance of the simulator. After a large-scale network simulation, the users are left with huge trace files, and the network simulators rely on other tools for analysis. These can be greatly improved with the recent methods from high-performance computing and software engineering literature.

We also observed two open issues related to the usability of the simulators. The first one is that the simulators do not provide means for realtime evaluation of the network models. The programs for large-scale network models tend to be very long; as such, it is very hard for the user to ensure that the network model parameters are setup correctly or the network model meets the goals. Currently, this is realized by analyzing the trace files generated by the simulator. If the simulation does not reach a goal, the user has to alter the parameters and restart the simulation. This process is time consuming and error prone.

The second issue is related to distributed simulations. Here, the user has to partition the network model into processes. However, such partitioning can lead to an unbalanced distribution increasing the overall execution time of the simulator. To illustrate the effects of the distribution on the execution time of the simulator, we conducted execution time analysis with 2 process distributed simulations. We implemented 10-by-10, 20-by-10 and 40-by-10 grid network models where each process is assigned an equal number of nodes. Then, we added 12, 14, 16 and 20 applications to these network models. However, we assigned the applications such that 4 network models were balanced, equal number of applications is assigned to each process, and 4 were unbalanced, one process is assigned 5 applications and the remaining applications are assigned to the other process. On the 10-by-10 grid network, the execution time of the simulations with unbalanced models has increased on average by 4 seconds. This average has increased to 15 seconds for the 40-by-10 grid. These results show that how balanced the distribution is does indeed effect the execution time of the simulator.

Both of these open issues can be addressed by introducing runtime steering to network simulators. In this survey, we also discuss how a runtime steering framework can be provided to the network simulators. As future work, we plan to develop a steering framework with a model checker for network simulators.

7. REFERENCES

[1] The georgia tech network simulator (gtnets) http://www.ece.gatech.edu/research/labs-/maniacs/gtnets/. online, August 2011.

[2] Inet framework http://inet.omnetpp.org/. online, August 2011.

[3] The message passing interface (mpi) http://www.mcs.anl.gov/research/projects/mpi/. online, August 2011.

[4] The network simulator ns-2 http://www.isi.edu/nsnam/ns/. online, August 2011.

[5] The ns3 network simulator http://www.nsnam.org/. online, August 2011.

[6] I. Gorton, Z. Huang, Y. Chen, B. Kalahar, S. Jin, D. Chavarría-Miranda, D. Baxter, and J. Feo. A high-performance hybrid computing approach to massive contingency analysis in the power grid. In *E-SCIENCE '09*, pages 277–283, Washington, DC, USA, 2009. IEEE Computer Society.

[7] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, 1998.

[8] V. Gungor, B. Lu, and G. Hancke. Opportunities and challenges of wireless sensor networks in smart grid. *IEEE Trans. on Industrial Electronics*, 57(10):3557 –3564, oct. 2010.

[9] M. Korkalainen, M. Sallinen, N. Karkkainen, and P. Tukeva. Survey of wireless sensor networks simulation tools for demanding applications. In *ICNS '09*, pages 102 –106, april 2009.

[10] K. S. Perumalla, A. Park, R. M. Fujimoto, and G. F. Riley. Scalable rti-based parallel simulation of networks. In *PADS '03*, pages 97–, Washington, DC, USA, 2003. IEEE Computer Society.

[11] G. Riley, M. Ammar, and E. Zegura. Efficient routing using nix-vectors. In *HPSR '01*, pages 390 –395, 2001.

[12] G. F. Riley, M. H. Ammar, R. M. Fujimoto, A. Park, K. Perumalla, and D. Xu. A federated approach to distributed network simulation. *ACM Trans. Model. Comput. Simul.*, 14:116–148, April 2004.

[13] C. P. Singh, O. P. Vyas, and M. K. Tiwari. A survey of simulation in sensor networks. In *CICA '08*, pages 867–872, Washington, DC, USA, 2008. IEEE Computer Society.

[14] M. Thomas and E. W. Zegura. Generation and analysis of random graphs to model internetworks. Technical report, College of Computing, 1994.

[15] E. Weingartner, H. vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *ICC '09*, pages 1 –5, june 2009.