Internet-Based Front-End to Network Simulator

Taosong He

Bell Laboratories
taosong@research.bell-labs.com

Abstract. We present a Java-based interactive visual interface to network simulators. Having successfully incorporated network visualization technologies, our system provides an effective front-end interface supporting real time control and monitoring of the back-end simulator through the Internet or intranets. We extend a traditional spring embedding graph layout model, and propose a new hierarchical node placement algorithm for presenting the structure of a complex network.

1 Introduction

Network simulators are the tools that enable users to simulate the operations and behavior of a network. For network designers and managers, these tools provide a cost-effective way of evaluating a network under different scenarios. This project focuses on a proprietary Lucent Technologies network simulator for switching-based voice networks. By manipulating the network parameters such as routing, topology, and traffic through the simulator, users would be able replicate an existing network, simulate different kinds of nodes/links failures and observe their impact, analyze the effectiveness of certain network management control, and optimize their network designs.

Network simulation is a computational intensive process. For this and other proprietary and security reasons, our simulator is implemented with C and running on a back-end multi-processor server. To support the simulator, it is of crucial importance to develop an interactive visual front-end system. Typically, users of our system range from the network designers working on high-end graphics workstations to the marketing managers working on their laptop PCs. They would have to be able to access and control the remote simulator from their local machines in *real time* through an *intranet* or the *Internet*. To satisfy these requirements, we have developed a Java-based interactive visual interface as the front-end to network simulators. We have chosen Java to implement our system for its free portability, strong support of network programming, and object-oriented environment. The main contributions of our system include a generic Java-based framework for designing network visual interfaces, as presented in Section 2; a new graph layout algorithm for interactively presenting hierarchical structure of complex networks, as introduced in Section 3; and the discussion of some issues and lessons related to the development of such an Internet-based application, as presented in Section 4.

2 Java Framework for Network Interfaces

Developing a network visualization system usually requires programmers to start from scratch and deal with all the issues such as handling of network data, rendering, and interactions. To address this problem, we have developed a generic framework for network interface development[5]. In this paper, we extend the framework from C++ in [5] to Java and enhanced it with networking capability. The new framework provides the tool developers an OO and Internet-based programming environment for the easy implementation of application dependent network visual interfaces. The main idea is to abstract the most fundamental functionalities of all the network views, and encapsulate them into several independent "building blocks". Some of the major building blocks include:

Network Data This building block handles the acquisition, storage, and communication of network data. Traditionally, a network is defined as a set of nodes and links. To handle the more general cases, we adopt the tuple-based network data representations of [5] where a network is defined as a set S of $n-tuples$: $(a_0, a_1, ..., a_n), a_i \in S, n \geq 0$. Each data entry (a tuple) in a network is associated with static dynamic attributes such as trunk capacities or current traffic volume on a switch. This building block supports some basic data operations including entry create/delete, information retrieve/modification, and message receive/send from/to other building blocks. The new extensions to [5] include a JDBC-based class for querying the possibly remote databases storing network configuration information; and a communication component for propagating data update messages between remote clients. Compared to the more commonly used table-based representation of network data, our tuple-based representation supports a unified handling of heterogeneous network data. It is more also flexible for dealing with edit operations such as delete, add, and connect.

Interactive Operations This building block supports three classes of operations: *selections*, *display control*, and *data* operations. The purpose of *selection* is to identify a set of display entities for a future operation, such as copy or delete. A more detail discussion of different *selection* operations can be found in [7]. *display control* operations are used to control the rendering environment, including how to display each individual element on the screen. They can be classified into *visual* control (highlight, visibility, and glyph property change) and *coordinate* control (scale, move). *Data* operations are used to modify the underlined network data, which could in turn change the display of the network. They include interactive editing of network topology such as create, delete, copy, and paste; and modification of data attributes such as trunk capacity. Data operations in this buidling block invoke the correspondent functions in the *network data* class discussed above.

Most of the views in our system for the displaying of networks are based on the traditional node and link diagram [2]. A main problem of such a diagram is that the display can be cluttered when the number of nodes or especially links is large. To address this problem, our display control operations also

support some standard clutter reduction methods including transformation, merging, fisheye[6], and interactive filtering (subnet, focus) [1]. We have also invented during this project a new clutter control operation: *aggregate*. The basic idea of "aggregate" is to recursively merge the network elements by their types and create "virtual" elements with user-defined aggregated attributes in a network. Our experiments have shown that as a generic network visualization tool, this operation is very effective for reducing the display complexity, while presenting a *skeleton* structure of a complex network.

View Linking At a top level, a network visualization system consists of a set of views presenting from different perspectives the status of a (dynamic) network. One of the main features of our system is that all the views are appropriately linked together through this building block, based on interactive operations such as selection, focus, and interactive editing. When an operation is performed in one view, it is passed to a view linking engine together with its parameters. Based on the type of the operation, the engine sends out the appropriate view link messages to all the corresponding views. In our framework, we have designed a three level view linking mechanism. At the *local* level, all the operations are restricted to the current display without propagating to other views. Most of the rendering control operations such as zooming abd grouping are by default local. The second view linking level is the *display* level where an operation propagates to all the other views that are displaying the same data. Selection and brushing fall into this level. An interesting feature of our framework is that linking between any two views at this level can be interactively turned off by users. The highest level of view linking is the *data* level. Similar to those at the *display* level, operations at this level affect all the related views. The difference is that a data level view link can not be turned off. Examples include editing operations such as create or delete.

Other important building blocks include *rendering* and *glyph design*. Each building block is implemented as a Java "interface". Since Java does not support multiple inheritance, our framework also includes an implementation of a $BaseView$ class for gluing all the building blocks. To develop a new view, programmers could simply inherit from the BaseView, and overload some of its functions. All the views in our front-end system are implemented within this framework. One of the biggest advantages of the framework is that each view developed based on it automatically supports generic network data and a rich set of interactive operations. Our experiences have also demonstrated that by using this framework, tool development time can be dramatically reduced.

3 Network Layout

The back-end network simulator in this project is designed to simulate two-level switching-based voice networks. The higher level, or the backbone, of this kind of networks is usually a fully meshed network deployed with several types

of switches such as service nodes and mobile service centers. The lower level consists of switches configured as local exchanges (LE). Each switch is located within a Local Calling Area (LCA), and trunks with different capacities are used to connect the switches. The main view of the front-end is a network editor; cf bottom left of Fig. 1. In this figure different kinds of switches are represented by different icons, and trunks are represented by blue lines. A user can modify a network with a variety of interactive operations as discussed in Section 2 through the element icon platter.

Network layout is essentially the most important task for network visualization. With the help of background maps, geographical-based layout is a straightforward yet powerful visual metaphor. Our system supports both polygon and image based maps. On the other hand, logic layout presenting hierarchical network structures or dynamic simulation patterns are usually more important for users. Particularly, our clients have identified two requirements for a logical view. First, all the switches within the same LCA need to be put into the same area on the screen (cluster problem). Second, within each LCA, switches with stronger task dependent relationships needs to be put closer to each other (standard node placement problem [8]). To satisfy these requirements, we have developed a new algorithm based on the classical spring embedding model (SEM) [4].

In an SEM, nodes are considered as mutually repulsive charges and edges (relationships) as springs attracting connected nodes. Starting from an initial layout, the nodes will move along the force directions until achieving minimal total energy. The force F is defined as the sum of repulsion and attraction forces:

$$F(n,m) = F_{att} + F_{rep} = \lambda_{att} \Delta(n,m) ||\Delta(n,m)||^2 - \lambda_{rep} \frac{\Delta(n,m)}{||\Delta(n,m)||^2} \qquad (1)$$

where $\Delta(n,m)$ is the distance vector between two nodes n and m. An edge (n,m) is at equilibrium if $F(n,m) = 0$. Unfortunately, classical SEM approaches can not be directly applied here since the repulsion and attraction coefficients λ_{rep} and λ_{att} are usually assigned based on physical models. They do not generate results satisfying the node placement requirements [8].

Our contribution is to prove that an SEM can be extended to provide an effective node placement solution. Mathematically, assuming that the weight on (n,m) is w, we simply assign $\lambda_{att} = w^4$ and $\lambda_{rep} = 1$. Edge (n,m) will then be at balance when $||\Delta(n,m)|| = 1/w$, exactly as required. Wills[8] has pointed out that for a good node placement solution, higher weight edges have to affect the layout output more than those more irrelevant edges. To prove this, a small perturbation is added to the optimal solution, $||\Delta(n,m)|| = 1/w + \epsilon$, and resulting:

$$F_{att}(n,m) + F_{rep}(n,m) = \Delta(n,m) w^2 [(1+w\epsilon)^2 - \frac{1}{(1+w\epsilon)^2}] \qquad (2)$$

As desired, Equation 2 illustrates the importance of stronger edges. Compared to NicheWork[8] and simulated annealing [3], our method is simpler, usually faster, and generates consistent results.

To solve the cluster problem, the classical SEM is extended so that a node is represented by a region instead of a point. $\Delta(n, m)$ is consequently defined as the shortest distance between region n and m. Starting from a random initial layout, Equation 1 guarantees that no regions intersect each other in the final layout. Our complete algorithm is therefore a two-step process. First, different LCA regions, represented by circles for simplicity, are placed using the extended SEM. To better use the screen estate, the area of each circle is proportional to the total number of switches in the correspondent LCA. The edge weights between different LCAs are task dependent. Second, switches within each LCA, represented by zero radius circle, are placed within the area.

Fig. 1 presents a logic layout of a network with 324 switches located within 12 LCAs. The bottom left displays the overall structure of the network. Inside each LCA, switches are placed by assigning higher weights to relations between an LE and its *toll home*. That is, the long distance traffic through an LE, represented by a smaller icon, has to be first routed through a "toll home" backbone switch before reaching other LCAs. The tree structured homing information is clearly presented in the overall view. In the bottom right of Fig. 1, we have focused on a specific region, and demonstrate the effectiveness of our algorithm by verifying that the outlier LE does not have toll home information. Our clients have reported that this is due to database incompleteness.

4 Implementation and Discussion

Our system supports real time visualization through a variety of monitoring and analytical tools such as histogram and table view. Fig. 2 presents a snapshot during a simulation run. In the main window, colors of switches and trunks are encoded to represent the current states of equipments, where red indicates emergencies, green for normal, and other colors for in-between situations. The dynamic strip charts present real time statistic information on certain parameters interactively demanded by users through the front-end. An animation control tool with full play-back functionality allows users to interactively examine simulation results at any time from the start of a simulation to the current time.

There are several important issues related to the development of such a real time Internet-based application:

Synchronization Two kinds of information generated by the back-end simulator need to be displayed in the front-end: statistics information interactively required by users such as number of calls on certain switches or trunks; and "alarm" information such as node/link failure. Since alarm messages require immediate attention, an independent thread with infinite loop is applied to continuously update the network views. To synchronize the statistical information, a "simulation clock" message is sent from the simulator every 10 secs to update the display.

Bandwidth To simulate a large network such as that displayed in Fig. 1, the back-end simulator could generate huge amount of log data. Fortunately, not all of them need to be sent to the front-end in real time through the socket

connection. For example, statistical information are generated every 10 secs and only the demanded parameters are sent back. In the mean time, limited by the screen size, a user can not simultaneously monitor too many strip charts. There could be a high number of alarm messages, especially when some key nodes and links fail. However, the length of each message is very short. Our experiences have shown that we can run the system through a 28.8Kb modem across the states with US, and through an intranet across the continents. In general, we suggest that current network bandwidth is enough for many non-graphica; real time applications.

Speed Not surprisingly, our system bottle-neck is the interactive rendering of large networks, especially those with large number of trunks. Generally, the performance of Java is not as good as native C++, and sometimes causes delay on low-end laptops. We expect this problem to be greatly alleviated with the advance of graphics card and Java implementations.

Security Our system can be run both as an application and as an applet through Netscape. We have applied the netscape.security package and digital ID to guarantee the security through browser.

5 Conclusions

In this paper we present an Internet-based visual interface to a switching-based voice network simulator. Implemented with a generic Java framework for developing network visualization package, our system allows users to interactively create and modify a network, remotely control the network simulation, and visualize simulation results with a variety of tools in real time. We have also proposed an innovative hierarchical network layout algorithm.

We are currently investigating connecting the front-end system to other kinds of network simulation and management systems. We have also finished a prototype of incorporating 3D views through VRML and Java3D into our system.

References

[1] R. Becker, S. Eick, and A. Wilks. Visulizing network data. *IEEE Transcations on Visualization and Computer Graphics*, 1(1):16–28, March 1995.

[2] J. Bertin. *Graphics and Graphics Information Processing.* Walter de Gruter & Co., Berlin, 1981.

[3] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Transactions On Graphics*, 15(4):301–331, 1996.

[4] P. Eades. Heuristic for graph drawing. *Congressus Numerantium*, pages 149–160, 1984.

[5] T. He and S. Eick. Constructing interactive network visual interfaces. *Bell Labs Technical Journal*, 3(2):47–57, April 1998.

[6] M. Sarkar and M. Brown. Graphics fisheye view. *CACM*, 37(12):73–84, 1994.

[7] G. Wills. Selections: 524288 ways to say "this is interesting". *Information Visualization'95*, pages 54–60, October 1996.

[8] G. Wills. Nicheworks - interactive visualization of very large graphs. *Graphics Drawing'97*, 1997.