Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCS-97-25

1997-01-01

Cappucino: An Extensible Planning Tool for Constraint-based ATM Network Design

Inderjeet Singh and Jonathan S. Turner

Cappuccino is a planning tool for topological design of ATM networks. It uses a novel constraint-based approach to ATM network design. Extensibility of the tool is a basic design goal and the tool provides an open interface to incorporate new algorithms.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation

Singh, Inderjeet and Turner, Jonathan S., "Cappucino: An Extensible Planning Tool for Constraint-based ATM Network Design" Report Number: WUCS-97-25 (1997). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/441

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

Cappuccino: An Extensible Planning Tool for Constraint-based ATM Network Design

Inderjeet Singh

Advisor: Jonathan S. Turner

WUCS-97-25

May 16, 1997

Department of Computer Science
Campus Box 1045
Washington University
One Brookings Drive
Saint Louis, MO 63130-4899

Abstract

Cappuccino is a planning tool for topological design of ATM networks. It uses a novel constraint-based approach to ATM network design. Extensibility of the tool is a basic design goal and the tool provides an open interface to incorporate new algorithms

Table of Contents

ATM Network Design 7
1. Introduction 7
2. Network Design Problem 8
2.1 Local access network design problem 8
2.2 Backbone network design problem 8
3. Issues in ATM Network Design 9
3.1 Ensuring low blocking probability 9
3.2 Ensuring low delays 9
3.3 Calculating link cost 9
4. Traffic Model 9
5. Decaf 10
Using Cappuccino 12
1. Network Design Using Cappuccino 12
1.1 Designing Network Topology 12
1.1.1 Creating Topology Visually 12
1.1.2 Automatic Creation Using Algorithms 14
1.1.3 Improving Topology by AdjustInitialNetwork Algorithm 14
1.2 Specifying Traffic Constraints 14
1.2.1 Creating Constraints Visually 14
1.2.2 Using ConstraintsWindow 15
1.2.3 Using Algorithms to define Constraints 15
1.3 Choosing Routing Algorithms 15
1.4 Dimensioning Links 15
2. Using Super Algorithms 15
3. Using Algorithms 16
4. GUI Features 16
4.1 HTML based Help 16
4.2 Log-in Screen 16
4.3 File Operations 16
4.4 The *Edit* Menu 17
4.5 The *View* Menu 18
4.6 Menu-options for Running Algorithms 19
4.7 The Options Menu 19
4.8 The *Window* Menu 20
4.9 The *Help* Menu 21
4.10 The Built-in Java IDE 21
4.11 Synchronized GUI Objects 23
5. Differences in the Applet and Application Version 23
Extending Cappuccino 24
1. Software Architecture 24
1.1 Guiding Rules and Design Principles 24
1.2 Applet as well as Application 24
1.3 Transparent File Handling 25
1.4 Incorporating Help 25
1.5 Showing Help Hints 25
1.6 Handling Error Conditions 25

1.7 Transient and Persistent Properties 26
1.8 Algorithm Representation 27
1.9 Document/View/Controller Approach 27
2. Package Structure 27
3. Understanding the *network* Package 28
3.1 Network Representation 28
3.1.1 The Class Switch 28
3.1.2 The Class Link 28
3.1.3 The Abstract Class Network 28
3.2 Constraint Representation 29
3.2.1 The Class SetPairConstraint 29
3.2.2 The Class Constraints 30
3.2.3 The Interface NetworkProperties 30
3.3 Interface *NetworkView* 30
3.4 Classes for Algorithm Parameters 31
3.4.1 The Class Param 31
3.4.2 The Class AlgoParams 31
3.4.3 The Class ParamsUnavailableException 32
3.4.4 The Class ViewProperties 32
3.5 The Class *NetworkUtils* 32
4. Writing Topology Algorithm 32
5. Writing Routing Algorithms 34
5.1.1 The Class ODPair 34
5.1.2 The Class RoutingInfo 35
6. Writing Dimensioning Algorithms 35
6.1.1 The Class DimensioningInfo 35
7. Writing Constraints Algorithms 35
8. Writing Super Algorithms 35
8.1 Understanding the *general* Package 36
8.1.1 The PropertiesHolder Interface 36
8.1.2 The Class Debug 36
8.1.3 The Abstract Class FileHandler 36
8.1.4 The Class HelpEngine 37
8.1.5 The Class CGI 37
8.1.6 The Class ObserveeTracker 37
8.1.7 Miscellaneous Utility Classes 38
Performance Evaluation and Project Statistics 39
1. Performance Evaluation 39
1.1 Efficient Network Representation 39
1.2 Algorithm Animation 39
1.3 Hiding Background 39
1.4 Observable Objects 39
1.5 Cappuccino Loading Time 40
1.6 File I/O over Internet 40
1.7 Storing the Network as Objects 40
1.8 Compilation in Java IDE 40
1.9 Optimizing the Best Star Algorithm 40
2. Key Contributions 40
2.1 Extensibility without Recompilation 41
2.2 Availability as an Applet and an Application 41
2.3 Document/ View/ Controller Approach 41

2.4 Web based Service Model ... 41
3. Project Statistics ... 41
4. Future Work ... 42
4.1 Adding New Algorithms ... 42
4.2 Cappuccino as a Trusted Applet ... 42
4.3 A Configuration Tool for Cappuccino ... 43
4.4 Using JAR to Package Class Files ... 43
4.5 Support for Adding New Cost Models ... 43
4.6 A Dialog Box for Algorithm Management ... 44
4.7 Allowing Enumerated Types as Property Values ... 44
4.8 Providing Selection Rectangle ... 44
4.9 Numerous Performance Enhancements ... 44
Credits and Acknowledgments ... 45
Appendix A: References ... 46
Appendix B: Glossary ... 48
Appendix C: Cappuccino Packages ... 49
1. The Default Package ... 49
1.1 The Main Classes ... 49
1.1.1 CappuApplet ... 49
1.1.2 CappuApplication ... 49
1.2 The Concrete File Handlers ... 49
1.2.1 AppletFileHandler ... 49
1.2.2 ApplicationFileHandler ... 49
1.3 The Concrete Help Engines ... 49
1.3.1 AppletHelpEngine ... 49
1.3.2 ApplicationHelpEngine ... 49
1.4 The Concrete Compiler Classes ... 50
1.4.1 AppletCompiler ... 50
1.4.2 ApplicationCompiler ... 50
1.5 Pre-caching classes ... 50
1.6 Classes Defining Properties ... 50
1.6.1 AppletProperties ... 50
1.6.2 ApplicationProperties ... 50
1.7 Concrete Classes for File Dialog ... 51
1.7.1 AppletFileDialog ... 51
1.7.2 ApplicationFileDialog ... 51
2. The *general* Package ... 51
2.1 Class *Debug* ... 51
2.2 Class *HelpEvent* ... 52
2.3 Class *HelpHintEvent* ... 52
2.4 Interface *HelpHintListener* ... 52
2.5 Interface SourceCodeCompiler ... 52
2.6 Class *ChangeCommand* ... 52
2.7 Exception Classes ... 52
3. The *network* Package ... 53
3.1 Class *NetworkImpl* ... 53
3.2 Class *ViewInfo* ... 53
3.3 Class *ConfigBean* ... 53
4. The *stk* Package: A GUI Toolkit ... 53
4.1 Revised versions of GJT classes ... 53
4.1.1 Border ... 53

4.1.2 ButtonPanel 53
4.1.3 ColumnLayout 54
4.1.4 RowLayout 54
4.1.5 DrawnRectangle 54
4.1.6 EtchedBorder 54
4.1.7 EtchedRectangle 54
4.1.8 Etching 54
4.1.9 ImageCanvas 54
4.1.10 MessageDialog 54
4.1.11 Orientation 54
4.1.12 Separator 54
4.1.13 QuestionDialog 55
4.1.14 YesNoDialog 55
4.2 New Classes in *stk* Tool-kit 55
4.2.1 PowerLayout 55
4.2.2 Circle 55
4.2.3 ExclusiveImageButtonPanel 55
4.2.4 ExclusiveMenu 55
4.2.5 ImageButton 55
4.2.6 ImageButtonPanel 55
4.2.7 StatusBar 55
4.2.8 StatusBarController 55
4.2.9 Util 55
5. The *gui* Package 56
5.1 GUI Entities 56
5.1.1 GUIManager 56
5.1.2 ViewInfoManager 56
5.1.3 IOGUIManager 56
5.1.4 AlgoGUIManager 56
5.1.5 ClipboardManager 56
5.1.6 ConfigBean 57
5.2 Common GUI Components 57
5.2.1 Menu items for Algorithms 57
5.2.2 Class ConstraintsWindow 57
5.2.3 FlexibleDialog 57
5.2.4 InputController 58
5.2.5 SwitchInfoDialog 58
5.2.6 LinkInfoDialog 58
5.2.7 PropertiesDialog 58
5.3 Visual Representation of the Network 58
5.3.1 NetworkGUIView 58
5.3.2 GUIViewController 59
5.3.3 AlgoExecuter 59
5.3.4 GUIViewManager 59
5.4 Textual Representation of a Network 60
5.5 Class *NetworkSwitchView* 61
5.6 Class *NetworkLinkView* 61
6. The *editors* Package 61
6.1 Integrated Development Environment for Algorithm Creation 61
7. The *TopologyAlgos* Package 62
7.1 *RandomSwitchAdder* 62

7.2 *RandomLinkAdder* 62
7.3 *CompleteNetwork* 62
7.4 *LinkComplement* 62
7.5 *DelaunayTriangulation* 63
7.6 *MinimumSpanningTree* 63
7.7 *Star* 63
7.8 *SetPrimaryNetwork* 63
7.9 *AdjustPrimaryNetwork* 63
8. The *ConstraintsAlgos* Package 64
8.1 The Class *PairwisePercentage* 64
8.2 The Class *Localized* 64
9. The *RoutingAlgos* Package 65
9.1 *ShortestPathRouting* 65
9.2 DistributedRouting 65
10. The *DimensioningAlgos* Package 65
10.1 *UseLinearProgram* 65
11. The *common* Package 66
11.1 The Class *DisjointSetElement* 66
11.2 The Classes *ShortestPathUtils* and *ShortestPathInfo* 66
12. The *SuperAlgos* Package 66
12.1 The Class *BestStar* 66
12.2 The Class *LowerBound* 67
13. The *SwitchCostAlgos* Package 67
13.1 The Class *LinearSwitchCost* 67
14. The *LinkCostAlgos* Package 67
14.1 The Class *LinearLinkCost* 67

ATM Network Design

1. Introduction

Deployment of any communication technology raises the issue of network design. Stated in its most basic form [Af94], the problem is to find a way to construct a network that meets the "desires of network users," and does so as cheaply as possible. Telephone networks were the first communication networks to be widely deployed. The "desire of network users" here was to have point-to-point connectivity at a fixed date rate and with a low probability of blocking. Narrow band ISDN networks also fall in this category. The success of the ARPANET brought forth packet switched communication networks. Here there were no circuits, hence blocking of connection attempts was not a consideration. Instead, packet delay was the main source of users' dissatisfaction. Starting in the late 70's, attempts were being made to integrate voice and data services. ATM is an emerging set of network standards that supports many kinds of services. An ATM network is a connection oriented network, which supports multi-point and multi-rate connections. The "desire of network users" here is to have point-to-point/ multi-point connectivity at a desired data rate with a low probability of blocking and a low network delay.

Any communication network needs to address issues of reliability and fault tolerance. Communication networks also undergo evolution due to changing needs of end users and hence it is important to find ways to upgrade a given network less expensively.

The optimal network design problem, even in its most simple forms, is a difficult problem to solve. Many of the involved sub-problems have been proven to be NP-Complete [ASJ96, ARSJ96] thereby precluding development of a tool which will give the cheapest network (for a moderate number of endpoints) while meeting the "desires of network users" in a reasonable time. A large number of heuristics are available that work well in a limited number of cases. The network planner, therefore, has to intelligently evaluate different heuristics and use his own expertise to design a network that meets the needs. It is also sometimes possible to compute a lower bound on cost of the network for a given network topology and traffic requirement. Such an estimate is often useful to determine when a particular solution is good enough.

This project is an attempt to build a tool to aid network planners. The goal is to provide an integrated design environment, called Cappuccino, in which a network planner can specify his network requirements and then apply a rich set of heuristics to create a "good" network. Similar tools are now publicly available (e.g., Planning workbench from Bellcore, PLANQUEST from NEC, etc.). Cappuccino differs from the existing tools in two main respects. First, it is based on a constraint based approach to network design that yields networks that do not block connection requests, so long as the offered traffic operates within a set of user-specified constraints. Second, it is designed to be extensible, so that new network design heuristics can be easily incorporated within the tool. In addition, it is being designed for use over the Internet (using the java programming language) making it easy for people to try it out and use it for developing network designs.

2. Network Design Problem

The network topological design problem can be formulated as follows [BF77]:

Given

- user terminal locations
- traffic requirements (source and sink capacities) of user terminals
- delay requirements
- reliability requirements
- candidate sites for switches
- link and switch cost functions
- An algorithm for routing traffic for connections

Total cost of communication network, C = access link costs + backbone link costs + switch costs

The objective is to minimize C such that the traffic, delay and reliability requirements are met.

Often, it is easier to break-up the global design problem into two parts, design of the local access network and design of backbone network.

2.1 Local access network design problem

Given several user terminals at specified locations, we want to find a location to install a switch that connects to all terminals and meets their traffic requirements. The combined outgoing traffic from the terminals is called the source capacity of the switch and combined incoming traffic is called the sink capacity of the switch.

For the local access problem, intra-network delays are often insignificant. Reliability requirements are also usually not stringent because the failures are typically due to faulty terminals or line access cards and hence affect only a single user.

2.2 Backbone network design problem

Given several switches at specified locations, we want to know how much capacity to install between each pair of switches such that the resulting network meets the blocking probability requirements for a given routing algorithm. We call this the link dimensioning problem.

End to end delays can be high due to large geographical distances and relatively large numbers of intermediate hops. Reliability requirements are also usually more stringent because a single failure can affect a large number of terminals directly or indirectly. Typically, the reliability of switches is high but that of line access cards and links is low. Also, repair time for link faults can be high. Hence, some kinds of fault tolerance guarantees (e.g., carrying a definite amount of traffic in the event of a single fault) are often required.

3. Issues in ATM Network Design

3.1 Ensuring low blocking probability

The connection oriented nature of ATM requires the network planner to ensure low blocking probability for valid connections. A valid connection is defined as a connection that is possible from the end user's point of view, i.e., all the endpoints involved in the connection have the required bandwidth available. It is possible to design a network that is fully non-blocking by designing the network for the worst case traffic distribution. However, significant cost savings can result by judiciously sharing resources if sometimes the connections are allowed to block. Standard bodies usually specify a minimum desired blocking probability and network planners must ensure that no valid connection in the network suffers a higher blocking probability than the specified maximum.

3.2 Ensuring low delays

Use of larger (in number of ports) switches lowers delays because of the reduced number of hops required to connect endpoints. Also, use of higher rate links implies a lower data transfer delay compared to that of lower rate links. To ensure low queuing delays, links cannot be loaded to their full capacity. The usable bandwidth of a link is often calculated by assuming a traffic arrival model and then performing queuing analysis [Jt01] to arrive at a rough figure (e.g., 75%). All link capacities available for network design are reduced by this factor.

3.3 Calculating link cost

When there are several different types of links available (e.g., OC-1, OC-3, etc.), the higher capacity links generally offer a lower cost per unit capacity than smaller capacity links. This advantage is magnified by fragmentation [Jt01], which has a more severe impact on small capacity links, and by long distances, since the cost of the fiber required for high capacity links differs little from that required for low capacity links. Consequently, whenever, there is sufficient traffic available to make good use of high capacity links, it makes sense to use them.

A linear link cost function is often simple to use. In this case, the cost of a link varies linearly with the capacity required on the link. A more realistic link cost model [ARSJ96] is a step or staircase cost function. This is obtained by calculating the optimum mix of different link types for a given capacity. Obtaining this function is a variant of the well-known knapsack problem and hence is NP-hard. However, the presence of only a small number of link types allows use of a dynamic programming solution.

4. Traffic Model

Generally, the desires of the network users are represented as magnitudes of traffic from a particular network node to other nodes. These are defined as a traffic matrix, which is a table in which each row corresponds to the traffic originating at a particular node, and each column corresponds to the traffic received at a particular node. Generating such a table can often be difficult or even impossible. For example, in an environment where traffic is primarily world wide web traffic, it is possible to specify the

amount of traffic a workstation receives but it is very hard to a-priori fix the source of that traffic. Most of the available network design tools currently use this approach for traffic specification.

In our traffic model [Af94], each node has an associated source and sink capacity for the traffic. The source capacity of a node is defined as the amount of the traffic it generates and its sink capacity is defined as the amount of the traffic it can receive. The connection pattern of the nodes is specified in terms of traffic constraints.

Different types of traffic constraints are possible [ASJ96, ARSJ96].

1. Flat constraints: Traffic is constrained only by the source and sink capacities.

2. Hierarchical clustering constraints: Clusters of switches are defined and intra-cluster and inter-cluster flat traffic constraints are specified.

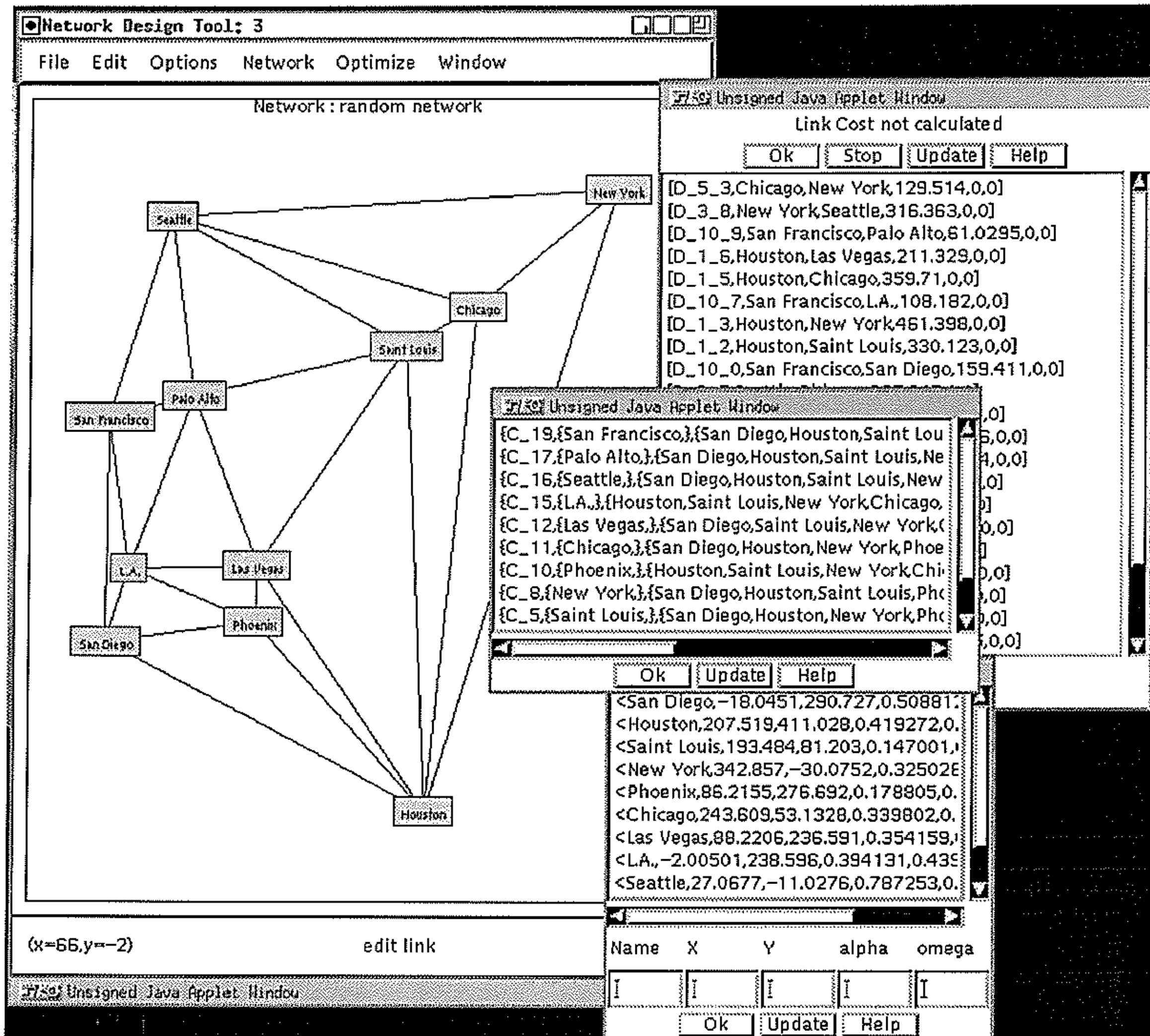
3. Distance constraints: Amount of traffic a node sends to another node is a (decreasing) function of the distance between the two nodes.

4. Node-set pair constraints: For any pairs of vertex sets, A and B, an upper bound on traffic flow between them is specified. This approach is most general and can be used to represent the flat, hierarchical and distance based constraints.

We are using the node-set pair constraints based approach for traffic specification in our tool. Nodes can be grouped in sets and these sets can share nodes. The bi-directional traffic flow between pairs of these sets specifies a node-set pair constraint. The tool will also allow convenient specification for the common cases.

5. Decaf

An initial version of Cappuccino (called Decaf) is available as a java applet at http://www.cs.wustl.edu/~javagrp/network-design-tool.html. It was implemented by Hongzhou Ma and this author with help from Rob Jackson and Andy Fingerhut. Decaf provides support for manual network creation. It also allows automatic creation of some simple topologies (e.g., best star network, minimum spanning tree, complete graph and Delaunay triangulation). The tool can also merge two different topologies to provide a "better" topology. The tool provides support for traffic specification. The user can specify local and remote traffic constraints. It also provides algorithms for calculating lower bounds and for dimensioning links (based on shortest path routing) for a given topology and traffic specification.



A Snapshot of Decaf

Using Cappuccino

Cappuccino is a tool to design ATM networks. A user will typically set a map as the background and place switches in different locations. He will then use the available algorithms to generate the optimum topology.

Cappuccino can also be used as a tool to do theoretical studies of the network design process. The user can generate random networks. The extensibility aspects of Cappuccino are very useful in this regard because the user can write algorithms to generate different kind of topologies like hexagonal, rectangular and triangular partitioning.. Random networks with arbitrary probability distributions can also be created and studied. The behavior of different kind of networks can be evaluated on different kind of constraints by defining new algorithms for constraint generation.

1. Network Design Using Cappuccino

Network design using Cappuccino is a four step process:

1.1 Designing Network Topology

The first step in the network design process is to decide where to place switches and links.

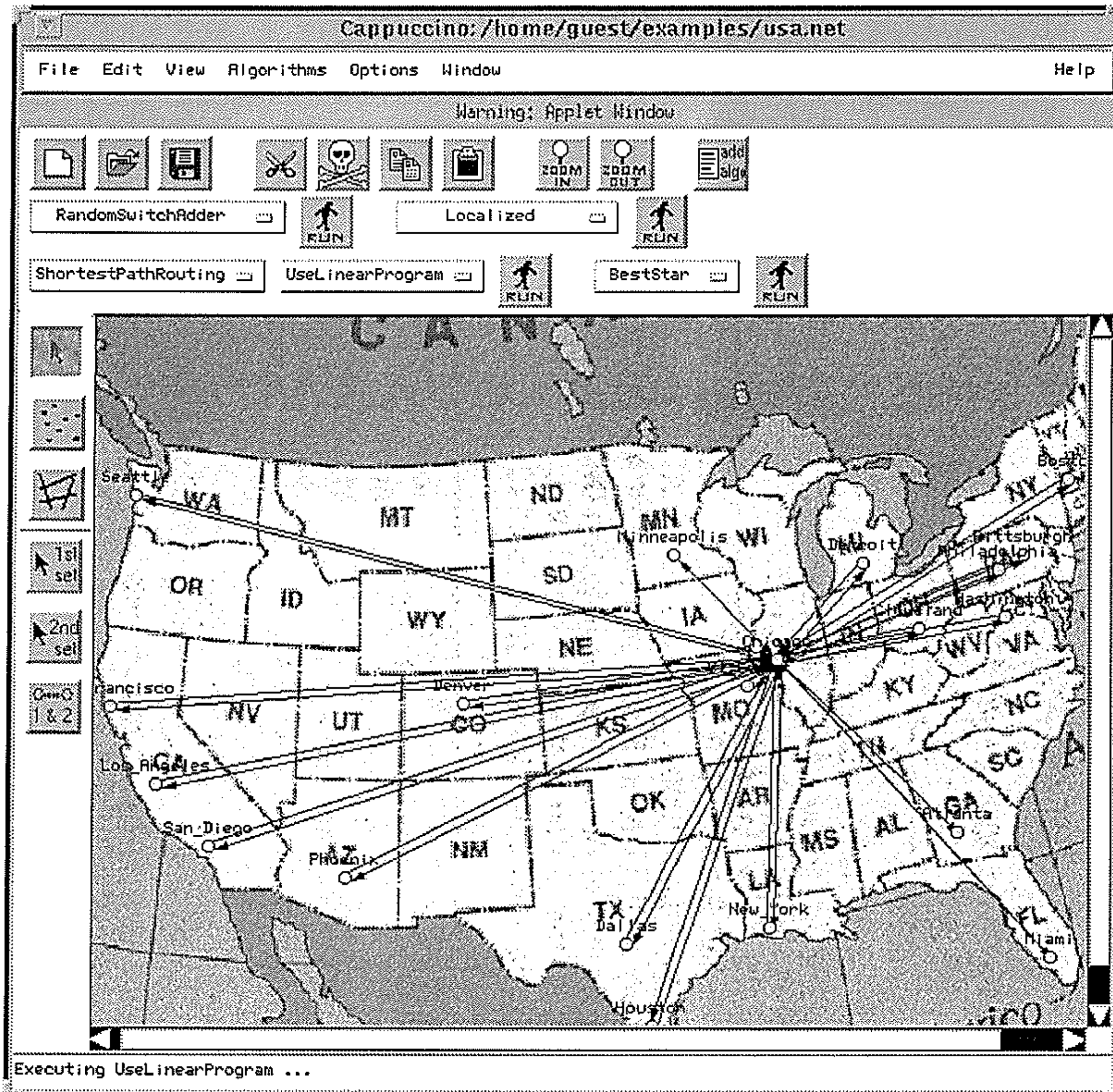
1.1.1 Creating Topology Visually

The graphical user interface of Cappuccino provides three modes of operation.

1. **Add Switch Mode:** In this mode, the user places switches at different positions by clicking on the canvas. The first click brings up a properties dialog box (unless the relevant properties are already present, for example, due to the use of an algorithm that required them) which asks for the default values of the different switch parameters. All subsequent switches are created using these default values. The values of the default parameters can be modified through the *Options|Defaults* dialog box.

2. **Add Link Mode:** In this mode, the user creates links by first selecting its *tail* and then its *head.* The first link brings up a properties dialog box (unless the relevant properties are already present) which asks for the default values of the different link parameters. All subsequent links are created with these default values. The value of these parameters can be modified through the *Options|Default* dialog box.

3. **Select Mode:** In this mode, the user can select different switches and links by clicking on them. Multiple objects can be selected by simultaneously pressing and holding the *ctrl* key. All selected items can be released by clicking (while not holding the *ctrl* key) at an empty location. The user can move a switch by double clicking on a switch and dragging it to another location.

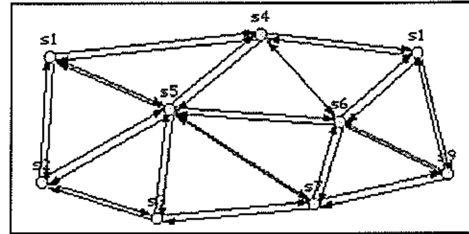


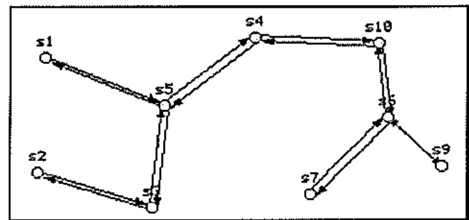
A Screen Snapshot of Cappuccino

In all modes, the user can right click on a switch and link to see its properties. The user can also perform cut, copy, delete and paste of switches and links by either using the command bar buttons or the menu-options. Keyboard accelerators for these operations are also available. The *Edit* menu provides various options to operate on the topology. The user can select or unselect all switches or links through menu options. The menu option, *Edit|Background,* can be used to put an image in the background of the network. This menu option brings up a file dialog box to select the image. The *View* menu provides options to turn off the display of the background image, name of the switches and links.

1.1.2 Automatic Creation Using Algorithms

The Cappuccino user can generate topology algorithmically. Presently, the following algorithms are available:





1. **Delaunay Triangulation:** This algorithm generates the Delaunay Triangulation topology for the network. It deletes the links that are not required in the triangulation.

2. **Minimum Spanning Tree:** This algorithm generates the minimum spanning tree based on the Euclidean distance between the switches. It also deletes the links that should not be present in the minimum spanning tree.

3. **Star Network:** This algorithm generates a star network rooted at the selected node. It also deletes all the links not belonging to the star network.

4. **RandomSwitchAdder/ RandomLinkAdder:** These algorithms are used to generate a random network.

5. **CompleteNetwork/ LinkComplement:** These are utility algorithms to help create a better topology.

1.1.3 Improving Topology by AdjustInitialNetwork Algorithm

The *AdjustInitialNetwork* algorithm provides the facilities to iteratively improve the topologies. When this algorithm is applied for the first time, it sets the given network to be the primary network. Subsequent invocations use the links present in the network (at that time) to improve the primary network. All those links are included in the primary network if they reduce the shortest path distances in the primary network by a ratio that is specified by the user.

1.2 Specifying Traffic Constraints

In this process, the network planner places constraints on the traffic flow.

1.2.1 Creating Constraints Visually

Cappuccino allows the users to manually create new constraints. In *select* mode, the user can select two sets of switches and create a set-pair constraint between them. The sets are defined by first selecting the nodes and then clicking the appropriate button. If no switch is selected at that time, then the set is taken to be the complement of the other set. The creation of the first constraint brings up a dialog box asking for default traffic flow between the two sets. All subsequent constraints use these default values. These default values can be modified through *Options|Defaults* dialog box.

1.2.2 Using ConstraintsWindow

The user can enter constraints in text-form through the *ConstraintsWindow.* In this window, the available constraints are presented in text form and the user can add new constraints by following the syntax. Cappuccino can parse the text to create actual constraints.

1.2.3 Using Algorithms to define Constraints

Some type of constraints are more suitable to be defined algorithmically. For example, distance based constraints are hard to specify through GUI but simple algorithms can be used to generate these constraints for each switch. Cappuccino provides the following algorithms for constraint generation:

1. **Localized:** This algorithm generates the constraints that specify the local traffic. The user can specify the percentage of local traffic and the distance within which this traffic is constrianed. The user can also specify the maximum and minimum number of switches to include in the local traffic.

2. **PairwisePerncentage:** This algorithm generates the set-pair constraints between each switch pair depending on the proportion of their source/ sink capacities to the total source/ sink capacity for all switches in the network.

1.3 Choosing Routing Algorithms

This process takes in a given network and its traffic requirements and constraints and generates the routing information for each of the links. Presently, the following algorithms are present in Cappuccino:

1. **Shortest Path Routing:** This algorithm generates the routing information based on the assumption that the traffic between two switches is routed along the shortest path between them.

2. **Distributed Routing:** This algorithm generates the routing information based on the assumption that the traffic between two switches is routed along alternate paths, each of which carry a percentage of traffic.

1.4 Dimensioning Links

The final step in the network design process is to decide the capacities required to be put in the links to support the traffic requirements and meet the constraints specified for the network. In Cappuccino, the user can select specific links (or all links) to dimension. The available algorithm uses a linear program to generate dimensioning information for each link. This dimensioning information is used to calculate the total cost of the network.

2. Using Super Algorithms

A super algorithm combines the above-mentioned four step process into a single step. It is either for convenience or to represent the algorithms that do not fit into the four step process (e.g., the legacy code). It can also be used to combine the four steps and iterate over a large number of topologies to find the best solution. Presently, the following algorithms fall in this category:

1. **Best Star:** This algorithm finds the best star network for the switches while taking into consideration the defined set-pair constraints. This algorithm ignores the set-pair constraints present in the network.

3. Using Algorithms

In Cappuccino, the user invokes different algorithms in the network design process. Most algorithms require some properties to run. On first invocation, such algorithms bring up a dialog box asking for the values while displaying the default values. On subsequent invocations, the algorithms use the chosen values. These properties can be modified later through the *Options|Defaults* menu option.

In Cappuccino, the algorithms are loaded and verified on demand. This may result in an occasional failure in the execution of an algorithm. The user is notified of such failures and the algorithm is removed from the choice list and the menu-bar.

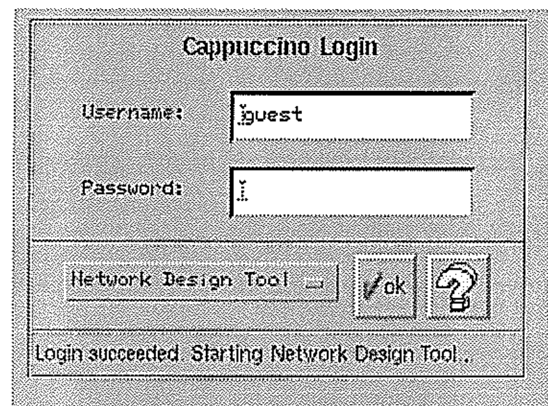
4. GUI Features

4.1 HTML based Help

Cappuccino provides its help by displaying the appropriate URL in the web browser. If the browser is not available then the URL of the help is displayed in a message box.

4.2 Log-in Screen

When the user visits the web-page of Cappuccino, he is prompted with a log-in panel. The user is required to enter a pre-existing user-name (with its password). A *guest* account is available that requires no password. The successful log-in results in the creation of a Cappuccino window. The user can create multiple independent windows by logging in many times. If the user is interested in creating different views of the same network, then *View|New View* menu-option must be used.



4.3 File Operations

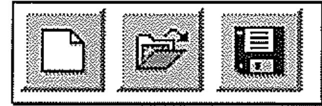
Cappuccino provides facilities for loading and saving different kinds of information in files. The following menu-options are available:

New	Ctrl+N
Open	Ctrl+O
Save	Ctrl+S
Save As	
Revert to Saved	
Print	Ctrl+P
Quit	Ctrl+Q

1. *File|New* discards the loaded network and creates an empty network after querying the user for any unsaved changes.

2. *File|Open* first queries the user to discard any unsaved changes and then presents the user with a file dialog box to select a network file.

3. *File|Save* saves the current network. The user is prompted with a file dialog box, if the present network has no associated file.



4. *File|Save As* prompts the user with a file dialog box to select the file in which the present network is to be saved.

5. *File|Revert* reloads the network from the file after prompting the user for discarding the unsaved changes.

6. *File|Print* prints the current network. A dialog box is presented in which the user can select the printer and its associated options.

7. *File|Quit:* This menu-option quits the current window (and all of its child window) after prompting the user to save his work.

4.4 The *Edit* Menu

This menu is used to edit the network graphically. The following menu-options are present:

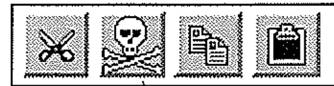
1. *Edit|Mode:* This menu-option is actually a sub-menu through which the user can select the GUI mode (one of *select, add switch,* and *add link*). The mode selection buttons are kept in synchronization with the selected mode.

Mode	
Cut	Ctrl+X
Copy	Ctrl+C
Delete	Ctrl+Delete
Paste	Ctrl+Y
Select	
Unselect	
Clear Links	
Clear Constraints	
Background	

2. *Edit|Cut:* This menu-option cuts the selected items to the clipboard. This is equivalent to a *copy* followed by a *delete*. This option is also accessible through the hot-key, *ctrl-x*.

3. *Edit|Copy:* This menu-option copies the selected items to the clipboard. This option is also accessible through the hot-key, *ctrl-c*.

4. *Edit|Delete:* This menu-option deletes the selected items from the network. This option is also accessible through the hot-key, *ctrl-d*.



5. *Edit|Paste:* This menu-option pastes the clipboard contents in the network. If the clipboard does not contain a valid representation of a network then an error dialog box is shown. The paste operation is done intelligently to ensure that the network does not get switches that are already present. While pasting a link the endpoints are inserted in the network if they are not already present. This menu-option is also accessible through the hot-key, *ctrl-y*.

6. *Edit|Select:* This menu-option provides convenient short-cuts for selecting the whole network, or all of its switches or links.

7. *Edit|Unselect:* This menu-option provides convenient short-cuts for unselecting the whole network, or all of its switches or links.

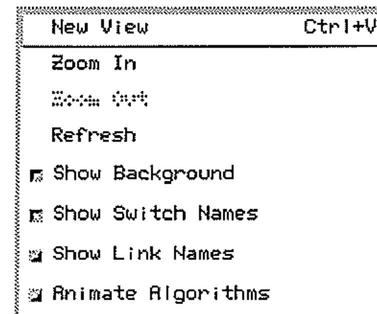
8. *Edit|Clear Links:* This menu-option deletes all the links present in the network.

9. *Edit|Clear Constraints:* This menu-options deletes all the constraints associated with the current network.

10. *Edit|Background:* This menu-option shows a file dialog box to select the background image for this network.

4.5 The *View* Menu

This menu contains menu-options to change the way a network is displayed.



1. *View|New View:* This menu-option is used to create multiple views of the same network. The user can view different portions of the same network in different windows. This menu-option is also accessible through the hot-key, *ctrl-v*.

2. *View|Zoom In:* This menu-option is used to zoom into the current network. The zooming is additive, i.e., the total viewable area is increased by a fixed amount. This menu-option is also accessible through the command bar buttons.

3. *View|Zoom Out:* This menu-option is used to zoom out the current network. The total viewable area is decreased by the same amount as it was increased in a zoom-in operation. This menu-option is also accessible through the command bar buttons.

4. *View|Refresh:* This menu-option is used to refresh the display.

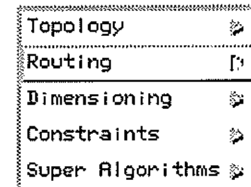
5. *View|Show Background:* This check box menu-option is used to toggle the display of the background image. GUI operations in Cappuccino (esp. involving dragging of switches) are slow when the background image is being displayed. Turning off show background improves the response time significantly. This menu-option is on by default and its status is saved with the configuration information.

6. *View|Animate Algorithms:* This menu-option is used to toggle the animation of algorithms. Animation of an algorithm can give useful insights into its nature but it slows down the algorithm considerably. This menu-option is off by default and its status is saved with the configuration information.

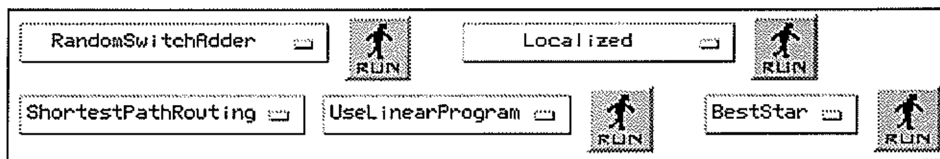
7. *View|Show Switch Names:* In a dense network, the names of switches might hide details that the user wants to see. This menu-option allows the user to toggle the display of switch names. This menu-option is on by default and its status is saved with the configuration information.

8. *View|Show Link Names:* Most users are not interested in knowing the names of the links. Moreover, in a dense network, the names of links might hide important details that the user wants to see. This menu-option allows the user to toggle the display of link names. This menu-option is off by default and its status is saved with the configuration information.

4.6 Menu-options for Running Algorithms



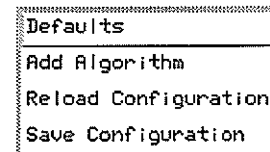
These menu options provide alternate mechanisms to execute the various algorithms present in Cappuccino. The *Routing m*enu is used to select the routing algorithm. The selection of a menu-option results in the execution of the corresponding algorithm. The choice lists of algorithms is kept in synchronization with the corresponding menu. For example, the selection of a routing algorithm through the menu-option results in the same algorithm being selected in the routing algorithm choice list.



Command Bar to Execute Algorithms

4.7 The Options Menu

This menu provides facilities to change the default values and configuration information of Cappuccino.



1. *Options|Default:* This menu-option is used to display all the default properties of the network, switches, links and the view. The default values can be changed by the user.

2. *Options|Add Algorithm:* This menu-option displays a file dialog box to the user and the user can select the class to add. The specified class is examined to be included in all categories of algorithms.

3. *Options|Save Configuration:* This menu-option prompts the user with a file dialog box to select a file in which all the configuration information is saved. The configuration information includes the default value of different algorithms, the status of various check-box menu-items (e.g., animate algorithm, show background, show switch names, etc.).

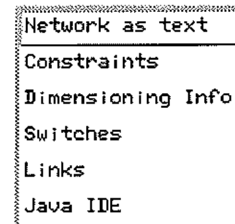
4. *Options|Reload Configuration:* This menu-option prompts the user with a file dialog box to select a file from which all the configuration information is reloaded. The user is notified if the file is not of the right type.

4.8 The *Window* Menu

1. *Window|Network as Text:* This menu-option is used to display the textual representation of the current network. This window keeps in synchronization with the GUI representation, i.e., a change in the network through the GUI (or by the algorithms) is automatically reflected in this window. The user can edit the network through this window. The format of the network is as follows:

```
// These are the dimensions (width, height) of the network.
1000.0, 1000.0
//Switches in this network:
{
// each switch is represented in the following way
// <name, x, y, alpha, omega, properties>
// the properties are represented as {name1=value1, name2=value2, ...}
   <s5,137.98,125.27,1.0,1.0,{}>
   <s6,229.45,448.35,1.0,1.0,{}>
}
//Links in this network:
{
// Each switch is represented as:
// <link name, tail switch name, head switch name, properties>
   <l11,s2,s1,{}>
   <l12,s1,s2,{}>
}
//Properties in this network:
{}
```

2. *Window|Switches* creates a window with a scrolling list of all the switches present in the network. The window provides menu options (which can also be activated by double clicking in the dialog box) to display the properties of the selected switches in a dialog box. The user can edit the properties in the dialog box.



3. *Window|Links* creates a window with a scrolling list of all the links present in the network. The window provides menu options (which can also be activated by double clicking in the dialog box) to display the properties of the selected links in a dialog box. The user can edit the properties in the dialog box.

4. *Window|Constraints* creates a window in which the constraints are presented as text. The user can edit the constraints. The text format of the constraints is as follows:

<(switches of the first set), (switches of the second set, empty means complement of the other set),

traffic from the second set to the first set, traffic from the first set to the second set>

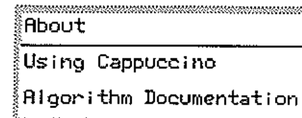
An example constraint is: <(s1, s2),(s3),0.5,0.2>

5. *Window|Java IDE:* This menu-option starts the built-in integrated development environment for creating java classes.

The text views are free format, i.e., the number and position of the white spaces is not important. The // is used to start a comment that extends till the end of the line. /* */ can be used for block comments. While separating items, it is not important if a "," is present or not, just any white space is enough.

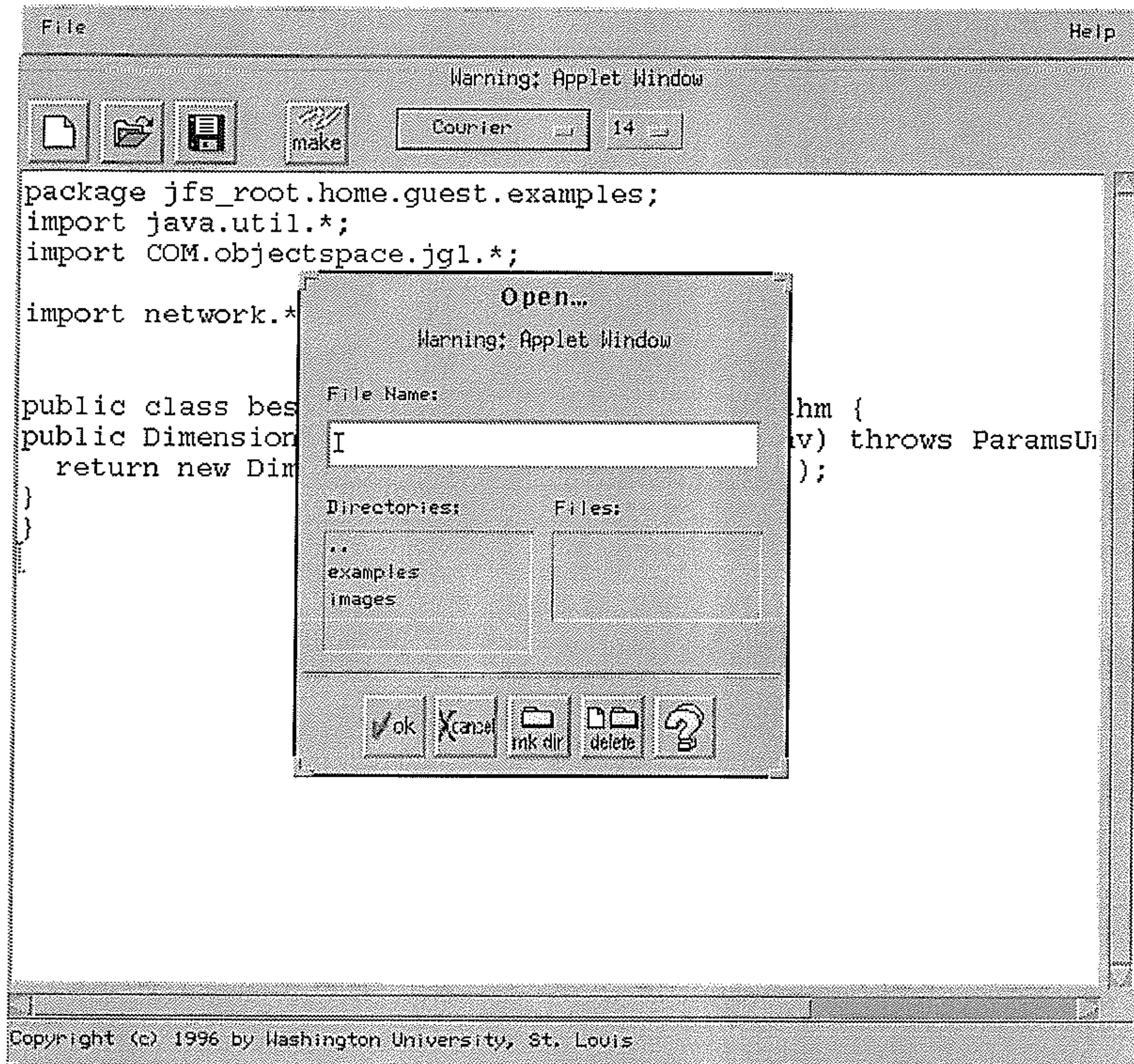
4.9 The *Help* Menu

This menu contains menu-options for general information related to Cappuccino project. *Help|About* takes the user to the Cappuccino home page. *Help|Using Cappuccino* takes the user to the online user manual. *Help|Algorithm Documentation* shows the webpage corresponding to the algorithms present in Cappuccino.



4.10 The Built-in Java IDE

Cappuccino comes with an in-built IDE for creating java classes. The editor is started from the *Options|Java IDE* menu. JavaIDE provides facilities to edit and compile java files.



The built-in Java IDE

The Java IDE provides following menu-options:

1. *File|New:* This menu-option discards the current contents after prompting the user for any unsaved changes. This menu-option is also accessible through the command-bar.

2. *File|Open:* This menu-option prompts the user to save any unsaved changes and then presents him with a file dialog box to load files. This menu-option is also accessible through the command-bar.

3. *File|Save, File|Save As:* These menu-options are used to save the current file. If the current contents are not saved to any named file or the *File|Save As* option is chosen, then the user is prompted with a file dialog box to select a file. *File|Save* is also accessible through the command-bar.

4. *File|Compile* : This menu-option prompts the user to save any unsaved changes and then compiles the file. The results of compilation are shown in a message box. This menu-option is also accessible through the command-bar (*make* button).

4.11 Synchronized GUI Objects

In Cappuccino, the different windows displaying the same network or constraints are kept in synchronization. A change made in one component is propagated to the other windows that depend on the same information.

5. Differences in the Applet and Application Version

Cappuccino is available as an applet as well as an application. The applet and application versions largely provide the same interface to the user. However, due to certain reasons (security restrictions, availability of the browser, etc.), the two versions are different in some key respects. The following differences exist in the two versions:

1. The java security restrictions do not allow untrusted applets to do any local file i/o. Hence, the applet version of the Cappuccino creates a virtual file system for the users on the server. Different accounts are created which have associated user name, password and home directories. The web-page presents a login window, through which a user can login. The login process creates a Cappuccino window that uses the associated home directory for all the file i/o.

2. In the applet version, the user can create multiple independent Cappuccino windows by logging in multiple times. In the application version, the user needs to re-run Cappuccino to get multiple windows. However, multiple views of the same network can be created in both versions.

3. In the applet version, the cut and paste operations through the menu-options do not access the system clipboard. However, the windows that display the text (e.g., Network as text, constraints window, etc.) can access the system clipboard. The application version has full access to the system clipboard.

4. In the applet version, the help is displayed in a separate frame in the same browser in which the applet is running. In the application version, the help is displayed by connecting to an already running browser (Netscape). If a browser is not already running then a new browser is started. If the browser does not exist then the help URL is displayed in a message box.

5. In the applet version, the compilation of the algorithm is done on the server. Moreover, the algorithm writer needs to define his class in a package that starts with *jfs_root.home.user_name.directory*. In the application version, the compilation is done locally and the user is free to define his algorithm in any package.

Extending Cappuccino

This chapter describes the packages and classes that an algorithm writer needs to understand. Hyper-linked reference documentation for these packages (generated using *javadoc)* is available at http://www.cs.wustl.edu/~inder/cappu/docs/api/packages.html.

1. Software Architecture

1.1 Guiding Rules and Design Principles

The following design principles were followed in Cappuccino:

1. Main emphasis in Cappuccino is on a loosely coupled design. Different functionalities in Cappuccino are implemented as separate packages. The public interface of a package exposes the abstraction of the services provided by that package while hiding the internal implementation details.

2. Software in which redundancy in code is present is very hard to maintain. In Cappuccino, the stress is on reducing the number of lines of code. This has an added advantage of a smaller footprint that has its own merits for java because it is a network oriented language. Reusability of the design and components [GHJV95] is a basic design goal in Cappuccino. Cappuccino provides general solutions for a help system, transparent file handling, etc.

3. Any extensible design is necessarily flexible. Cappuccino exploits the runtime linking capabilities of java to provide the flexibility to plug-in arbitrary IDEs, adding new algorithms, and choosing alternate implementation classes for representing networks.

4. Cappuccino tries to achieve efficiency without sacrificing a good design. It uses efficient data structures provided by the JGL library for representing networks. Its design allows the algorithm writers to directly use efficient JGL algorithms for sorting, and searching directly on the network. The class *Network* exposes sufficient operations (like indexed access to switch and link arrays) to obviate the need for unnecessary data copying.

1.2 Applet as well as Application

Cappuccino is available both as a java applet and a java application. Applet and application models differ in four basic ways, viz., file handling, showing help, compilation and clipboard management. For maximum code reuse, these operations are abstracted out as java interfaces. The rest of the Cappuccino code uses these abstract interfaces for these activities. The concrete classes for these interfaces differ for the applet and the application versions. The applet class (*CappuApplet*) and the application class (*CappuApplication*) instantiate relevant classes and plug them in.

1.3 Transparent File Handling

Java applets face greater security restrictions than java applications. In the present sandbox model, all applets are forbidden to access the local file system. In Cappuccino, file i/o is done on the local disk and for the applet version, file i/o is done on the server. File i/o is done through the general mechanisms provided through the *FileHandler* class that is implemented as an abstract singleton [GHJV95]. The concrete file i/o mechanisms are implemented by *AppletFileHandler* and *ApplicationFileHandler* classes.

1.4 Incorporating Help

Cappuccino provides its help support as HTML pages. Different components provide their help information in different URLs. URL tagging is used to keep information related to multiple events in a single HTML file. The best way to view an HTML based help is through a web browser. The applet version of Cappuccino has an advantage in this regard because it is already running inside a browser and hence can display help information in a separate frame in the browser. The standalone version of Cappuccino starts a browser, of the user's choice, the first time help is requested.

The *HelpEngine* is the abstraction for handling help in Cappuccino. The *AppletHelpEngine* and *ApplicationHelpEngine* provide concrete implementations for applet and application scenarios respectively. The *HelpEngine* is a singleton and all help requests must be channeled through it. Each entity that needs to provide help for itself obtains the URL for its help information through a configuration file. This approach allows the flexibility of changing the URL without affecting Cappuccino code. Although the singleton abstraction of *HelpEngine* looks similar to the *FileHandler* abstraction, yet they are technically different. The fundamental difference between *HelpEngine* and *FileHandler* is that the help activity can be asynchronous (i.e., the application can proceed without waiting for it to complete) whereas the file operations are synchronous because typically the application needs to know the result of the operation.

1.5 Showing Help Hints

Contemporary GUI systems provide useful hints and tips to the user depending on his available choice of actions. Often, there are different ways to show a hint, for example in a status bar or as a tool tip next to a GUI button. The architecture of Cappuccino provides the necessary flexibility to show hints in different ways by using an event driven approach. Different components generate their hint tips and provide interfaces to register listeners (*HelpHintListener*) for this purpose. Presently, in Cappuccino all hint tips are shown in the status bar.

1.6 Handling Error Conditions

Any big software system must expect and handle occurrence of error conditions. Cappuccino encounters three types of errors:

1. Errors due to software bugs: Different classes in Cappuccino take a pessimistic approach while accepting parameters in their public methods. Invalid parameters are handled either by using a default parameter (if it is possible to do so) or by aborting the method or the program.

2. Errors due to improper configuration and setup. Cappuccino provides a lot of flexibility in choosing representation classes for different entities. This raises the possibility of an invalid specification for a representation class. This is handled by either choosing the default representation (if one exists) or by terminating the program.

3. Errors due to invalid user actions and inputs. Cappuccino tries to minimize such errors by making invalid actions impossible (e.g., by graying out a menu option that is not valid).

In handling any type of error, it is desirable to give some form of feedback to the user. However, the feedback contents are often different for different types of users. Cappuccino defines three different levels of feedback:

1. Feedback for Network Planners: The network planners receive information about invalid actions and inputs. The also are notified of presence of the invalid algorithms.

2. Feedback for Administrators: Administrators are notified of configuration errors resulting from the choice of invalid representation classes.

3. Feedback for Programmers: Programmers are notified of software bugs.

The *Debug* class decides the error handling policy. It defines different methods for handling different types of error conditions. Warnings are generated for non-fatal errors.

1.7 Transient and Persistent Properties

Cappuccino is designed to incorporate new algorithms without any modifications to the internal network representation and other code. A static representation for the network, in which the data structures for network, links and switches are predefined, is bound to fail because different algorithms often need different kind of information for the same network. For example, an algorithm for calculating the *minimum spanning tree* needs to know the *weight* of each link, whereas, an algorithm to generate a random network needs to know *the number of nodes and links* to create.

One approach to handle this problem could be to guess all possible algorithms that might ever need to be incorporated in Cappuccino and define fields corresponding to all of them. This approach has a few serious drawbacks:

1. Anticipating all such parameters is not easy.

2. All such fields will be visible to all the algorithms. This is not a good software practice (OO concepts call for maximum data hiding). For example, in the example given above, a person who is writing minimum spanning tree algorithm, will be confused to see the *number of nodes and links to create.*

3. Many such fields might need suitable initializations even by those algorithms that do not deal with them. This might be inconvenient and error prone for the algorithm writer, apart from having performance implications.

Cappuccino uses a novel approach to handle this problem. It defines the notion of *properties,* which are *name-value* pairs. In Cappuccino, the representation of a switch, link or a network allows addition of new properties (by different algorithms) at run-time. Different algorithms store their parameters as properties. There can be two kind of properties, *transient* and *persistent.* As the name suggests, the transient properties do not last from session to session and are meant for the exclusive use of a single algorithm. The persistent properties are associated permanently with an object and can be shared by different algorithms. These are used by the different algorithms to refer to the same parameters, for example, link weight is required by all algorithms that generate a minimum spanning tree.

1.8 Algorithm Representation

Algorithms in Cappuccino are implemented as function objects. A function object implements a specific java interface that defines a method *execute()* in which the algorithm does its work.

Each of these set of algorithms are kept in *hashtables* that are indexed by the function names. An algorithm is loaded (and verified to be of the right type) the first time it is invoked in the tool (unless it is cached).

1.9 Document/View/Controller Approach

Cappuccino uses the Document/View/Controller approach [MNB97] for GUI Design. This approach is useful for reducing coupling between different software components for large GUI projects. In this approach, a representation of an object is separated from its display and the display is further separated from the mechanisms to modify it. For example, in Cappuccino, the physical representation (called *the document*) of a network (implemented by the class *network.Network*) provides abstract services that a network should provide. A network can have different graphical representations called *views,* e.g., visual representation, network as text, etc. Each view is responsible for displaying the network in its own particular way. The modifications to a view (e.g., by user interactions) are handled by its *controller* that propagates the changes not to the view but directly to the document. Each view *listens* to the changes in the document. Cappuccino uses the *push* model of change propagation meaning that each change in the document is pushed to the views and the views need not go back to the document to know what exactly changed.

2. Package Structure

Cappuccino consists of seventeen packages and some bootstrapping classes. Bootstrapping code configures Cappuccino to work either in applet or application mode. The *gui* package presents the graphical user interface to the user and drives the system. The network package contains definitions for switch, link, network and constraints. It also contains interface definitions for different type of algorithms. Algorithms of different types are present in their own packages. The *config* package provides some configuration information that is used by bootstrapping code, gui and network packages. The *general* package contains some general purpose utility classes. The *stk* package provides the GUI toolkit used to develop the graphical user interface of Cappuccino.

3. Understanding the *network* Package

This is the most important package an algorithm writer needs to understand to write algorithms effectively. It consists of classes that define the network, constraints, algorithms and their parameters.

3.1 Network Representation

This section describes the network representation as it is seen by an algorithm writer. The network consists of switches, links and network. The class *Switch, Link,* and *Network* are *observable* objects and they push their changes to the observers. Extensibility in the data structure is provided by providing *transient* and *persistent* properties (by implementing *general.PropertiesHolder* interface). These classes are *cloneable* and also provide constructors to construct object from their textual representation. They also provide *setter* and *getter* methods to obtain and change values of their member data objects. These classes implement *java.io.Serializable* interface but do not save transient properties to the disk.

3.1.1 The Class Switch

The class *Switch* represents an ATM switch. A switch has a name and a location (x, y). A switch also defines a source capacity, *alpha,* and a sink capacity, *omega.* The name of each switch must be unique.

3.1.2 *The Class* Link

The class *Link* represents a directed link connecting two ATM switches, *head* and *tail.* It also has a name. It provides utility methods to reverse its direction, and to find the other end point given one endpoint. In Cappuccino, self loops and multi-edges are not allowed.

3.1.3 *The Abstract Class* Network

This class represents a network of ATM switches and links. The class *Network* defines following methods:

1. *getNumSwitches()/ getNumLinks():* Used to obtain the number of switches and links in the network.

2. *add(Switch)/ add(Link):* Used to add switches and links to the network. The network does not allow self-loops and multiple links between two switches.

3. *remove(Switch)/ remove(Link):* Used to remove the switches and links from the network.

4. *getSwitches()/ getLinks():* These methods return an *Enumeration* of all the switches (or links) present in the network. The enumeration becomes invalid on any subsequent addition or deletion in the network.

5. *getOutgoingLinks()/ getIncomingLinks():* These methods return an *Enumeration* of outgoing/ incoming links of a switch. The enumeration becomes invalid on any subsequent addition or deletion in the network.

6. *getSwitchAt()/ getLinkAt():* These methods provide indexed access to the switch / link containers. The index becomes invalid on any subsequent addition or deletion to the network.

7. *indexOf(Switch)/ indexOf(Link):* These methods provide index of a switch/ link in its container. The value becomes invalid on any subsequent addition or deletion in the network.

8. *getLinkGoingFrom(Switch s1, Switch s2):* This method returns the link whose tail is s1 and the head is s2. If no such link is present then it returns *null.*

9. *contains(Switch)/ contains(Link):* This methods are used to find out if a switch or a link belongs to the network.

10. *createNetwork():* The class *Network* is an abstract class. Hence, it cannot be instantiated directly. This class method is used to create a new empty network or a network from its textual representation.

11. *clear()/ clearLinks():* This method deletes all the switches/ links of the network.

12. *setHeight()/ setWidth():* These methods determine the overall dimensions of the network. The units of the height and width is left for the user to interpret.

3.2 Constraint Representation

Cappuccino uses set-pair constraints to represent traffic flow between set of switches in the network. These constraints are in addition to the implicit constraints that are defined by the *source* and *sink* capacities of the switches. The set-pair constraints are the most general kind of constraints and can be used to represent any other kind of constraints, e.g., distance based constraints, capacity based constraints etc.

In Cappuccino, the constraints are separated from the network to which they can be applied, i.e., the network data structure does not contain any reference to the constraints. Instead, each constraint keeps reference to the network it is being applied to. The constraints observe the associated network for any changes and automatically update themselves to handle the addition and deletion of switches.

3.2.1 The Class SetPairConstraint

This class represents a single set-pair constraint. A constraint is associated with a network and has two sets of switches, *FIRST* and *SECOND.* These sets can share switches. A set can also be defined to be the *complement* of the other set. Each set-pair constraint defines two traffic capacities, *out* and *in,* representing the traffic flowing from set 1 to set 2 and vice-versa. A constraint observes its associated network and modifies itself appropriately on deletion of switches in either of its sets. This class provides following methods:

1. *add(setId, Switch)/ remove(setId, Switch):* These methods are used to add and remove switches to the sets. The *setId* must be either *FIRST* or *SECOND.*

2. *clear(setId):* This method removes all the elements of the specified set.

3. *getIn()/ getOut:* These methods are used to obtain the traffic going from one set to the other.

4. *isTrivialConstraint():* This method is used to findout if this constraint is a trivial constraint, i.e., both sets contain all the switches of the network.

5. *elements(setId):* This method returns an enumeration of all the switches present in the specified set.

6. *size(setId):* This method returns the number of elements in the specified set.

3.2.2 The Class Constraints

This class represents a collection of set-pair constraints for a network. It is an *observable* object and initiates a *pull* on any change. It also monitors the associated network and clears itself if it is replaced. Its methods are:

1. *add(SetPairConstraint)/ remove(SetPairConstraint):* These methods are used to add and remove constraints.

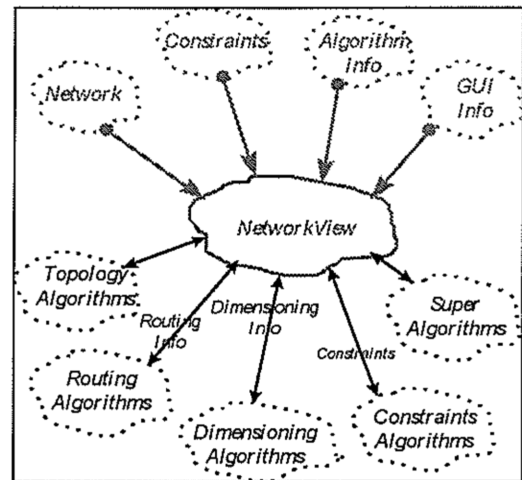
2. *clear():* Removes all constraints.

3. *elements():* Returns an enumeration of all the constraints.

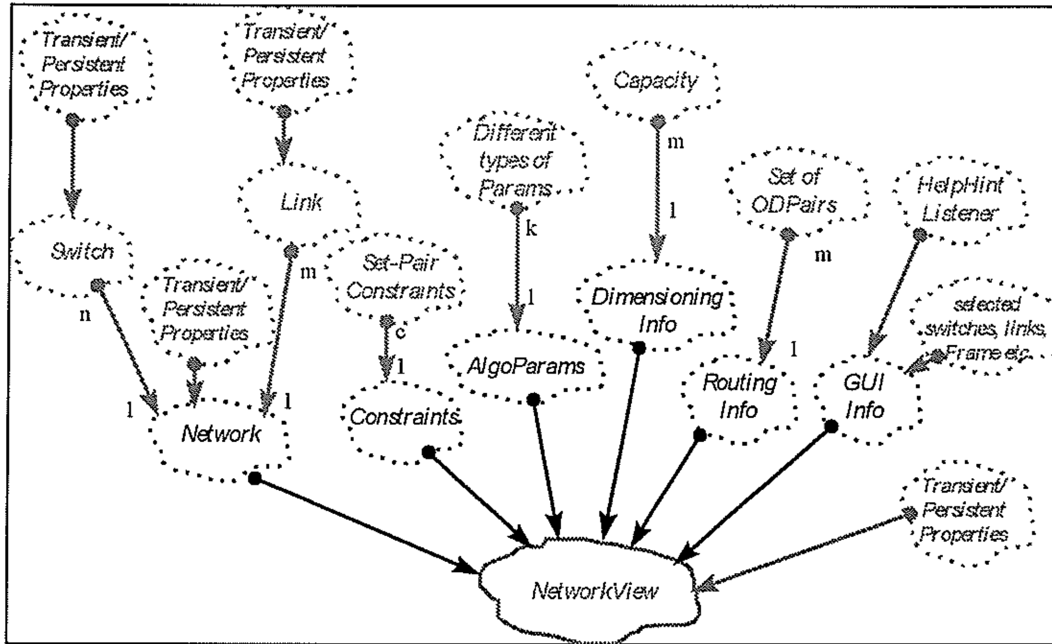
3.2.3 The Interface NetworkProperties

This interface defines some common properties of a network. It also defines keys to access different types of algorithms.

3.3 Interface *NetworkView*



This interface represents an abstraction to represent all kinds of information algorithms need. The algorithms interact only with a network view and need to obtain only the information they want. A network view consists the network, constraints, and the different parameters required by different algorithms. It also contains the routing and dimensioning information related to the network. It provides extensibility in its data structure to store general parameters required by algorithms by implementing the *general.PropertiesHolder* interface. A network view provides mechanism to access the GUI information of the tool. For example, it provides methods to find the switches and links selected by the user, and access to the help hint displaying mechanisms.



3.4 Classes for Algorithm Parameters

3.4.1 The Class Param

This class represents a parameter desired by an algorithm. It contains the key to access the parameter and the default value for the parameter.

3.4.2 The Class AlgoParams

This class defines the set of parameters that an algorithm can require. The parameters are classified in four types,

1. View Parameters: The parameters associated with the network view. For example, the number of switches to add for *RandomSwitchAdder* algorithm.

2. Network Parameters: The parameters associated with the network.

3. Link Parameters: The parameters associated with each link of the network. For example, the *weight* of a link.

4. Switch Parameters: The Parameters associated with each switch of the network.

This class defines following methods:

1. *add(ParamType, Param):* This method is used to add a parameter of the specified to the this object.

2. *contains(AlgoParam):* This method returns true if all the parameters present in the specified object are present in this object as well. This method is used by the algorithm writers to check if the parameters required by them are already present or not.

3.4.3 The Class ParamsUnavailableException

This exception is thrown by an algorithm when it does not find the parameters it desires. This exception contains an *AlgoParam* object that describes all the parameters that are desired by the algorithm but are not present.

3.4.4 The Class ViewProperties

This class defines some commonly used properties associated with the view. It also defines constant parameters objects for some of these properties.

3.5 The Class *NetworkUtils*

This class contains some convenience methods for the classes of this package. Following methods are present:

1. *distance(Switch s1, Switch s2)/ distance(Link):* Returns the Euclidean distance between the specified switches / the length of the link.

2. *isComplete(Network):* returns true if the specified network is a complete network i.e. it contains all possible links.

3. *merge(Network net1, Network net2):* merges two networks.

4. *sumAlpha(Network), sumOmega(Network):* returns the sum of alpha (or omega) capacities of the all of the switches of the specified network.

5. *getTotalCost(DimensioningInfo):* returns the sum of the cost of each link. The cost of a link is defined to be the product of its euclidean length and its capacity.

4. Writing Topology Algorithm

A Topology algorithm modifies the topology of the network i.e. it adds or deletes switches and links in the network. In Cappuccino, a topology algorithm implements the *TopologyModifier* interface. The algorithm does its work in the *execute()* method. The *execute()* method is defined as:

```
public interface TopologyModifier {
   public abstract NetworkView execute(NetworkView nv)
                              throws ParamsUnavailableException;
}
```

The algorithm is applied to the parameter *nv*. The algorithm should throw *ParamsUnavailableException* if it doesn't not find all the required parameters An example algorithm is given here:

```
package jfs_root.home.guest;  // "guest" user's work directory
                              // relative to the cappuccino
                              // classes.
import java.util.*;           // Used for Hashtable etc.
import COM.objectspace.jgl.*; // Might want to use JGL datastructs
                              // and algorithms.

import general.*;             // Cappuccino specific utilities.
import network.*;             // Defines network, constrains and
                              // algorithm classes

/** A functor to add switches at random positions in the network.
  * @author Inderjeet Singh */
public class RandomSwitchAdder implements TopologyModifier {
// This algorithm wants to define a new parameter, NumSwitchesToAdd
public static final String ParamKey = "NumSwitchesToAdd";

// This method creates the required set of parameters by this
// algorithm
public AlgoParams getRequiredParams(Network net) {
            AlgoParams ap = new AlgoParams();
      ap.add(AlgoParams.VIEW_PARAM,
             new Param(ParamKey, new Integer(10)));
      ap.add(AlgoParams.VIEW_PARAM,
             new Param(ViewProperties.intDefaultSwitchNameIndex,
                       new Integer(net.getNumSwitches())));
      ap.add(AlgoParams.VIEW_PARAM, ViewProperties.pDefaultAlpha);
      ap.add(AlgoParams.VIEW_PARAM, ViewProperties.pDefaultOmega);
      return ap;
}

// This is the main method in which RandomSwitchAdder will add
// new switches to the network.
public NetworkView execute(NetworkView nv)
                        throws ParamsUnavailableException {

      Network net = nv.getNetwork();      // extract network to work on.

      // check if the desired parameters are actually present
      // in the network view.
      AlgoParams requiredParams = getRequiredParams(net);
            if(!nv.getAlgoParams().contains(requiredParams))
            throw new ParamsUnavailableException(requiredParams);

      // Parameters were present, so get them.
      int count = ((Integer)nv.getPersistentProp().
                  get(ParamKey)).intValue();
            String switchName = (String)nv.getPersistentProp().
                  get(ViewProperties.strDefaultSwitchNameTag);
            int nameIndex = ((Integer)nv.getPersistentProp().
         get(ViewProperties.intDefaultSwitchNameIndex)).intValue();
```

```
                double maxAlpha = ((Double)nv.getPersistentProp().
                get(ViewProperties.dblDefaultAlpha)).doubleValue();
                double maxOmega = ((Double)nv.getPersistentProp().
                get(ViewProperties.dblDefaultOmega)).doubleValue();

            Random random = new Random();
            for(int i=0; i<count;) {
                Switch s = new Switch(switchName + nameIndex,
                                random.nextDouble()*net.getWidth(),
                                random.nextDouble()*net.getHeight(),
                                maxAlpha*random.nextDouble(),
                                maxOmega*random.nextDouble());
                if(net.add(s)) {  // switch was successfully added
                    ++nameIndex;
                        ++i;
                }
                }

            // update any parameter if it is required
            nv.getPersistentProp().put(ViewProperties.
                intDefaultSwitchNameIndex, new Integer(nameIndex));
            return nv;
        }    // end of execute method
}    // end of RandomSwitchAdder algorithm
```

The above example represents a typical manner in which an algorithm is written. The algorithm first checks if its required parameters are available and if not then it throws an exception. Note that the algorithm writer is not required to write any GUI code. However, if the need ever arises, the *NetworkView* interface exposes sufficient information to write sophisticated GUI input mechanisms.

5. Writing Routing Algorithms

Routing algorithms create routing information for a network. In Cappuccino, a routing algorithm needs to implement the *NetworkRouter* interface.

```
public interface NetworkRouter {
    public RoutingInfo execute(NetworkView nv)
            throws ParamsUnavailableException;
}
```

A routing algorithm returns an object of the type *RoutingInfo,* which keeps the routing information for each link. The routing information for a link is described by the set of *ODPairs* passing through it.

5.1.1 The Class ODPair

This class represents an origin-destination pair of switches and the percentage of traffic flowing between the two switches using the link to which this ODPair is associated.

5.1.2 The Class RoutingInfo

This class, for each link in the network, maintains a list of *ODPairs* that send traffic through the link. It observes the associated network for any changes and invalidates itself if the network changes.

6. Writing Dimensioning Algorithms

Dimensioning algorithms are used to dimension the links of the network, that is, to find out the required traffic carrying capacity of each link to support the desired traffic flow. In Cappuccino, the dimensioning algorithms implement *LinkDimensioner* interface.

```
public interface LinkDimensioner {
      public DimensioningInfo execute(NetworkView nv,
                                      RoutingInfo ri)
                        throws ParamsUnavailableException;
}
```

The parameter *ri*, is the routing information generated by a routing algorithm. The *execute()* method returns an object of type *DimensioningInfo*.

6.1.1 The Class DimensioningInfo

This class maintains information about the capacity associated with dimensioned links of the network. It observes the network for any changes and invalidates itself if the network changes.

7. Writing Constraints Algorithms

The constraints algorithms are used to generate the set pair constraints in the network. In Cappuccino, a constraints generating algorithm must implement the *ConstraintsGenerator* interface.

```
public interface ConstraintsGenerator {
      public NetworkView execute(NetworkView nv)
                  throws ParamsUnavailableException;
}
```

The *execute()* method of this algorithm is identical to that of the *TopologyModifier* interface. This highlights the fact that the differentiation in the type of an algorithm is largely by convention. A constraint generator algorithm is free to modify the topology if it so desires.

8. Writing Super Algorithms

Super algorithms are used to combine the different steps in the network design process. In Cappuccino, a super algorithm implements the *SuperAlgorithm* interface.

```
public interface SuperAlgorithm {
      public DimensioningInfo execute(NetworkView nv)
```

```
                                throws ParamsUnavailableException;
    }
```

The super algorithm directly returns the dimensioning results in an object of the type *DimensioningInfo*.

8.1 Understanding the *general* Package

This package provides important services and many utility classes in Cappuccino. The following classes are for the use of an algorithm writer:

8.1.1 The PropertiesHolder Interface

This interface is implemented by the objects that support extensibility in their data structure through properties. The method *getTransientProp()* is used to obtain transient properties of the object and the method *getPersistentProp()* is used to obtain the persistent properties of the object. Cappuccino shows the persistent properties of the switches, links, network and the view to the user and allows the user to change these at will. The persistent properties of an object are also streamed to the disk and used in *toString()* and *equals()* methods. The transient properties are not shown to the user and are not used in the *toString()* and *equals()* methods of the object. The keys to obtain transient properties should be unique and can be generated using the *general.Utils.generateUniqueKey()* method. The keys to obtain persistent properties must be of the type *java.lang.String* and need not be unique. It is even recommended that the different algorithms use the same key to obtain same persistent properties to reduce the number of properties shown to the user.

8.1.2 The Class Debug

This class is used to show debugging or error information in Cappuccino. It defines different methods for handling different types of errors. For an algorithm writer, the most important methods of this class are *warnUser()* and *errorUser()*. These two methods are available in multiple forms but the algorithm writer must use the form that takes a *Frame* argument.

1. *errorUser():* This method is invoked to report a fatal error that must result in the termination of the algorithm. An algorithm writer should use the

2. *warnUser()*: This method is invoked to report a non-fatal error that will be meaningful to the Cappuccino users..

8.1.3 The Abstract Class FileHandler

An algorithm writer sometimes needs to do some file operations. In Cappuccino, all file i/o happens through the *FileHandler* interface. The actual object for the FileHandler can be obtained by calling the *getInstance()* class method. Following methods of this class are useful for an algorithm writer:

1. *getInstance():* This *class* method returns the actual FileHandler object for the Cappuccino.

2. *getFileDialog():* This method is used to obtain a file dialog. It is important to remember that the complete path name for the selected file is *getDirectory()* + *getFile()*, and not *getFile()* alone.

3. *getInputStream(fileName):* This method is used to open a file for reading. If an error is encountered while opening the file an *InvalidFileOperationException* is thrown.

4. *getOutputStream(fileName);* This method is used to open a file for writing. If an error is encountered the *InvalidFileOperationException* is thrown.

5. *getImage(imageFileName):* This method is used to create an image corresponding to the specified image file name.

The following example shows how to use a file dialog to obtain a file name chosen by the user and then open it as an input stream.

```
FileHandler = FileHandler.getInstance();  // obtain filehandler object
// obtain information requird by the FileHandler
Hashtable prop = nv.getCappuProp();
try {
      // NetworkView nv is the parameter passed to the algorithm
      FileDialogInterface fdi = fh.getFileDialog(nv.getFrame(),
                                "Open Network",
                                FileDialog.LOAD, prop);
            fdi.show(); // show the dialog box to the user
      if(fdi.getFile() == null)     // null indicates that the
            return;                 // user canceled the operation.
      String fileName = fdi.getDirectory() + fdi.getFile();
      InputStream is = fh.getInputStream(fileName, prop);
      // file successfully opened
} catch(InvalidFileOperationException ifoe) {
      Debug.warnUser(fm, "Error Opening file: "+ifoe.getMessage());
}
```

8.1.4 The Class HelpEngine

This class *HelpEngine* is used to show the help information in Cappuccino. An algorithm writer can use *showHelp()* method to show help to the user.

Example code:

```
URL helpURL = new URL("http://siesta/~user/myalgo.html");
HelpEngine.getInstance().showHelp(new HelpEvent(this, helpURL));
```

8.1.5 *The Class CGI*

This class defines method to access different CGI scripts used in Cappuccino. It defines a generic *post* method that implements HTTP POST. It also defines a method to run linear programming CGI script and obtain its results.

8.1.6 The Class ObserveeTracker

This class provide a convenient mechanism to keep track of the objects being observed by an *Observer* object. It provides mechanisms to stop observing a single object or all objects currently being observed.

8.1.7 Miscellaneous Utility Classes

1. **FiniteStateMachine**: The class *FiniteStateMachine* is sub-classed to create a class that needs to run like a finite state machine. The important methods in this class are *start(), stop()* and *process(Event)*.

2. **Utils:** This class provides common utility functions. The function, *generateUniqueKey()* is used to obtain a unique key to access transient properties. This class also provides some functions to generate concrete classes using meta-classes to support extensibility features in Cappuccino. It also provides some methods that aid in parsing text and formatting floating point numbers.

3. **Assert:** This class defines functions to assert post-conditions and pre-conditions. Failure of an assert results in the *java.lang.IllegalArgumentException* being thrown.

Performance Evaluation and Project Statistics

1. Performance Evaluation

In Cappuccino, the performance evaluation has two aspects, the speed of the execution of the algorithms and the general performance of Cappuccino itself. The performance of an algorithm depends largely on its implementation. However, Cappuccino does have some features to improve performance.

1.1 Efficient Network Representation

The underlying network representation used in Cappuccino is built using JGL data structures that are very efficiently implemented. The *Network* class exposes sufficient methods to avoid creating subsidiary data structures for the network by the algorithms, a problem that plagued *Decaf*, the earlier version of the network design tool. Moreover, in Cappuccino, only an abstract form of the network is used, hence, the underlying representation can be changed by an alternate representation that works better for the critical and performance hungry algorithms. The default implementation of the *Network* class permits the changing of the actual container classes used to keep switches and links without recompiling itself.

1.2 Algorithm Animation

Animation of an algorithm animation can provide useful insights into its nature. However, it can slow down the execution time considerably. Cappuccino provides menu-options to turn on and off algorithm animation.

1.3 Hiding Background

In Cappuccino, the user can move a switch by dragging it. The dragging process generates a *repaint* event for each of the intermediate mouse positions. In the presence of a background image, the dragging process can slow down the response time of the GUI considerably. This slow down occurs due to the slow image rendering by the java run-time system. Cappuccino provides menu-options to turn on and off the background display.

1.4 Observable Objects

In Cappuccino, all the switches, links and the network are *observable* objects meaning that other objects can register themselves with the object to be notified of any state changes. This approach makes possible the automatic synchronization of different windows displaying the same information. However, if the number of observers of an object is very large (or the number of objects being observed by an object is very large, e.g., the display observes each switch and link for any changes) the number of generated *push* events can be very large resulting in slow GUI response times. In Cappuccino, most of the observable objects

provide methods for disabling and enabling notifications, so, if a large number of changes are to be made then it is prudent to turn off notifications for a brief while.

1.5 Cappuccino Loading Time

Cappuccino is a large software system consisting of hundreds of class files. The java run time system downloads and verifies the classes on demand, a process that can result in noticeable GUI slowdown when a class is referenced for the first time. Cappuccino solves this problem by allowing the pre-caching of the classes. The classes to be pre-cached are specified in *config.Caching* and Cappuccino runs a low priority thread in the background that forces the java run-time system to load and verify these classes.

1.6 File I/O over Internet

The applet version of Cappuccino does its file I/O on the server. The file i/o is done by starting a file server daemon that listens to a pre-determined port for file i/o requests. This mechanism does not go through firewalls but has the advantage of being considerably faster than the alternate CGI mechanisms that are firewall friendly.

1.7 Storing the Network as Objects

Cappuccino stores the network as an object instead of as text files. Textual representation of a network is, somewhat surprisingly, more compact than the object representation (typically, by a factor of 1.5 - 2).

1.8 Compilation in Java IDE

The compilation in the applet version of Java IDE is done on the server through a CGI script. This mechanism is considerably slower than the corresponding mechanism used in the application version. This is due to the inefficiencies of the CGI execution process and the network delays.

1.9 Optimizing the Best Star Algorithm

Cappuccino provides the *Best-Star* algorithm to generate the cheapest star network for the switches of a network given all the constraints. The algorithm uses a linear program to find the cost of each star which is very slow for reasonably sized networks. However, typically the best star network is required to generate a good topology to start with and the constraints are not very much important. The present implementation of the best star algorithm uses a considerably faster version (which does not use the linear program) if no set-pair constraints are present.

2. Key Contributions

Cappuccino provides a user friendly tool to design ATM Networks. This sub-section describes the key contributions and novel ideas demonstrated by the Cappuccino project.

2.1 Extensibility without Recompilation

Cappuccino allows run-time incorporation of algorithms without recompilation of the Cappuccino code. To implement this functionality, Cappuccino uses following novel approaches:

1. **Extensible Data Structures:** In Cappuccino, the concept of properties to achieve extensibility in data structures is introduced.

2. **Algorithms as Function Object:** Algorithms are usually thought of as procedural programs that operate on the data structures. Cappuccino presents the algorithms as function objects (an object-oriented approach), which are applied to the data.

3. **Zero GUI code in Algorithms:** In a similar project, this author had created a tool for graph visualization that used a function object approach for the algorithm representation. However, in that project, the addition of almost any algorithm required some GUI code to get parameters from the user. Cappuccino provides an ingenious solution to this problem in which an algorithm throws an exception if it does not get the parameters it requires. GUI mechanisms to get those parameters are built into Cappuccino.

2.2 Availability as an Applet and an Application

Cappuccino is a successful demonstration of software that can be used both as an applet and an application. Most of the code (except for some bootstrapping classes) is not aware of whether it is being executed as an applet or an application even though it does use services in which these two models differ. Cappuccino achieves this by providing abstractions for the functions in which the two models differ. The bootstrapping code plugs-in the required concrete classes while the rest of the code uses the abstractions to access the required services. The Cappuccino design was largely motivated by *ACE* [Ds01, Ds02] which also resolves incompatibilities by providing abstractions.

2.3 Document/ View/ Controller Approach

The Cappuccino design involves significant GUI components that interact with each other in complicated manners. The clean internal design of Cappuccino is a demonstration of the power of the Document/ View/ Controller approach [MNR97] in managing large GUI applications.

2.4 Web based Service Model

Cappuccino is available as a java applet and hence can be run in any JDK1.1 enabled browser. This frees the user from the typical download, install, and upgrade cycle. This is especially important due to the extensibility features in Cappuccino because an extensible system often means more frequent upgrades.

3. Project Statistics

Cappuccino consists of around 25,000 lines of java code distributed in over 300 classes defined in around 250 java files. Out of these, the java file server (JFS) accounts for nearly 14,000 lines that comprise about

80 classes. The remaining code of about 11,000 lines generates around 400 kbytes of classes that are sent over to the browser as an applet.

4. Future Work

4.1 Adding New Algorithms

The easiest enhancements in Cappuccino are done by adding new algorithms. Different kinds of algorithms can be added for theoretical studies or for improving the network design process. Some of the algorithms that can be added to the Cappuccino:

1. **Partitioning Algorithms:** Different kind of partitioning algorithms (Hexagonal, triangular etc.) can be added in Cappuccino.

2. **Random Network Generators:** These algorithms generate random networks that contain switches according to specific probability distributions.

4.2 Cappuccino as a Trusted Applet

Currently, Cappuccino is downloaded as an untrusted applet by the browser. Due to the security restrictions on the untrusted applets, Cappuccino provides the file i/o and compilation services on the server. The compilation is done by invoking a CGI script that is a slow process. To provide the file services, while ensuring the security of the server itself, Cappuccino uses JFS to define different accounts and their associated home directories. This has resulted in the need for tools for JFS server administration (e.g., account and directory management, etc.). Moreover, the JFS server, not being professional software, lacks robustness and uses a proprietary protocol for file transfer between applet and the server. This prevents the connections from going through the firewalls as well.

The right way to solve these problems is not to use the server based file-i/o but to rewrite the applet as a *signed* applet. If a Cappuccino user wants to permit local file I/O then he can grant those rights to the trusted applet.

This will have further benefits for the Cappuccino project:

1. It allows the user to develop (write/compile etc.) his algorithms in the IDE of his choice, thereby obviating the need of the Java IDE.

2. Presently, Cappuccino resolves applet/application differences by providing abstractions. Although, this approach is clean, it results in the minimum common denominator API's being available to the Cappuccino programmer. By making Cappuccino a signed applet, many of these differences will go away and so will be the need for these abstractions.

3. The user will see an improved GUI for file dialog and account management because these services will be provided by the native system.

4. The dependency on the JFS code will go away. JFS is a big piece of code (around 14,000 lines in java). It was written using JDK1.0.2 and it does not have a very nice GUI. Although, a lot of time was spent in its re-organization and interworking with Cappuccino, it still needs to be overhauled in some major ways.

5. File i/o and compilation performance will be much better (local vs remote).

4.3 A Configuration Tool for Cappuccino

Cappuccino design allows the flexibility in choosing the classes for the network representation, the algorithms to be shown in the choice lists, the different image files used in Cappuccino (for image buttons, icons etc.), the size of different windows, etc. However, presently these parameters are taken either from the text files or from the class methods. JDK1.1 introduces the concept of *Javabeans* that can be manipulated in GUI builder tools to create applications without writing any code. The different components in Cappuccino can be implemented as javabeans and a configuration tool (which can also be a javabean) can be used to configure Cappuccino easily.

4.4 Using JAR to Package Class Files

To run Cappuccino, the browser needs to download more than 300 classes and numerous image files. The browser downloads each of the these as a separate file through the web-server which results in significant loading delays. JDK1.1 defines the JAR format to package more than one class files and data files (e.g., image files) in a single archive that can then be downloaded efficiently by the browser. Cappuccino should be packaged as a .jar file to take advantage of this feature.

4.5 Support for Adding New Cost Models

Currently, Cappuccino assumes a linear link and switch cost model, i.e. the cost of a switch or a link increases linearly with its capacity, and the cost of a link also increases linearly with its length. It would be desirable to allow more general cost models in Cappuccino. One idea is to allow each switch and a link to have an associated cost model that can be changed by the user. The *network* package defines interfaces for these general cost models but these interfaces are not currently being used in Cappuccino.

```
public interface SwitchCostFunction {
      public double getCost(Switch s, double capacity);
}



public interface LinkCostFunction {
      public double getCost(Link l, double capacity);
}
```

The *getCost()* method of the *SwitchCostFunction,* returns the cost of the specified switch for the specified capacity. The *getCost()* method of the *LinkCostFunction,* returns the cost of the specified link for the specified capacity. These functions can also use any of the properties of the associated switch or link to arrive at the cost.

4.6 A Dialog Box for Algorithm Management

Currently, Cappuccino provides a file dialog box for adding new algorithm. The user has no way of deleting an algorithm once it is successfully added. A sophisticated dialog box is required which presents all the algorithms to the user and the user is allowed to put them in different choice lists.

4.7 Allowing Enumerated Types as Property Values

Currently, the properties that can be associated with a *PropertyHolder* object can either be a string or a number. The input for the properties is taken in text-fields. For some algorithms, the property needs to be an enumerated type (or boolean) which needs to be presented as a choice list.

4.8 Providing Selection Rectangle

Currently, Cappuccino allows the selection of switches and links, one by one. However, Menu options exist for selecting/deselecting all switches and links. It would be more user friendly if the user can draw a selection rectangle to select multiple switches and links at once. The facility to move all switches together can also result in conveniences to the user.

4.9 Numerous Performance Enhancements

1. Currently, when the user dimensions all the links, the linear program is run independently for each link. However, if the network and the constraints are symmetric, then the link dimensioning information is same for (s1, s2) and (s2, s1). Hence, link dimensioning algorithm need to be run only half the time.

2. Currently, whenever a dimensioning algorithm is executed, the routing information for the network is generated again. However, the routing algorithm is required to be re-run only if it is the information generated in the last run has become stale. However, presently, the link dimensioning takes "orders of magnitude" more time than routing algorithm and this enhancement is not going to improve speed significantly.

Credits and Acknowledgments

The constraints based approach to ATM network design was developed by Andrew Fingerhut in his Ph.D. thesis under the supervision of Professor Jonathan Turner. Cappuccino implements this approach and is based on the work done in *Decaf* which was implemented by Hongzhou Ma and me with help from Andrew Fingerhut and Rob Jackson. Cappuccino uses a modified version of the JFS file system to provide server based file i/o in the applet mode. The JFS was implemented by Jamie Cameron (jcameron@letterbox.com) at Monash University, Australia. In the design and implementation of Cappuccino, I have taken numerous ideas from the postings on *comp.lang.java.** news-groups.

Professor Jonathan Turner has been a good advisor, providing insight, advice and ideas for enhancing the overall quality of this software project. I wish to thank my former manager at Sun Microsystems, Dr. Abhay K. Parekh, who provided some valuable inputs regarding this tool. Hongzhou Ma deserves special thanks for converting some *Decaf* algorithms to Cappuccino and by providing general java related help. I also wish to thank my friends, Abhishek, Nikhil and Sridhar for providing ideas related to the network design tool.

My foremost and deepest respect goes to my parents, Late S. Bhagat Singh and Satwant Kaur, who have provided immeasurable motivation to achieve the very best in the life. I also wish to thank my brother, Narinderjeet Singh, and my sister, Baljeet Kaur, whose deep affection has always kept me in high spirits.

Appendix A: References

1. [BF77] Boorstyn and Frank, "Large Scale Network Topological Optimization." IEEE Transactions on Communications, Vol. COM-25, No.1, Jan 1977.

2. [Af94] Andrew Fingerhut, "Approximation Algorithms for Configuring Non-blocking Communication Networks." D.Sc. thesis, Washington University Computer Science Department, May 1994.

3. [ARSJ96] Andrew Fingerhut, Rob Jackson, Subhash Suri, and Jonathan S. Turner, "Design of Non-blocking ATM Networks." Washington University Computer Science Department WUCS-9603, 1/96

4. [ASJ96] Andrew Fingerhut, Subhash Suri, and Jonathan S. Turner, "Designing Least Cost Non-blocking Broadband Networks." Washington University Computer Science Department WUCS-9606, 1/93

5. [Jt01] Jonathan Turner, "Design and analysis of ATM Switching Systems." Course notes for CS 577. Department of Computer Science, Washington University, St. Louis.

6. [HIJ01] Hongzhou Ma, Inderjeet Singh, Jonathan Turner, "Constraints based Design of ATM Networks, an Experimental Study." WUCS-97-15 Technical Report, Department of Computer Science, St. Louis.

7. [Ak01] Aaron Kershenbaum, "Telecommunications Network Design Algorithms." McGraw Hill, 1993.

8. [Ret01] R.E. Tarjan, "Data Structures and Network Algorithms." SIAM, 1983.

9. [TCR01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introductin To Algorithms, MIT Press.

10. [GHJV95] Gamma, Helm, Johnson, Vlissides, "Design Patterns: Elements of Reusable Object Oriented Software." Addison-Weseley, 1995.

11. [MNR97] Robert C. Martin, James W. Newkirk, Bhama Rao, "Taskmaster: An Architecture Pattern for GUI Applications" C++ Report, March 1997

12. [Dm96] David McGeary, "Graphic Java -- Mastering the AWT," SunSoft Press, 1996.

13. [JDK1.1] JDK 1.1 Documentation, JavaSoft Inc.

14. [JGL2.0] JGL 2.0 Documentation, Object Space Inc.

15. [Ds01] Douglas C. Schmidt, "Distributed Software System Development." Course notes for CS544. Department of Computer Science, Washington University, St. Louis.

16. [Ds02] Douglas C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in Proceedings of the 6th USENIX technical conference, (Cambridge, Massachussetts), USENIX Association, April 1994.

Appendix B: Glossary

1. Singleton: Singleton is a design pattern [GHJV95] that is used when only one system wide instance of an object is desired.

2. Iterator: An iterator is a design pattern [GHJV95] that provides access to a sequence of elements of a *Container*, without providing the power to add or remove elements. Hence, an iterator provides a read-only way of looking into a container.

3. Pre-conditions of a method: The conditions that must be true for the method to run correctly.

4. Post-conditions of a method: The conditions that must be true, if the method is correctly implemented and the pre-conditions are met.

5. Class methods: The methods that are associated with the class itself and not with any instance of the class.

6. IDE: IDE stands for Integrated Development Environment. It refers to a software development environment consisting of an editor and a compiler.

7. Java Beans: A javabean is a software component that can be manipulated in a GUI builder tool. By convention, a javabean defines various *setter* and *getter* methods to provide access to its different properties that are presented to the user by a GUI builder tool.

Appendix C: Cappuccino Packages

1. The Default Package

1.1 The Main Classes

1.1.1 CappuApplet

This class creates the login panel in the browser that can be used to create Cappuccino windows. It also initializes the *FileHandler* and *HelpEngine* singleton with applet specific concrete classes. It also creates a clipboard specific to the Cappuccino applet.

1.1.2 CappuApplication

This class creates a single Cappuccino window that is an instance of *gui.GUIManager*. It also initializes the *FileHandler* and *HelpEngine* singleton with application specific concrete classes. It also sets the clipboard to be the system wide clipboard.

1.2 The Concrete File Handlers

1.2.1 AppletFileHandler

AppletFileHandler is the applet specific concrete class for the *FileHandler* singleton. It uses *java.net* package to open files on the server and uses CGI scripts for writing files. It uses *getImage()* method of *java.applet.Applet* for obtaining images corresponding to image files.

1.2.2 ApplicationFileHandler

ApplicationFileHandler is the application specific concrete class for the *FileHandler* singleton. It uses *java.io* package for implementing various file operations.

1.3 The Concrete Help Engines

1.3.1 AppletHelpEngine

AppletHelpEngine is the applet specific concrete class for the *HelpEngine* singleton. If the help key is a valid URL then it displays it in a frame in the browser. Otherwise, it displays the help key in a message box.

1.3.2 ApplicationHelpEngine

ApplicationHelpEngine is the application specific concrete class for the *HelpEngine* singleton. If the help key is a valid URL then it first tries to show help in an already running browser. If a browser is not already running then it tries to start a browser with the desired webpage. If all attempts fail or the help key is not a valid URL then it displays the help key in a message box.

1.4 The Concrete Compiler Classes

1.4.1 AppletCompiler

This class is the applet specific concrete class for the *SourceCodeCompiler* singleton. It uses a CGI script to compile the class and get the results of the compilation.

1.4.2 ApplicationCompiler

This class is the application specific concrete class for the *SourceCodeCompiler* singleton. It invokes the *javac* compiler on the specified class file.

1.5 Pre-caching classes

Cappuccino is a large software system and hence consists of a large number of classes. In most implementations of java, the first reference to a class suffers significant delays due to on-demand downloading and verification of the class files. This delay is more pronounced in an applet environment where the classes are downloaded over the network. Cappuccino reduces the overall downloading time of the classes by launching a separate low priority thread, *PreCacher,* which pre-caches the list of class and image files specified in *config.Caching.*

1.6 Classes Defining Properties

1.6.1 AppletProperties

This class defines some properties that are specific to the applet. The other applet specific classes use these properties to configure themselves.

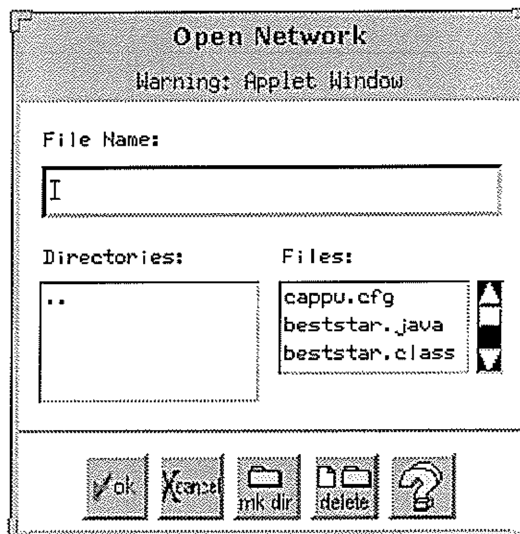
1.6.2 ApplicationProperties

This class defines some properties that are specific to the application. The other application specific classes use these properties to configure themselves.

1.7 Concrete Classes for File Dialog

1.7.1 AppletFileDialog

The applet version of Cappuccino can not use the default file dialog box that comes with the java run time system because the file dialog needs to display the directory information corresponding to the user's home directory on the server. This class interacts with the JFS server to obtain the directory information and display it in the dialog box. It also provides buttons to create and delete directories.



1.7.2 ApplicationFileDialog

This class represents the file dialog used for the application version of Cappuccino. It uses *java.awt.FileDialog* to implement this functionality.

2. The *general* Package

This section describes those implementation details and the classes that are not covered in the *Extending Cappuccino* chapter.

2.1 Class *Debug*

This class defines the debugging policy in Cappuccino. It defines different methods for handling different types of errors:

1. *error():* This method is invoked to report a fatal bug that will not be meaningful to the Cappuccino users or administrator.

2. *warning()*: This method is invoked to report a non-fatal bug that will not be meaningful to the Cappuccino users or administrator.

3. *errorAdmin():* This method is invoked to report a fatal bug that will be meaningful to the Cappuccino administrator but not to the users.

4. *warnAdmin()*: This method is invoked to report a non-fatal bug that will be meaningful to the Cappuccino administrator but not to the users..

5. *errorUser():* This method is invoked to report a fatal bug that will be meaningful to the Cappuccino users.

6. *warnUser()*: This method is invoked to report a non-fatal bug that will be meaningful to the Cappuccino users..

Each method comes in two forms, one that takes a string parameter representing the message and the other that takes a message string as well as an argument of the type *Exception*. The second form is available as a convenience because most of the errors in Cappuccino result from unexpected exception.

2.2 Class *HelpEvent*

This class represents an event that represents a request to show help. A help event has a help key associated with it that either represents a URL or the help text itself.

2.3 Class *HelpHintEvent*

This class represents an event that is a request to show a help hint. An event of this type has a single line help hint associated with it.

2.4 Interface *HelpHintListener*

This interface describes a listener for the help hint events. It provides a method, *showHint()*, to act on the specified help hints.

2.5 Interface SourceCodeCompiler

This interface represents an abstraction for providing java source code compilation facilities. It defines a *compile(fileName)* method that compiles the specified file and returns the compilation results. It also defines a *getClassName()* method that returns a class name corresponding to a file name.

2.6 Class *ChangeCommand*

Cappuccino uses the *push* model of the Document/View/Controller paradigm. This class represents the change being pushed. It defines methods to identify the change that has occurred and the old value of the field that has changed. So, the observers need not pull whole state of the document every time a change occurs.

2.7 Exception Classes

1. *CappuInternalError:* This class represents an exception that is thrown if a fatal error occurs in Cappuccino. This class derives from *java.lang.RuntimeException* to obviate the need for catching this exception everywhere.

2. *IncompatibleClassException:* This exception is thrown to indicate an unexpected meta-class. It derives from *java.lang.Exception.*

3. *InvalidFileOperationException:* This exception is thrown to indicate an unsuccessful file operation. It derives from *java.lang.Exception.*

4. *ParseException:* This exception is thrown when an error occurs while parsing text to create desired object forms.

3. The *network* Package

This section describes those implementation details and the classes that are not covered in the *Extending Cappuccino* chapter.

3.1 Class *NetworkImpl*

This class is the default concrete implementation of the *Network*. It allows the user to select different container representations for the switches and links. The container so chosen must implement *jgl.Container* interface.

3.2 Class *ViewInfo*

This class contains the complete view information that is saved in a file. It contains data members for the network, constraints, routing and dimensioning information and the persistent properties of the *NetworkView*.

3.3 Class *ConfigBean*

This class represents the configuration java bean for this package. It provides mechanisms to select different representation of the *Network*, the switch and link containers for the default Network implementation and other configurable properties of the package.

4. The *stk* Package: A GUI Toolkit

The Cappuccino project was started when JDK1.1 was not available and hence JDK1.0 was the only implementation choice. The initial design of Cappuccino used a toolkit, called *GraphicJava Toolkit (GJT)*, for providing an aesthetically pleasing graphical user interface. Later, Cappuccino was migrated to JDK1.1 which differs significantly (and incompatibly) from JDK1.0 AWT event model. GJT was originally provided as an example toolkit along with [Dm96] and was, unfortunately, not revised for JDK1.1. Hence, Cappuccino provides its own GUI toolkit called the STK (Small toolkit) which contains its own JDK1.1 versions of some GJT classes along with its own set of GUI components.

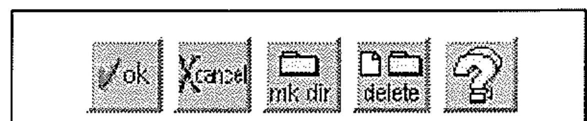
4.1 Revised versions of GJT classes

4.1.1 Border

A panel containing a single component around which a border is drawn. The component can itself be a panel containing multiple components.

4.1.2 ButtonPanel

A button panel employs a *java.awt.BorderLayout* to layout a separator in the north and a panel to which buttons can be added in the center.



4.1.3 ColumnLayout

This is a layout manager to lay out components in a column. The orientation of the components within a cell can be specified in the constructor.

4.1.4 RowLayout

This is a layout manager to lay out components in a row. The orientation of the components within a cell can be specified in the constructor.

4.1.5 DrawnRectangle

A rectangle that draws itself inside a component.

4.1.6 EtchedBorder

An extension of a *Border* that draws an itched border.

4.1.7 EtchedRectangle

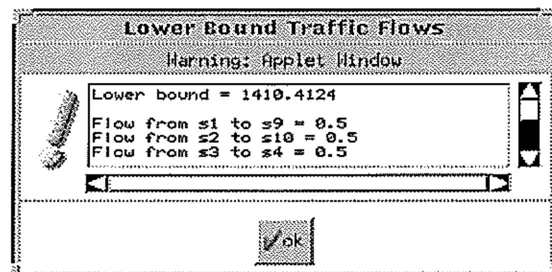
A *DrawnRectangle* that draws an etched border.

4.1.8 Etching

This class defines some constants for etching. It is not instantiable.

4.1.9 ImageCanvas

A canvas that displays an image.



4.1.10 MessageDialog

A MessageDialog is a non-modal dialog box to display a message.

4.1.11 Orientation

This class defines constants for orientations and alignments. It is not instantiable.

4.1.12 Separator

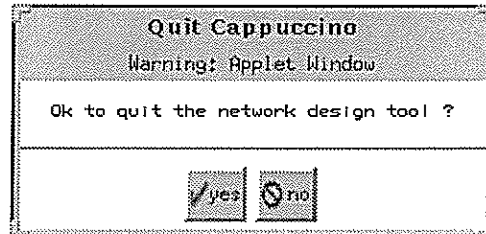
A separator that is drawn either vertically or horizontally depending on how it is laid out. Can be drawn etched-in or etched out and with varying thickness.

4.1.13 QuestionDialog

A dialog that presents a prompt and a textfield into which a reply can be entered.

4.1.14 YesNoDialog

A modal dialog box that prompts a question that is answered by selecting either a yes or no button.



4.2 New Classes in *stk* Tool-kit

4.2.1 PowerLayout

PowerLayout is an easy to use layout manager built on top of *java.awt.GridBagLayout.*

4.2.2 Circle

This is a class on the lines of *java.awt.Rectangle* for circles.

4.2.3 ExclusiveImageButtonPanel

A panel consisting of sticky image buttons of which only one can be pressed at one time. It implements *java.awt.ItemSelectable* interface and hence behaves like a choice list.

4.2.4 ExclusiveMenu

A *java.awt.Menu* consisting of *java.awt.CheckboxMenuItems* of which only one can be selected at one time. It implements *java.awt.ItemSelectable* interface and hence behaves like a choice list.

4.2.5 ImageButton

An image button is a button with an image on it. The button can be specified to be either sticky or springy. It is springy by default.

4.2.6 ImageButtonPanel

A panel containing image buttons arranged either horizontally or vertically.

4.2.7 StatusBar

This class implements a status bar to show single line messages.

4.2.8 StatusBarController

A class that implements *general.HelpHintListener* interface and shows the help hint events in a status bar.

4.2.9 Util

This class contains a collection of utilities to create image buttons, menu options, waiting for images, etc.

5. The *gui* Package

Cappuccino allows users to work with multiple networks at the same time. Each of the windows is launched in its own thread and is independent of the other windows. However, all windows share the same *FileHandler* and *HelpEngine* singletons. All windows also use the same clipboard (the system-wide global clipboard in case of the application version) for cut and paste. For ease of network creation, Cappuccino provides various modes of operations. In the switch or link placement mode, the user can place switches and links (with default parameters) with just a click of the mouse. The user can also select groups of switches and links and do operations on them. Cappuccino presents all the available algorithms in different choice lists.

5.1 GUI Entities

5.1.1 GUIManager

This class creates and displays the Cappuccino GUI. It launches the Algorithm IDE, the different views of the network (*NetworkGUIView, NetworkTextView, NetworkSwitchView,* and *NetworkLinkView*). It also manages the status bar.

5.1.2 ViewInfoManager

This class manages the master information for a network view that is displayed in the multiple windows. It also keeps an instance of the class *AlgoExecuter.* Different windows displaying the same network view keep a reference to their master *ViewInfoManager.*

5.1.3 IOGUIManager

This class provides the GUI corresponding to the I/O functions of Cappuccino. An instance of this class is kept by all windows. It handles all the menu-items of the file menu except file|print.

5.1.4 AlgoGUIManager

This class managers the GUI corresponding to the algorithms. It handles events corresponding to the menu-bar and the choice lists of the algorithms. It also provides the GUI for adding new algorithms. Different window displaying the same network view create instances of this class to manage their algorithm GUI.

Cappuccino is designed to allow incorporation of a large number of algorithms in it. *AlgoGUIManager* displays all the available algorithms (specified in a text file) as choice lists and menu options. However, it loads and verifies any algorithm only on its first use. For subsequent uses, the algorithm object is reused. All commonly used algorithms should be specified in the class *config.Caching* for reducing delays in the first invocation of the algorithm.

AlgoGUIManager shows the algorithms in choice lists and menu items. It also maintains different choice list and menu in sync.

5.1.5 ClipboardManager

This class manages the Cappuccino clipboard and is a singleton. It provides methods to place and obtain a network object (or a textual representation of a network) on the clipboard.

5.1.6 ConfigBean

The class is the java bean for configuring the gui package.

5.2 Common GUI Components

5.2.1 Menu items for Algorithms

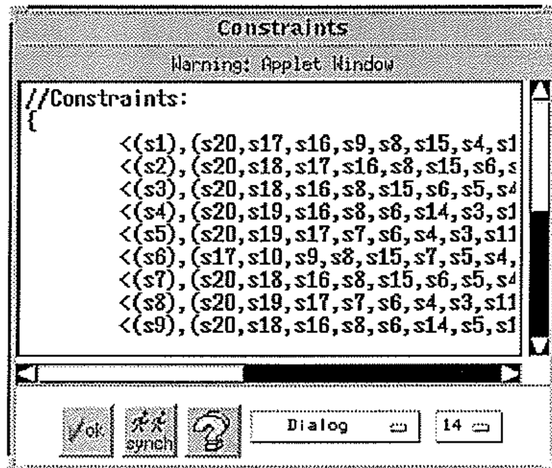
Cappuccino shows different algorithms as menu-items. To differentiate the received menu events, Cappuccino defines the menu items corresponding to different algorithms as separate classes:

1. *AlgoMenuItem* is used to show a menu item corresponding to a topology, dimensioning, constraints and super algorithms.

2. *RoutingAlgoMenuItem* is used to show a checkbox menu item corresponding to a routing algorithm.

5.2.2 Class ConstraintsWindow

The class represents the GUI form of the set of constraints associated with a network view. The constraints are represented as text. This class observes the *Constraints* object and reflects any changes appropriately. If the user presses the *update* button then it parses the text and creates a constraint object

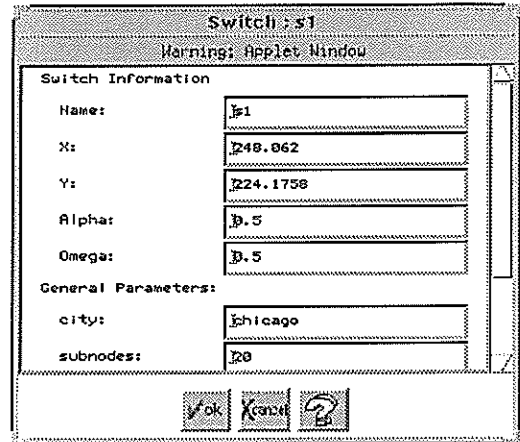


5.2.3 FlexibleDialog

This class is sub-classed to create various information dialog boxes, for example, *SwitchInfoDialog*, *LinkInfoDialog*, etc. It uses *stk.PowerLayout* to lay out rows of label, textfield pairs. It determines if a scrollbar is necessary and automatically fits it if required.

5.2.4 *InputController*

This class is used to monitor a text field for any changes. It takes a text field and an object and shows the textual representation of that object in the text field. If the text field is changed by the user then it constructs an object of the same type using the java reflection API.

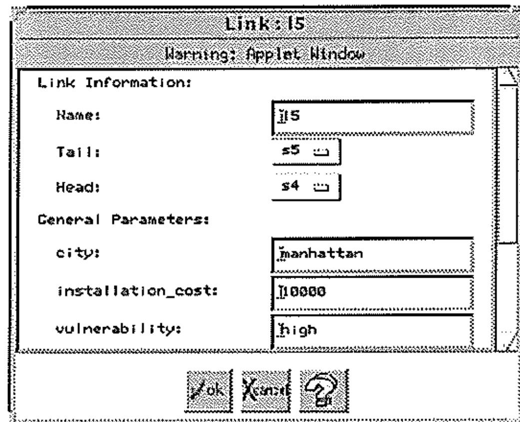


5.2.5 *SwitchInfoDialog*

This class shows all the parameters of a switch in a dialog box and allows the user to modify them. This class derives from *gui.FlexibleDialog*. If the user presses the ok button, then it parses all the parameters and sets the corresponding switch parameters.

5.2.6 *LinkInfoDialog*

This class shows all the parameters of a link in a dialog box and allows the user to modify them. This class derives from *gui.FlexibleDialog*. If the user presses the ok button, then it parses all the parameters and sets the corresponding link parameters.



5.2.7 *PropertiesDialog*

This class represents a dialog box that shows a list of *properties* and allows the user to modify them.

5.3 Visual Representation of the Network

5.3.1 *NetworkGUIView*

The class *NetworkGUIView* implements the *NetworkView* interface and represents a network visually. It observes the network for addition or deletion of switches and updates the screen accordingly. It also supports drawing of a background image and provides various service methods to its controller, *GUIViewController*. It uses following four classes to draw the network:

1. Interface *NetworkGUIInfo:* This interface provides methods for transforming the coordinates of switches from the screen coordinates to the actual normalized values. *NetworkGUIView* implements this interface.

2. Class *GUISwitch:* This class represents the graphical image of a switch. It observes the switch for changes and modifies the screen image accordingly.

3. Class *GUILink:* This class represents the graphical image of a link. It observes the link for changes and modifies the screen image accordingly.

5.3.2 GUIViewController

This class implements the *Controller* for the *NetworkGUIView.* It runs as a finite state machine and maintains three modes, *Select, Add Switch,* and *Add Link.* Each of these modes also runs as finite state machines. *GUIViewController* receives all the GUI events and forwards them to the appropriate state machine. It intercepts the mouse right click events and displays the properties of the selected items.

1. Class *StateMachineForSelectMode:* This state machine handles the *Select* mode. It has 2 states:

   a) SELECTING_ITEMS: In this state, a click on a switch or link selects it while deselecting earlier selections. A click with control key down keeps the earlier selections. A double click on a switch changes state to MOVING_SELECTED_ITEMS.

   b) MOVING_SELECTED_ITEMS: In this state, on a mouse drag event, the position of the selected switch is changed to the coordinates of the event. A mouse release event changes the state back to SELECTING_ITEMS.

2. Class *StateMachineForAddSwitchMode:* A mouse click results in a switch being created using the default name and position of mouse event.

3. Class *StateMachineForAddLinkMode:* This state machine acts on mouse clicks. It has two states:

   a) SELECT_HEAD: In this mode, a mouse click on a switch, selects the head of the link and changes the state to SELECT_TAIL.

   b) SELECT_TAIL: A click on a switch results in a link being created joining the head and tail and changes state to SELECT_HEAD. A mouse click on empty screen area changes state to SELECT_HEAD.

5.3.3 AlgoExecuter

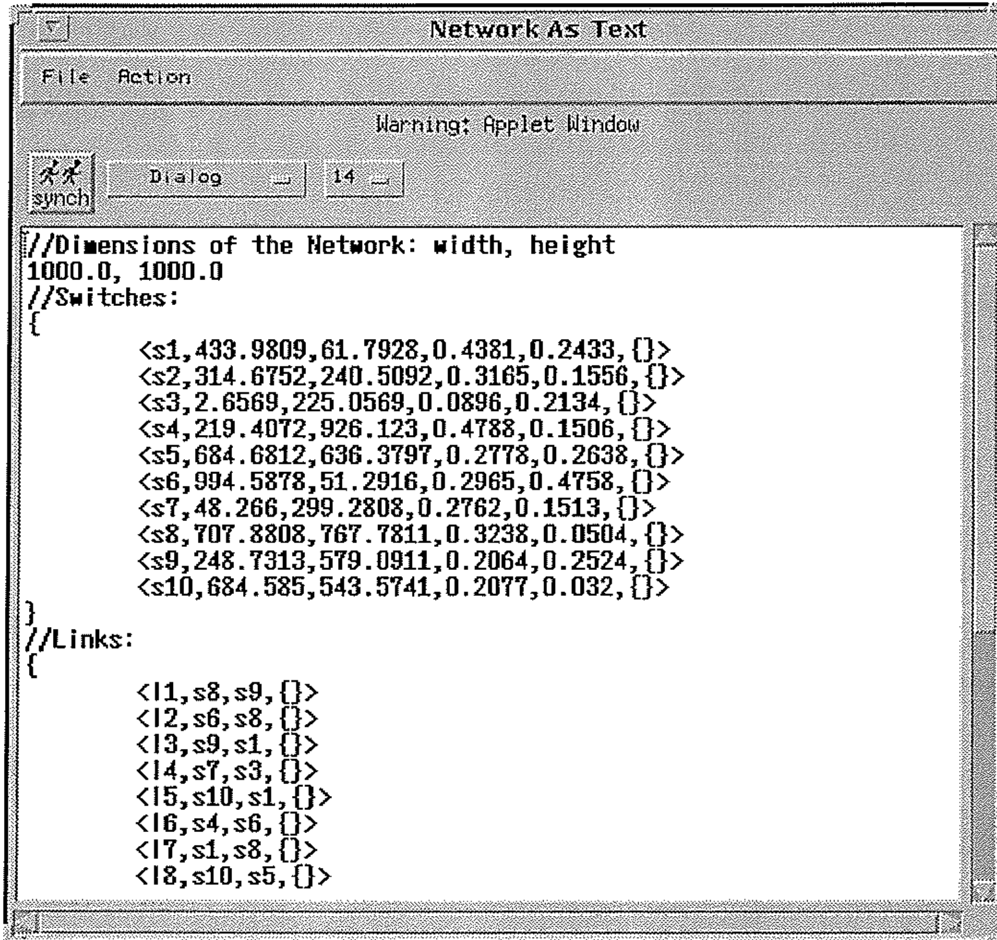
The class *AlgoExecuter* handles the execution of all the algorithms. To increase the speed of algorithm execution, it disables change notifications to the observers of the network view before executing an algorithm. After algorithm completion, it enables the notification and refreshes all the observers by forcing a *pull.* This disable/enable notification can be disabled by a GUI option to allow algorithm animation.

5.3.4 GUIViewManager

GUIViewManager manages all these entities and also handles the interactions with the clipboard. It also handles various other menu options (showing background image, zooming in and out, etc.) that affect the view being shown on the screen.

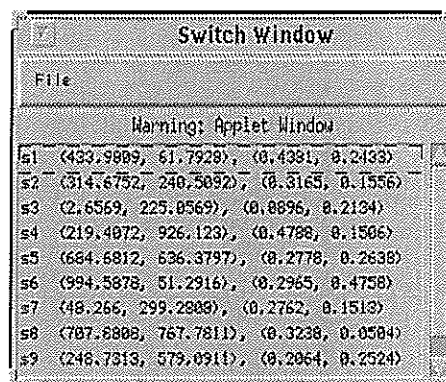
5.4 Textual Representation of a Network

Cappuccino provides a textual view of the network as well. In this view, the network can be edited as a text file. The *Action|Update* menu command is used to parse the text and create the actual network out of it.



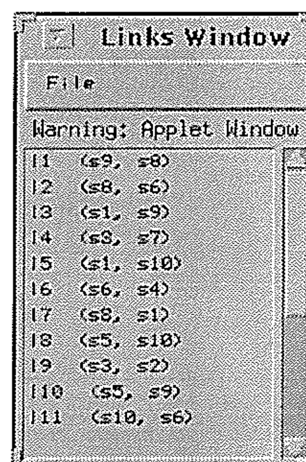
Textual Representation of the Network

5.5 Class *NetworkSwitchView*



In this view of the network, all the switches present in the network are shown in a list box. The user can select multiple items in the list box and view their properties as *SwitchInfoView* dialog boxes. Any change made in a switch is reflected back in the network. This class observes the network and handles any changes in the switches appropriately.

5.6 Class *NetworkLinkView*



In this view of the network, all the links present in the network are shown in a list box. The user can select multiple items in the list box and view their properties as *LinkInfoView* dialog boxes. Any change made in a link is reflected back in the original network. This class observes the network and handles any changes in the switches and links appropriately.

6. The *editors* Package

6.1 Integrated Development Environment for Algorithm Creation

Cappuccino provides run-time creation and incorporation of new algorithms.
This implies that cappuccino needs to provide basic capabilities for writing, compiling, verifying and invoking a java class. However, the providing a good development environment for writing java code is not one of the main goals of Cappuccino. Ideally the best thing would be to have the capability of invoking the IDE of the user's choice. However, it is not possible for following reasons:

1. One of the key requirement in any editor is the capability to do file operations. However, applets and applications have different file i/o capabilities, and any existing application will not have the capability to work in both applet and application mode.

2. An IDE also needs to provide facilities for compiling a java class and viewing its results. For an applet case, the compilation is to be invoked on the server whereas for an application it will be done on the local disk. Again, most of the existing IDEs run only as a java application.

In Cappuccino, these design forces were resolved by making the IDE independent of Cappuccino to the maximum possible extent. The goal is to provide an IDE that stands on its own.

The default IDE in Cappuccino is called *JavaIDE*. It uses *FileHandler* and *HelpEngine* singletons for file operations and showing help. It provides the compilation facilities through a *SourceCodeCompiler* interface. It uses the *stk* package for creating its GUI.

7. The *TopologyAlgos* Package

This package contains all the algorithms that modify the topology of a network. At present, there are 8 algorithms available in this package:

7.1 *RandomSwitchAdder*

This algorithm is used to generate switches at random location. It takes five view parameters:

1. *NumSwitchesToAdd:* Specifies the number of switches to add.
2. *DefaultSwitchNameTag.* Specifies the default name tag for the switch, e.g., if the tag is "s" then name of switches will be s1, s2, etc.
3. *DefaultSwitchNameIndex:* Specifies the starting index to attach to the name tags.
4. *DefaultAlpha:* Specifies the maximum value of the alpha to use for the switches.
5. *DefaultOmega:* Specifies the maximum value of the omega to use for the switches.

7.2 *RandomLinkAdder*

This algorithm is used to generate random links in the network. It takes three view parameter:

1. *MaxNumLinksToAdd:* The algorithm tries to add the specified number of links until the network is complete.
2. *DefaultLinkNameTag.* Specifies the default name tag for the link, e.g., if the tag is "l" then name of links will be l1, l2, etc.
3. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.3 *CompleteNetwork*

This algorithm puts all possible links in the network. It takes two view parameters:

1. *DefaultLinkNameTag.* Specifies the default name tag for the link.
2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.4 *LinkComplement*

This algorithm generates the complement links in the network, i.e., it deletes all the existing links and puts in all those links that were not present earlier. It takes two view parameters:

1. *DefaultLinkNameTag.* Specifies the default name tag for the link.

2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.5 *DelaunayTriangulation*

This algorithm generates the Delaunay Triangulation topology for the network. The properties of Delaunay Triangulation are explored in Joseph O'Rourke's *"Computational Geometry in C"*. The algorithm is implemented in C by Steve Fortune. It deletes the links that are not required in the triangulation. It uses a CGI script to invoke the C implementation of this algorithm at the server. It takes two view parameters:

1. *DefaultLinkNameTag.* Specifies the default name tag for the link.

2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.6 *MinimumSpanningTree*

This algorithm generates the minimum spanning tree based on the Euclidean distance between the switches. It uses Kruskal's algorithm(cf. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Introductin To Algorithms, p505) to find the minimum spanning tree. Those existing links that are part of the MST are retained. All other links are deleted from the network. It uses *jgl.PriorityQueue* as a heap for obtaining links in the increasing order of their lengths. It takes two view parameters:

1. *DefaultLinkNameTag.* Specifies the default name tag for the link.

2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.7 *Star*

This algorithm generates a star network rooted at the selected node. It takes two view parameters:

1. *DefaultLinkNameTag.* Specifies the default name tag for the link.

2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

7.8 *SetPrimaryNetwork*

1. This algorithm sets the present network to be the primary network. The primary network is used by the *AdjustPrimaryNetwork* algorithm to selectively improve topology.

7.9 *AdjustPrimaryNetwork*

When this algorithm is invoked for the first time, it sets the current network to be the initial network. On subsequent invocations, it uses the links present in the current network to decrease the shortest path distance between the switches in the initial network. It takes two view parameters:

1. *AdjustInitialNetworkRatio:* This is the ratio by which a link should improve at least one shortest path distance in the initial network to get added.

2. *InIncreasingOrder:* This boolean parameter specifies if the links are to added in the increasing order of their lengths.

8. The *ConstraintsAlgos* Package

This package contains algorithms that can be used to generate constraints for a network. Right now, it contains two algorithm:

8.1 The Class *PairwisePercentage*

This algorithm generates set-pair constraints which is proportional to the relative source and sink capacity of switch pair.

The constraints are generated in the following manner:

For each pair of switches (u, v):

*f(u, v) = alpha(u) * omega(v) / (sum(omega) - omega(u))*

*g(u, v) = omega(v) * alpha(u) / (sum(alpha) - omega(v))*

*Mu(u, v) = RelaxFactor * min(f(u, v), g(u, v))*

SetPairConstraint(v, v) = { u, v, Mu(u, v), Mu(v, u) }

It takes one view parameter, *RelaxFactor.*

8.2 The Class *Localized*

This class implements the localized traffic constraints. It takes four view parameters:

1. *Percentage:* This is the percentage of the traffic that must go to the local switches.

2. *Radius:* The distance within which a switch is considered local.

3. *Minimum_Number_of_Neighbors:* The minimum number of neighbors a switch must have. If the number of neighbors of a switch within the specified radius is less than this number then the radius is increased for the switch to include more switches.

4. *Maximum_Number_of_Neighbors* The maximum number of neighbors a switch can have. If the number of neighbors within the specified radius is larger than this number then the radius is decreased to exclude some switches.

For each switch, the algorithm generates the set-pair constraints for the switches in the local set and the switches in the remote set.

9. The *RoutingAlgos* Package

This package contains the algorithms that are used to generate routing information for a network. At present following algorithms are present in this package:

9.1 *ShortestPathRouting*

This algorithm generates the routing information based on the assumption that the traffic between two switches is routed along the shortest path between them. It runs the all pair shortest path algorithm to generate the shortest path links for each pair of nodes. Then, it generates *RoutingInfo* by finding the list of node pairs that use a particular link.

9.2 DistributedRouting

This algorithm takes a set of switches and a set of links and distribute the traffic between switches among different routes. The user can specify the maximum number of routes to use. It takes three view parameters:

1. *Number_of_Routes_to_Use:* This parameter specifies the maximum number of different routes to use for routing traffic.

2. *Upper_Limit_on_the_ratio_of_longestRoute/shortestRoute:* This parameter specifies the maximum allowed ratio of the traffic routed on the longest and the shortest route.

3. *Traffic_Distribution:* This string parameter specifies the relative weights of the different routes for distributing traffic.

10. The *DimensioningAlgos* Package

This package contains algorithms that are used to dimension links of a network. At present it contains the following algorithms:

10.1 *UseLinearProgram*

This algorithm uses linear programming to generate dimensioning information for a network, given routing information and a set of constraints. It uses a CGI script to run linear programming code

written in C. An introduction to linear programming dis available at http://www.mcs.anl.gov/home/otc/Guide/faq/. The code is written by Michel Berkelaar(michel@es.ele.tue.nl) and is available at ftp://ftp.es.ele.tue.nl/pub/lp_solve/. For every link l, the capacity of the link is the maximum value of

sigma(x[i][j])

where *x[i][j]* are traffics route through link *l* under the chosen routing algorithm. So, this is a linear programming problem, with objective function:

max: sigma(x[i][j]), summation over all traffics routed through link *l*

and constraints:

sigma(x[i][j]) over j < alpha[i]

sigma(x[i][j]) over i < omega[j]

and all those set-pair-constraints.

11. The *common* Package

This package is intended to contain some common classes for all three types of algorithms. Right now it contains following classes:

11.1 The Class *DisjointSetElement*

This class defines a data structure for efficient union-find operation.

11.2 The Classes *ShortestPathUtils* and *ShortestPathInfo*

These classes provide algorithm to generate shortest path information for a network.

12. The *SuperAlgos* Package

This package contains algorithms that combine the four-step network design process used in Cappuccino.

12.1 The Class *BestStar*

This algorithm iterates over all possible star networks in the network and chooses the one that has the least link cost. Given *N* switches, each switch *i* has source termination limit, *alpha,* and destination termination limit, *omega*. The distance between switches *i* and *j* is *dist[i][j]*. If switch *c* is center of the star network, the cost is

SIGMA(dist[j][c](out[j]+in[j])), summation is over j*

When there is no set-pair-constraints, out[j] and in[j] is simply:

out[j] = min(alpha[j], sigma(omega[i])-omega[j])

in[j] = min(omega[j], sigma(alpha[i])-alpha[j])

When there are some set-pair-constraints, out[j] and in[j] should be calculated by a link dimensioning considering all the constraints(c.f. page 58 of JAF's PhD thesis). However, the algorithm become considerably slower and hence, this implementation of best star always ignores the set-pair constraints. It takes two view parameters:

1. *DefaultLinkNameTag:* Specifies the default name tag for the link.

2. *DefaultLinkNameIndex:* Specifies the starting index to attach to the name tags.

12.2 The Class *LowerBound*

This algorithm generates a lower bound for the network given the source and the sink capacities and the set-pair constraints. It displays the traffic flows for the lower bound in a message box.

13. The *SwitchCostAlgos* Package

This package contains algorithms that define different cost models for switches. These algorithms implement *network.SwitchCostFunction* interface, which defines costs of a switch for different capacities. Right now, it contains one algorithm. However, this package is currently not being used in Cappuccino.

13.1 The Class *LinearSwitchCost*

This algorithm represents the linear switch cost model, i.e., the cost of each switch is linearly proportional to its total source and sink capacity. Moreover, the cost is independent of the actual switches.

14. The *LinkCostAlgos* Package

This package contains algorithms that define different link cost models. These algorithms implement *network.LinkCostFunction* interface, which defines costs for different capacities for each link. Right now, it contains following algorithms. However, this package is currently not being used in Cappuccino.

14.1 The Class *LinearLinkCost*

This algorithm represents the linear link cost model, i.e., the cost of each link is linearly proportional to its capacity and euclidean length. Moreover, the cost is independent of the actual links.