Washington University in St. Louis

Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2003-75

2003-12-04

Resource Configuration and Network Design in Extensible Networks

Sumi Y. Choi

The goal of packet-switched networks has conventionally been delivering data to users. This concept is changing rapidly as current technologies make it possible to build network processing engines that apply intermediary services to data traffic. This trend introduces an extensive range of ways to develop and operate applications by allowing processing services customized for applications' needs at intermediate network users, as it can relieve individuals from the need to acquire, install, and maintain software in end systems to perform required functions. As such network services become more widely used, it will become increasingly important for service providers to have... **Read complete abstract on page 2.**

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

Choi, Sumi Y., "Resource Configuration and Network Design in Extensible Networks" Report Number: WUCSE-2003-75 (2003). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/1121

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

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1121

Resource Configuration and Network Design in Extensible Networks

Sumi Y. Choi

Complete Abstract:

The goal of packet-switched networks has conventionally been delivering data to users. This concept is changing rapidly as current technologies make it possible to build network processing engines that apply intermediary services to data traffic. This trend introduces an extensive range of ways to develop and operate applications by allowing processing services customized for applications' needs at intermediate network users, as it can relieve individuals from the need to acquire, install, and maintain software in end systems to perform required functions. As such network services become more widely used, it will become increasingly important for service providers to have effective methods to configure applications sessions so that they use resources efficiently. On the other hand, it is equally important to design such extensible networks properly in order to ensure desirable performance of applications. This dissertation addresses these two key problems that arise in operation and provisioning extensible networks: configuring application sessions and designing extensible networks. First, we present a general method, called layered networks, for the problem of configuring application sessions that require intermediate processing. The layered network method finds optimal configurations by transforming the session configuration problem into conventional shortest path problem. We show, through a series of examples, that the method can be applied to a wide variety of situations. We also discuss how to configure applications that require reserved capacity and propose effective heuristic algorithms that are based on the layered network method. Second, for designing extensible networks, we generalize the constraint-based network design methods originally developed for conventional networks. We show how to incorporate arbitrary requirements that are allowed by extensible networks in a flexible and general way. We also show how to extend the original framework to dimension both processing resources and link bandwidth. These results have been incorporated into software packaged the Extensible Network Planner (XNP).

Washington University in St. Louis

School of Engineering & Applied Science

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

RESOURCE CONFIGURATION AND NETWORK DESIGN IN EXTENSIBLE NETWORKS

by

Sumi Y. Choi

Prepared under the direction of Professor Jonathan S. Turner

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

December, 2003

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

RESOURCE CONFIGURATION AND NETWORK DESIGN IN EXTENSIBLE NETWORKS

by Sumi Y. Choi

ADVISOR: Professor Jonathan S. Turner

December, 2003
Saint Louis, Missouri

The goal of packet-switched networks has conventionally been delivering data to users. This concept is changing rapidly as current technologies make it possible to build network processing engines that apply intermediary services to data traffic. This trend introduces an extensive range of ways to develop and operate applications by allowing processing services customized for applications' needs at intermediate network nodes. The provision of such services is potentially a significant benefit for network users, as it can relieve individuals from the need to acquire, install, and maintain software in end systems to perform required functions. As such network services become more widely used, it will become increasingly important for service providers to have effective methods to configure applications sessions so that they

use resources efficiently. On the other hand, it is equally important to design such extensible networks properly in order to ensure desirable performance of applications. This dissertation addresses these two key problems that arise in operating and provisioning extensible networks: configuring application sessions and designing extensible networks.

First, we present a general method, called *layered network*, for the problem of configuring application sessions that require intermediate processing. The *layered network* method finds optimal configurations by transforming the session configuration problem into a conventional shortest path problem. We show, through a series of examples, that the method can be applied to a wide variety of situations. We also discuss how to configure applications that require reserved capacity and propose effective heuristic algorithms that are based on the layered network method.

Second, for designing extensible networks, we generalize the constraint-based network design methods originally developed for conventional networks. We show how to incorporate arbitrary application requirements that are allowed by extensible networks in a flexible and general way. We also show how to extend the original framework to dimension both processing resources and link bandwidth. These results have been incorporated into a software package, the Extensible Network Planner (XNP).

This thesis is dedicated to my family.

Contents

List of Tables . . . . **vii**

List of Figures . . . . **viii**

Acknowledgments . . . . **xii**

1 Introduction **1**

1.1 The extensible network environment 2

1.2 Configuring resources in extensible networks 5

1.3 Designing extensible networks 8

1.4 Dissertation overview 10

2 Resource Configuration **12**

2.1 Specifying the Session Format and Network 13

2.2 Configuring Generic Sessions 16

2.3 Configuring unicast sessions 19

2.3.1 Layered networks for single step processing 21

2.3.2 Using layered networks for multiple processing steps 23

2.4 Applications with optional processing 25

2.5 Congestion Control Processing 27

2.6 Configuring single source multicast sessions 29

2.7 Related Work 37

3 Resource Configuration in Capacity Constrained Networks **40**

3.1 Generalized Resource Configuration Problem Redefined 41

3.2 Heuristics: Selective resource consideration 45

3.3 Heuristics: capacity tracking 47

3.4 Simulation Results 51

4 Designing Extensible Networks 60
4.1 Constraint-based Network Design 61
4.2 Introduction to Designing Extensible Networks 64
4.3 Designing Extensible Networks 70
4.3.1 Application Format 71
4.3.2 Traffic Constraints 71
4.3.3 Traffic Configuration 72
4.3.4 Routing Policy 72
4.3.5 Load Factor 73
4.3.6 The Resource Dimensioning Problem 74
4.3.7 Least-cost Routing Policy 75
4.4 Resource Dimensioning using Flow Graphs 77
4.5 Computing Lower Bounds on Network Cost 84
4.6 Computing Lower Bounds using Flow Graphs 86
4.7 Discussion 89

5 Extensible Network Planner 91
5.1 Obtaining and Starting XNP 92
5.1.1 Installing XNP 92
5.1.2 Running XNP 92
5.2 Basics of Extensible Network Planning 93
5.2.1 Constructing a network 93
5.2.2 Application formats 98
5.2.3 Describing traffic expectations 99
5.2.4 Resource dimensioning 102
5.2.5 Evaluating network configurations 104
5.2.6 Design Space 105
5.3 Design Operations 106
5.4 File Operations 107
5.5 Algorithms implemented in XNP 108
5.5.1 Generating and modifying network topologies 108
5.5.2 Placing network nodes 110
5.5.3 Generating traffic constraints 111

6 Extensible Network Design Using XNP **112**
6.1 Experimenting with Various Design Choices 112
6.2 Enhancing the Trial Designs 126

7 Summary and Future Work **137**
7.1 Session Configuration in Extensible Networks 137
7.2 Constraint-based Design of Extensible Network 139

References **142**

Vita **148**

List of Tables

4.1 Resulting Capacities after Dimensioning 66
4.2 Resource Capacities for Multiple Applications 69
4.3 Traffic constraints associated with Figure 4.5 for resource dimensioning 80
4.4 Load factor of the pairs whose path is routed through (Phili, NY), relative to the initially configured session bandwidth 80
4.5 Traffic constraints associated with Figure 4.5 for computing a lower bound 87

6.1 Sources and their receiver locations for the US network 113
6.2 Sources and their receiver locations for the Western Europe network . 121
6.3 Links added and removed for the US network 130
6.4 Links added and removed for the Western Europe Network 135

List of Figures

1.1 Current Network Uses 2
1.2 Example of communication between two corporate networks 3
1.3 Video transfer application with decompression processing 7
1.4 Example of network design 8
1.5 Example of network design with processing 10

2.1 Session graphs 13
2.2 Processing types with location constraints 14
2.3 Network graph 15
2.4 Session configurations 16
2.5 Unicast with single step processing 20
2.6 Session graph with a single processing step 22
2.7 Session configuration using a layered network 22
2.8 Session configuration using a layered network for multiple processing steps 24
2.9 Layered networks for optional compression/decompression 26
2.10 Transformed Network for Optional Processing 28
2.11 Multicast session graph for a video transfer application 29
2.12 Multicast session configurations 30
2.13 Layered network for multicast 31
2.14 Comparison of the tree costs resulting from the heuristic methods, where the result of the shortest path tree method costs less than the multi-step tree augmentation 32
2.15 Comparison of the tree costs resulting from the heuristic methods, where the result of the multi-step tree augmentation costs less than the shortest path tree method 33
2.16 Metro 50 Network 34

2.17 Comparison of costs for resources in multicast trees 35
2.18 Comparison of diameters of multicast trees 36

3.1 Video transfer application with bandwidth requirements 41
3.2 Network graph with a cost and capacity specified for each resource . . 43
3.3 Session configuration in capacity constrained networks 44
3.4 Shortest path tree in a layered graph 49
3.5 Blocked path in capacity tracking 50
3.6 Torus network 51
3.7 Metro 20 Network 52
3.8 Metro 50 Network 53
3.9 Performance of heuristics for session configuration 55
3.10 Performance of heuristics for session configuration, the metropolitan area networks 55
3.11 Configuration Cost 57
3.12 Configuration Costs of the metroarea networks 57
3.13 Blocking rates at 75% traffic load 58
3.14 Configuration cost at 75% traffic load 58
3.15 Time requirements for session configurations 59

4.1 XNP snapshot for Acme corporation network 65
4.2 XNP snapshot with a lower cost design 67
4.3 XNP snapshot for video-on-demand application 68
4.4 Layered network with least-cost path 76
4.5 Least-cost route from Det to NY 77
4.6 Flow graph used for dimensioning (Phili,NY) 82
4.7 Flow graph used for computing a lower bound 88

5.1 XNP main menu and buttons 94
5.2 Example popup windows for adding a node 94
5.3 XNP snap shots while generating a topology 95
5.4 Popup window for entering a link cost 96
5.5 XNP snap shots for placing processing capability 97
5.6 Buttons for editing network configurations 97
5.7 Buttons for applications and traffic constraints 98
5.8 Popup window for adding an application format 98

5.9 Popup window for describing an application format 99
5.10 Popup window for removing application formats 99
5.11 Popup window for selecting a traffic constraint type 100
5.12 Popup window for adding traffic constraint between two sets of nodes 101
5.13 Popup window for adding traffic constraint when the destinations include all nodes 101
5.14 Popup window for adding traffic constraint when the sources include all nodes 102
5.15 Popup window for removing traffic constraints 102
5.16 Buttons for evaluating network configurations 102
5.17 XNP snap shots for dimensioning information 103
5.18 XNP snap shots for lower bound information 104
5.19 Two other views in XNP 105
5.20 Buttons for managing designs 106
5.21 Buttons for file operations 107

6.1 Traffic constraints for maximum incoming/outgoing traffic at Minneapolis, proportional to the population 114
6.2 Traffic constraints for the video on-demand application 114
6.3 Networks for the 20 largest metropolitan areas for US 116
6.4 Networks for the 20 largest metropolitan areas for US 117
6.5 Networks for the 20 largest metropolitan areas for US 118
6.6 Cost comparison for US topologies 120
6.7 Networks for the 20 largest metropolitan areas in Western Europe . . 122
6.8 Networks for the 20 largest metropolitan areas in Western Europe . . 123
6.9 Networks for the 20 largest metropolitan areas in Western Europe . . 124
6.10 Cost comparison for Western European topologies 125
6.11 Networks for the 20 largest metropolitan areas in US, processing at all possible locations 127
6.12 Networks for the 20 largest metropolitan areas 128
6.13 Networks for the 20 largest metropolitan areas 129
6.14 Modified hand-crafted topology, Placement C 130
6.15 Comparison of different placements on hand-crafted and modified hand-crafted topologies for the US network 131
6.16 hand-crafted topology, processing at all possible locations 132

6.17 Networks for the 20 largest metropolitan areas in Western Europe . . 133
6.18 Networks for the 20 largest metropolitan areas in Western Europe . . 134
6.19 Comparison of different placements on hand-crafted and modified hand-crafted topologies for the Western Europe network 135

Acknowledgments

This dissertation would not have been possible without the help and the support of many people. My foremost thanks go to my advisor, Dr. Jon Turner, who has been incredibly patient with me for the past five years. He guided and encouraged me throughout the study with his broad perspectives and continuous dedication for research and teaching. I also thank my committee members, Dr. Dan Fuhrmann, Dr. Chris Gill, Dr. Sally Goldman, and Dr. Ellen Zegura, for their valuable advice and suggestions.

I thank all the professors and staff members in ARL and Computer Science and Engineering, who have been always there for me with constant supports and inspiration. I also thank my fellow students for their wit and humor, which cheered me up immensely at my dissertation defense talk.

My special thanks go to Prof. Jim Ballard for guiding me patiently throughout the dissertation writing process.

Finally and from the bottom of my heart, I thank Samphel, and the rest of my family and future family for their unconditional love and support. Thank you.

Sumi Y. Choi

Washington University in Saint Louis
December 2003

Chapter 1

Introduction

Conventionally, the goal of packet-switched networks has been delivering data to users. To do this, each network node (switch or router) maintains information about where to forward data packets, makes a routing decision for each packet, and finally forwards the packet according to the decision. In this paradigm, every node is restricted to operating fixed protocols that have been agreed upon in advance. The collaboration of all nodes in the network eventually ensures that all packets will arrive at their intended destination. This restriction enables global Internet connectivity, which improves many areas of human life, such as scientific research, medical collaboration, personal communication, and education. In the business realm, commercial solutions for marketing and sales are making Internet-based applications ever more appealing. In all of these areas, users expect networks to be a pervasive and flexible environment in which they can use devices that vary in capability. Figure 1.1 shows these current uses of networks.

However, the restriction of fixed processing limits the functionality of networks considerably, making current network frameworks unsuitable for future network applications. In this dissertation, we consider more "extensible" network environments, where new protocols and packet processing can be deployed at network nodes dynamically. In such extensible networks, software developers can install packet processing customized for their applications at network nodes, so that packets belonging to those

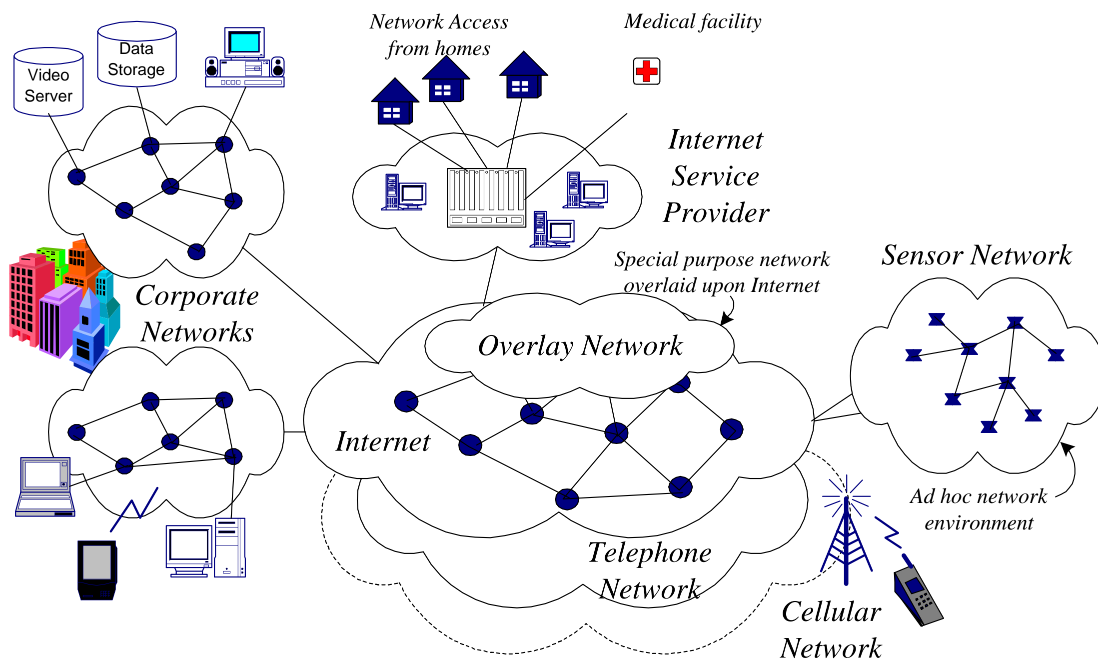


Figure 1.1: Current Network Uses

applications can be processed as they are forwarded. Now that there is significant use of the Internet for many different purposes, extensible networks can play a crucial role in providing advanced processing appropriate for varying application criteria.

There has been a substantial volume of research on realizing and efficiently operating extensible networks. Many different approaches have been proposed for the network framework and the node infrastructure. In this dissertation, we study techniques for managing resources to ensure consistent performance of applications. These techniques can be applied to most realistic approaches to extensible networks.

1.1 The extensible network environment

In order to better understand the resource management issues in extensible networks, we begin by examining the core concepts and the currently available frameworks for extensible networks in more detail. Two features are essential for extensible networks. First, the network must provide programmable components on network nodes that

will facilitate the dynamic installation or the update of customized processing. Second, the network must maintain information about the resources required for such applications, and configure appropriate resources to individual application sessions as needed.

Let us illustrate how applications might operate in such networks. Consider an application which involves an enterprise network with multiple sites that communicate over the Internet. As a matter of corporate policy, it is required that all communication between corporate sites be encrypted. This policy can be implemented by installing encryption modules and decryption modules in programmable components of the network nodes. Then, during the communication session, all packets are encrypted at one of the network nodes before leaving the remote corporate network and decrypted at another of the network nodes after they enter a corporate network. To conserve the use of network bandwidth, the encryption and decryption may be combined with compression and decompression modules if the data is in a video format. This example is shown in Figure 1.2, where each node with a programmable component is highlighted and the processing module installed on it is specified. While a node

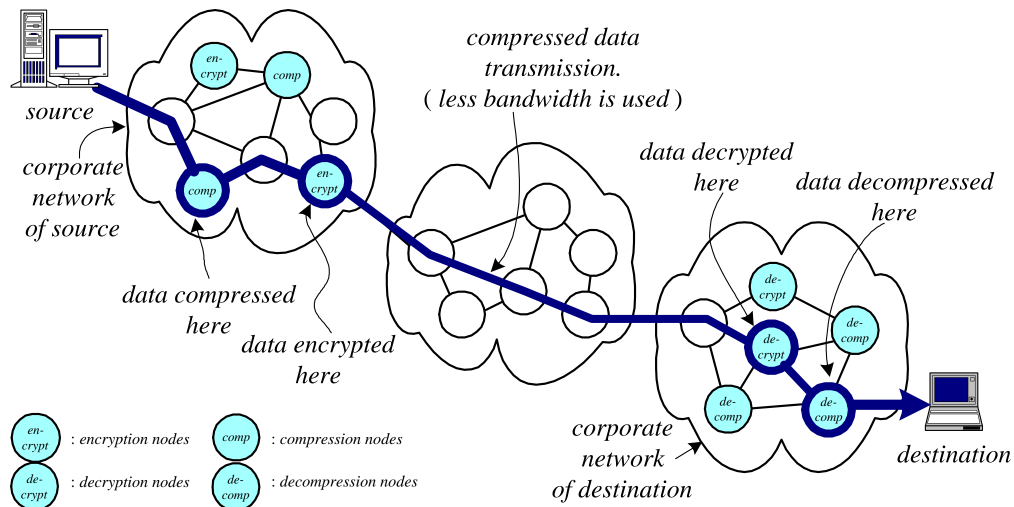


Figure 1.2: Example of communication between two corporate networks

can contain multiple processing modules, we are showing a simplified case where only one processing module is installed on each of the processing-enabled nodes. Once all

the necessary processing modules are installed at network nodes, to set up an application session, we must configure appropriate resources. In our example, processing resources are required to handle the compression, the encryption, the decryption, and the decompression processing, while also satisfying the location restriction that maintains the secureness of the communication. Also, adequate bandwidth is required on the route that connects the source and the destination via the selected processing nodes in the given order. The links and the processing nodes used by the communication are shown using thick lines in Figure 1.2. Other application examples may include congestion control, format conversion, and data caching, along with others yet to be recognized.

There are several possible approaches to realizing the potential of this extensible network technology. Active networking [1, 2, 3, 4] is one such approach. In the best-known variant of active networking, the capsule model [5, 6, 7], packets are interpreted by routers as programs to be executed, rather than just as data to be forwarded. Such execution, however, poses serious security concerns. More realistic variants of active networking involve the dynamic execution of trusted programs on behalf of individual application sessions [8, 9], in response to signaling messages exchanged at the start of a session [10]. Other forms of extensible networks include overlay networks. Overlay networks have been exploited to introduce new services to an existing network. Overlay networks comprise a subset of servers that communicate through the underlying network, and can be viewed as extensible networks where the servers act as extensible nodes, or programmable routers, capable of handling new services. Because the servers will be shared only by applications supported by the overlay network, we can effectively program and deploy new services customized for the applications. This dissertation is oriented toward a view of extensible networks where customized processing or services can be installed off-line into programmable network elements, and can be used by individual application sessions, as in realistic

variants of active networks or overlay networks. Here, the use of programmable network elements can be controlled either by administratively-determined policies or by user-initiated signaling messages.

Now that we have briefly discussed our view of extensible networks, let us look at the main resources to be considered and how to manage them in extensible networks. As mentioned, conventional applications are passive in their use of network resources: They conform to fixed protocols in the underlying networks, and only the link bandwidth is considered a shared resource. However, the current trend in applications, particularly in extensible networks, shows more comprehensive use of network resources, requiring some network nodes to provide program modules and associated processing resources. While link bandwidth media is still an essential resource in extensible networks, new resources are necessary to perform customized processing or advanced services on network nodes. There are two broad tasks in supporting the applications requiring customized processing from networks.

- Configure each application session with resources sufficient to guarantee consistent performance and efficient usage
- Provision the network with sufficient network resources to satisfy the application requirements and also accommodate the maximum anticipated traffic

These tasks, configuring application sessions and provisioning the network, are studied extensively in this dissertation. The next two sections introduce them in more detail.

1.2 Configuring resources in extensible networks

Let us first consider the task of configuring resources for application sessions in extensible networks. Here, application sessions are instances of network applications that are initiated for individual users. Because of fixed protocol layers, every node in conventional networks treats all packets in a homogeneous way. Each simply decides where to route each packet, mainly based on its destination, and forwards it

according to the decision. Therefore, there is no need to distinguish individual application sessions in conventional networks. In extensible networks, however, each application session may need to be processed differently according to its processing requirement. It may even need to be routed through a different path in order to ensure that the packets belonging to the session are routed through processing-capable nodes. To meet these needs, two requirements must be satisfied. First, a route must be determined for each application session that satisfies the processing requirement (and possibly the bandwidth requirement) of the session. Second, to guarantee their availability, resources must be secured for the duration of the session.

The issue of determining routes is not new. Even in conventional networks, the routing decision must still be made for each packet, although no distinction is necessary for different applications, and one routing method is sufficient for all applications. There has been a good deal of research on computing optimal routes. Most algorithms employ either the distance vector approach or the link state approach [11]. The implemented protocols include the RIP protocol[12], OSPF[13], and PNNI[14]. The common goal of the routing algorithms is to find the optimal paths for potential pairs of nodes that may need to communicate, given a metric for links in the network. Each of these algorithms is based on a shortest path algorithm. In the routing algorithms, the metric used to measure the quality of a route can be as simple as the number of hops [12]. To achieve more specific goals, other metrics have also been applied to routes, such as delay or available bandwidth [15, 16].

These routing algorithms are not, however, directly applicable to configuring application sessions in extensible networks. For a given application session, it may not be sufficient to find a shortest path to satisfy the processing requirement. The shortest path computed in the conventional way may not include any processing-capable node. A valid route, however, must include at least one set of nodes that is capable of the processing. An optimal route must be one of those valid routes, where, in addition to the impact of using link resources, the impact of using processing resources is also considered. Furthermore, in a situation where the processing changes

the link bandwidth, the route selection may also need to reflect the effect of the varying bandwidth. Figure 1.3 shows an example of such sessions, where compressed video data is sent from a source to a receiver which is incapable of performing the associated decompression processing. Here, the initial video stream, compressed by 10:1, is sent from the source and is decompressed at a processing node en route to the receiver. Because decompression processing expands data bandwidth, it is preferred to perform this processing close to the destination. The chosen route and processing node are shown by thick lines in Figure 1.3. In general, routing methods in extensible

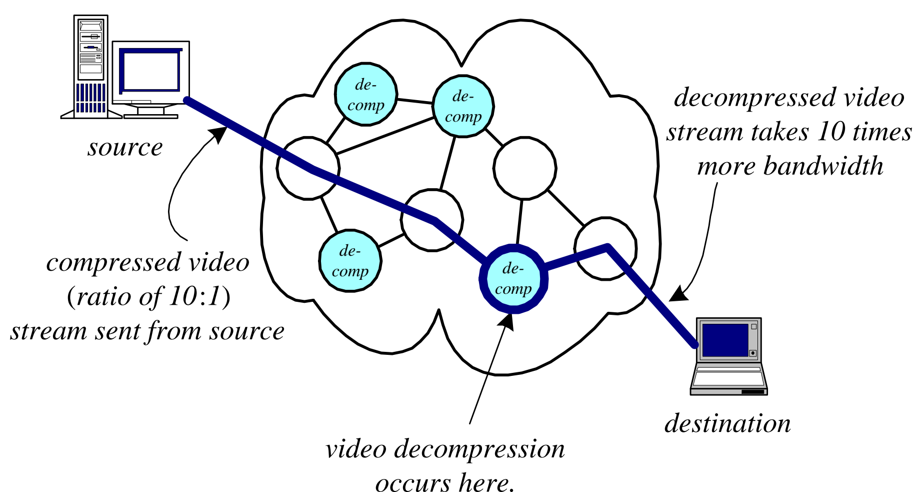


Figure 1.3: Video transfer application with decompression processing

networks must take into account the impact of using link resources, the impact of using processing resources, the bandwidth variation due to processing, and other application characteristics that affect the resource selection. Chapters 2 and 3 study methods that can be applied to diverse applications and situations.

As mentioned, the second requirement in configuring application sessions is to guarantee resource availability. Let us consider a situation where the resources required to configure an application session have been identified by the routing method. In order to guarantee consistent performance for the session, proper capacity must be maintained at each of the resources for the duration of the session. Signalling is required to poll each resource for availability and to secure the required capacity.

We will briefly discuss possible ways to implement the signalling and the resource reservation for extensible networks.

1.3 Designing extensible networks

In Chapter 4 of this dissertation, we consider a different perspective on resource management in extensible networks, the network design. A well designed network is crucial to delivering performance guarantees for applications. By carefully determining where to place resources and how much capacity to assign to each resource, one can plan a network that will effectively serve the expected traffic while minimizing the cost of deployment. Depending on the purpose of the network, the characteristics of applications, and the traffic patterns of users, the optimal design may vary. The objective is to design and configure extensible networks that are guaranteed to have sufficient network bandwidth and processing resources to meet expected traffic demands. Figure 1.4 shows a network composed of five locations in the northeastern

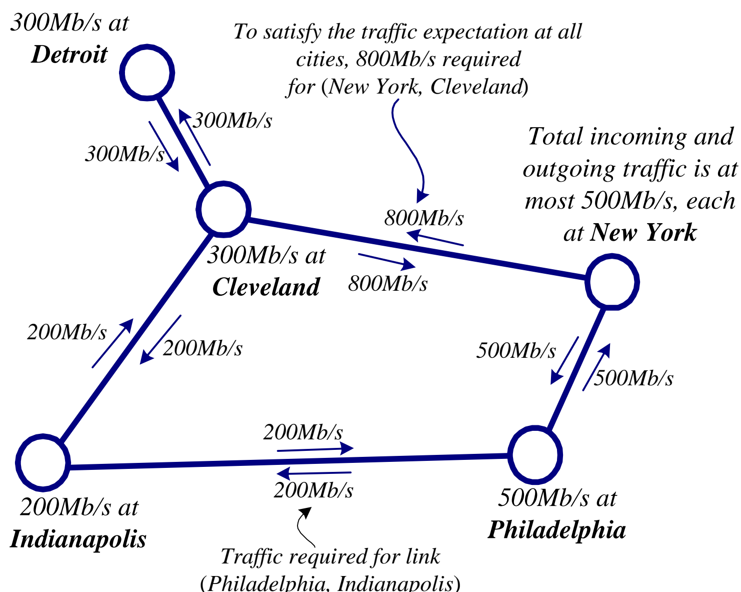


Figure 1.4: Example of network design

United States. In this network, the outgoing and incoming traffic is at most 500 Mb/s at New York and Philadelphia, 300 Mb/s at Detroit and Cleveland, and 200 Mb/s at

Indianapolis. To accommodate any traffic that obeys these constraints, the network must have bandwidth for each link as specified in the figure, assuming that traffic is routed through shortest paths. Our approach extends this constraint-based network design methodology [17] to extensible networks.

Constraint-based design of conventional networks starts with a set of network locations, or sites, and a set of traffic constraints. In the simplest case, the traffic constraints simply place upper bounds on the amount of traffic that can originate or terminate at each site [18, 19, 20, 21], as exemplified in Figure 1.4. In addition to simple ingress/egress constraints, network designers may specify constraints that bound the amount of long distance traffic or limit the traffic between pairs of sites. For example, we may add a pairwise traffic constraint to Figure 1.4 that restricts the traffic from New York to Indianapolis to be less than or equal to 100 Mb/s. This reduces the bandwidth needed from New York to Cleveland to 700 Mb/s in each direction. The general form of a traffic constraint is a bound on the traffic between any two subsets of sites. Given a set of constraints, the design problem is to find a least-cost network configuration in which the link capacities ensure that any traffic configuration allowed by the constraints can be handled.

We show how this original framework can be generalized to handle the configuration of processing resources and link bandwidth in extensible networks. To briefly illustrate the design of extensible networks, let us take an example. In Figure 1.5, we show an extensible network with the same topology as in Figure 1.4 and with two processing nodes (highlighted). In this extensible network, we consider a video transmission application requiring decompression processing in the network as introduced in Figure 1.3. During the sessions configured for this application, the compressed video stream is sent from a source and decompressed on the route to a receiver. Due to the compression ratio, which is assumed to be 10:1 in this case, data bandwidth changes during the transmission. The link capacity at each link must then be determined by consideration of this application characteristic. The processing resources

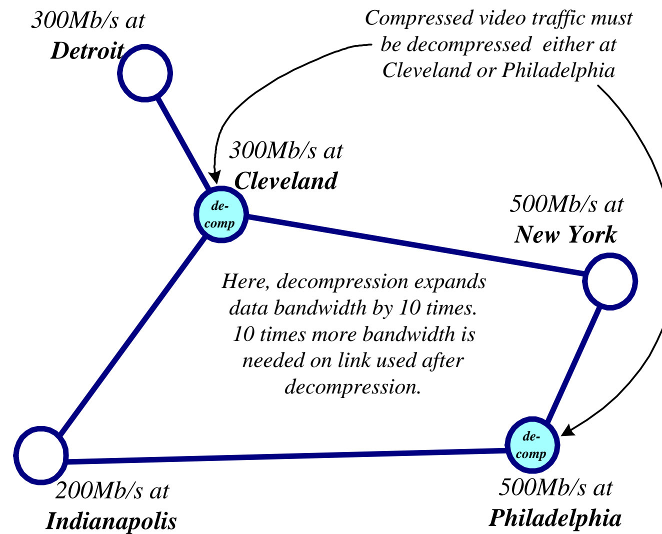


Figure 1.5: Example of network design with processing

and their capacity must also comply with the decompression specification and traffic expectation.

Because the resources needed by a network are highly dependent on the characteristics of the applications, we introduce a general method for describing the resource requirements of application classes and show how these can be used in the network design process. We have incorporated the methods developed for extensible network design into a software package called the Extensible Network Planner (XNP), which is based on an earlier, unpublished tool for designing conventional networks [22]. To demonstrate the utility of these methods, we include examples of how XNP can be used to design real networks.

1.4 Dissertation overview

Chapter 2 describes resource configuration methods for applications that require intermediate processing. We present an efficient solution called the *layered network*, and show its applicability and scalability by describing various situations where the method can be applied. Chapter 3 extends the resource configuration problem to networks with explicit limits on resource capacity. We illustrate the intractability of the

problem for general cases and describe heuristic methods which perform effectively in most practical situations.

Chapter 4 investigates the problem of designing extensible networks to satisfy given applications and traffic patterns. After exemplifying the problem in simple, and yet realistic, situations, we describe a systematic and pragmatic approach to designing extensible networks, and illustrate the algorithmic methods used in the design process. Chapter 5 introduces the *Extensible Network Planning* tool, which helps network designers plan extensible networks through an interactive process. The methods described in Chapter 4 are implemented in XNP. In Chapter 6, we demonstrate the process of network design for several medium size networks. Chapter 7 summarizes the dissertation.

Chapter 2

Resource Configuration

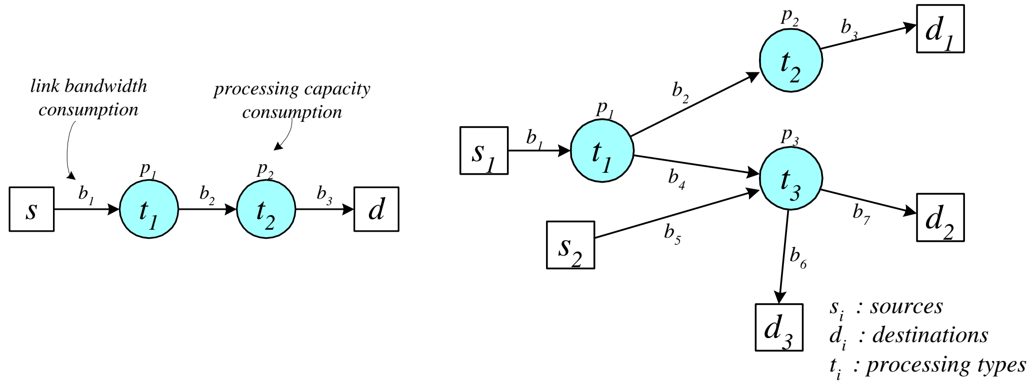
This chapter details the problem of configuring resources for individual application sessions, given candidate resources in an extensible network. Application sessions are instances of network applications which initiate and maintain connections for specific users. The goal of the resource configuration problem is to map each application session onto the best set of resources, according to the criteria relevant to the session. For example, consider an application that requires specific encryption and decryption of its session data transmitted between two remote corporate networks. The processing component on routers can be programmed to perform the encryption and the decryption. The session must then be mapped to links for the data flow and to the routers where the encryption and decryption are performed on the data. The requirements, which include processing cycles, memory, or the ability to execute a particular program, are used to identify the set of routers capable of doing the processing.

Consider another example, a video transmission application, whose goal is to transform the video data at an intermediate router so that an incompatible receiving device can view it. The configuration of this session should identify the set of links used to form the data path and the router which will do the video transformation. It may also be preferred that the bandwidth and processing resource usage on the path be minimized.

2.1 Specifying the Session Format and Network

In general, the task of configuring a session is composed of identifying the session format, particularly the communication and the processing that occur in the session, and then mapping the format onto the resources in the network that can fulfill the goals of the session.

Depending on the purposes and the goals, network applications can take different forms. In this work, the session formats are used to describe the terminals that are involved in individual application sessions, the data that flows among the terminals, and the processing applied to the data flows. We adopt a graph model to express each session format, where the nodes stand for the terminals or the processing steps, and the edges stand for the data flows. Each edge in the graph is associated with the link bandwidth consumed by the corresponding data flow. Similarly, each non-terminal node is associated with the processing capacity consumed by the corresponding processing step. Figure 2.1(a) shows an example that describes a unicast



(a) Unicast session graph

(b) Complex session graph

Figure 2.1: Session graphs

session with two steps of intermediate processing. In the figure, the terminal nodes are s and d, and the processing steps are $t1$ and $t2$, while the edges specify the data flows from one terminal to the other terminal. The bandwidths consumed by the data flows are b_1, b_2, and b_3, and the processing capacities are p_1 and p_2. We refer to

the graph describing a session format as the session graph. Figure 2.1(b) is another example of a session graph that involves more terminals, processing, and data flows. In general, a session graph forms a directed graph that involves one or more terminal nodes. While arbitrary topologies are possible, we focus our attention here on paths and trees.

We also describe the network as a graph. While the session graph describes only the properties of a session, the network graph describes the entire network, which is composed of terminals, routers, and physical links that connect them. For extensible networks, some routers are specially labeled as processing nodes representing extensible routers. We may further categorize the processing nodes so that each of them is labeled with the types of processing that it can handle. The categorization is

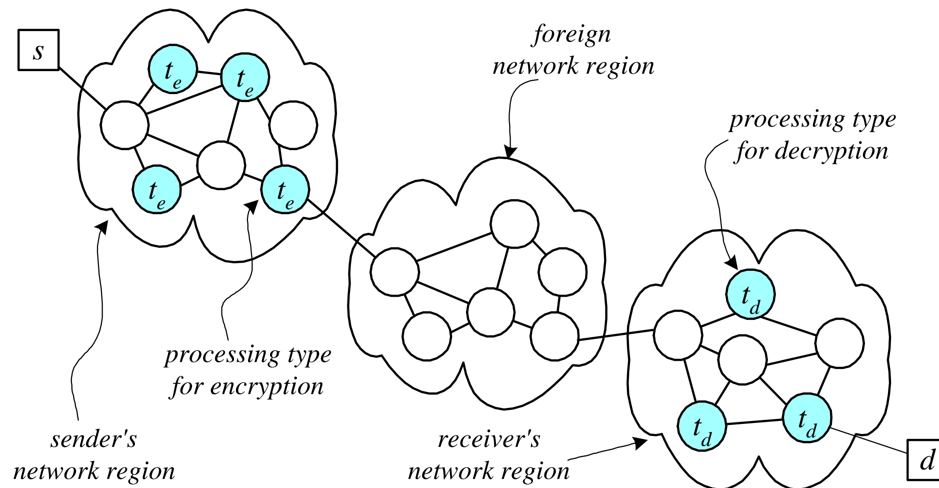


Figure 2.2: Processing types with location constraints

partially determined by the system specifications for the routers, in which properties such as processing power, memory size, or software capabilities may be considered. Processing can also be categorized by application constraints. For instance, recall the example of the encryption and decryption application. In this case, the application intends to perform the encryption before the data leaves the network region that includes the sender and to perform the decryption after the data enters the network region that includes the receiver. Figure 2.2 shows the two regions. To reflect this

location constraint, we label all processing nodes in the sender's network region to show the nodes where the encryption can be done. Similarly, we label all processing nodes in the receiver's network region for decryption. The processing nodes of the two types are highlighted in Figure 2.2.

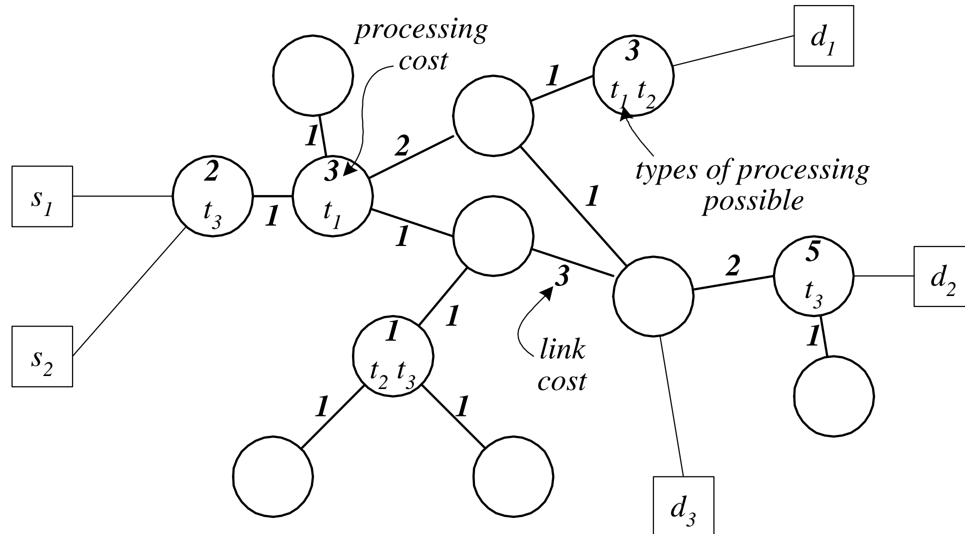
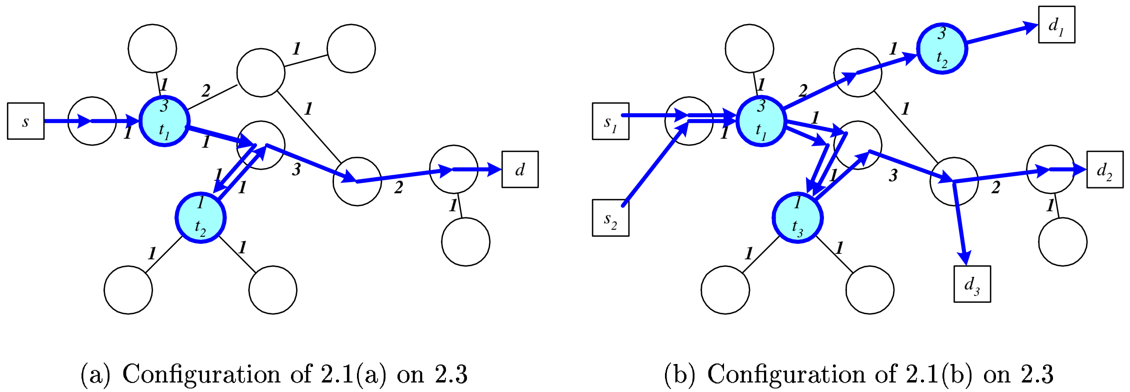


Figure 2.3: Network graph

Figure 2.3 shows an example of a network graph, where terminals are drawn as squares and router nodes are drawn as circles. Each processing node is also labeled with the processing types (t_1, t_2, t_3) that it can handle.

Now that we have described the sessions and the network as graphs, the task of configuring a session is a matter of mapping the session graph onto the network graph. Here, the session terminals are mapped to the corresponding network terminals, the processing steps are mapped to processing nodes of the appropriate types, and finally the connections between two nodes that are adjacent in the session graph are mapped to paths that connect the corresponding nodes in the network graph. Following this principle, we can consider configuring the session graphs in Figure 2.1 onto the network graph in Figure 2.3. Figure 2.4 shows the configuration using thick arrows for the paths that connect the sources to the destinations and shaded nodes for processing locations.



(a) Configuration of 2.1(a) on 2.3

(b) Configuration of 2.1(b) on 2.3

Figure 2.4: Session configurations

Meanwhile, in networks where resources such as processing nodes and links are costly and limited commodities that are shared by multiple parties, sessions incur expenses when they consume resources that are configured and assigned to them. In this work, we refer to the expense of consuming a unit capacity of a resource as its unit cost. While the actual measure for the cost may vary depending on the network models, we limit it to a positive value as illustrated by the numbers labeling processing nodes and links in Figure 2.3.

In the following section, we formally describe the problem of configuring sessions for the optimal cost, and we present efficient solutions for the most important specific case of the problem (sessions which define paths). Later, we also discuss heuristics targeting other cases, as well as issues related to resource capacities.

2.2 Configuring Generic Sessions

In this section, we define the configuration cost of sessions and formally state the optimal session configuration problem. We have already introduced the cost associated with each resource as it is configured for a session. Given a mapping that identifies the resources to be configured for a session, we can compute the cost at each resource as the product of its unit cost and the capacity of the resource consumed by the session. The configuration cost of the session is then defined as the sum of the costs

of all resources designated by the mapping. We now attempt to find the mapping that results in the least cost configuration, which is the goal of the optimal session configuration problem.

In PROBLEM 2.2.1, we formally state the optimal session configuration problem, given a network graph $G = (V, E)$ and a session graph $G_s = (V_s, E_s)$.

PROBLEM 2.2.1 *Session Configuration Problem*

Given: *A directed session graph $G_s = (V_s, E_s)$, with a type $t(u)$ and a capacity requirement $p(u)$ for each vertex $u \in V_s$, and a bandwidth requirement $b(u, v)$ for each edge $(u, v) \in E_s$. Also, a directed network graph $G = (V, E)$ with a type set $T(u)$ for each node $u \in V$. In addition, the unit costs, $c(u)$ and $c(u, v)$, are given for each vertex and edge as a positive value.*

Find: *A location function $l : V_s \rightarrow V$ and a routing function $r : E_s \rightarrow 2^E$ that satisfy*

$$\forall u \in V_s, \qquad t(u) \in T(l(u)) \tag{2.1}$$

$$\forall (u, v) \in E_s, \qquad r(u, v) \text{ is a simple path in } G \text{ from } l(u) \text{ to } l(v) \tag{2.2}$$

and that minimize the cost $C_{l,r}$ where

$$C_{l,r} = \textstyle\sum_{u \in V_s} c(l(u))p(u) + \sum_{(u,v) \in E_s} c(r(u, v))b(u, v). \tag{2.3}$$

This problem statement assumes that network resources have unlimited capacities and focuses on optimizing the configuration cost. In Chapter 3, we will restate the problem to consider resources with strictly limited capacities, and discuss the session configuration problem in such cases.

The mapping that designates the resources for a session is defined by the location function l and the routing function r, where the function l maps each node in G_s to a node in G, and the function r maps each edge in G_s to a path in G. The functions should also satisfy conditions (2.1) and (2.2), where Condition (2.1)

ensures that selected nodes can perform the required processing and Condition (2.2) ensures that adjacent nodes in the session graph are connected by a path in the network graph. Condition (2.2) also constrains the path to be simple in order to avoid unnecessary link usage. Here, a path is simple if it repeats no node. Because link costs are positive values, by removing loops in the path, a non-simple path can always be reduced to a simple path that connects the same end points and that has a lower cost. Problem 2.2.1 only considers these simple paths.

Finally, we define the configuration cost $C_{l,r}$ to be the total sum of the costs of all resources used by the session. The optimal mapping must minimize the configuration cost $C_{l,r}$.

Among graph problems, the graph embedding problem [23] is most closely related to the session configuration problem. In the graph embedding problem, an embedding of a graph G onto another graph G' is defined as a mapping from the former to the latter, where each distinct node in G is mapped to a distinct node in G', and each edge in G to an edge in G'. The graph embedding problem has been used for designing parallel algorithms on interconnection networks, which are composed of processors in distributed memory machines. Here, the algorithms are represented by graphs where each node stands for a processing step, and each edge stands for the sequence and/or the data flow between steps. The embeddings are then used to implement and operate the algorithms in the network.

In more realistic situations, the edge mapping in the graph embedding problem is often relaxed so that an edge in G is allowed to be mapped to a path in G'. This formulation is called weak graph embedding and closely resembles our session configuration problem. In fact, the weak graph embedding problem is a special case of the session configuration problem, where the session graph and the network graph have a single processing type. In addition, the optimal weak embedding problem is defined as the problem of finding the least-cost embedding, given values assigned to the edges and the nodes as for the optimal session configuration problem.

There has been research effort for the weak embedding problem particularly where the underlying graph G' onto which another graph G is to be mapped bears certain regular properties as in grid networks [24, 25]. However, the general weak embedding problem is known to be $\mathcal{NP}$-hard [23] and thus so is the session configuration problem, because it contains the weak embedding problem. For those general cases, little is known about approximation issues.

Nevertheless, most application sessions belong to categories of session patterns, in which the optimal session configuration can be solved or closely approximated efficiently. We will discuss these cases next.

2.3 Configuring unicast sessions

Unicast is the most common session form. A unicast session takes the form of a path with a single source and destination plus zero or more intermediate processing steps. The session graph of a unicast session is shown in Figure 2.1(a) with two steps of intermediate processing.

When no processing is involved, the optimal configuration is identical to the least cost path between the terminals. For this particular case of unicast sessions, standard shortest path algorithms can be used to find the best session configuration.

When intermediate processing is required, the configuration must also select processing nodes. As described in Problem 2.2.1, the function l maps the processing steps in the session graph to nodes with the corresponding processing types. Here, the configuration still forms a path from the source to the destination, and the processing nodes designated by the function l are included as intermediate nodes. Figure 2.4(a) shows a configuration for the unicast session graph given in Figure 2.1(a), where the configured path from source s to destination d passes through two designated processing nodes with matching types.

The optimal configuration in this case should also have the least configuration cost, which now includes the cost of each of the configured processing nodes, in

addition to the path cost. Because of the node selection, standard shortest path algorithms are not directly applicable to the problem. Nevertheless, the shortest path information is still crucial to solving the problem. Next, we illustrate a method that computes the least cost configuration for unicast sessions and discuss related issues. Initially, we focus on unicast sessions with single step processing, and then we expand the discussion to multiple steps.

Let us assume a unicast session in the network graph $G = (V, E)$, with the source s, the destination d, and one processing step of type t_1 to be done on the data flow from s to d, where R is the set of nodes which can handle the processing. The session's graph is shown in Figure 2.5 with its bandwidth requirements (b) and processing capacity requirement (p). The following describes how the method computes the least cost configuration for this session.

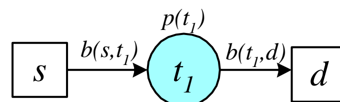


Figure 2.5: Unicast with single step processing

First, for each processing node r in R, compute the shortest paths from s to r and from r to d, and construct the configuration using the shortest paths as the data paths and the node r as the processing node. Also, compute the cost of this configuration by summing the cost of the shortest paths and the cost of the processing node r. Here, the cost of each link in the path from s to r is scaled up by the bandwidth consumption $b(s, t_1)$ from its unit cost, and similarly each link in the path from r to d is scaled up by $b(t_1, d)$. The cost of r is scaled up by $p(t_1)$. Note that this configuration gives the least cost, given r as the processing node. Then, among all nodes in R, select the node that results in the smallest configuration cost when used as the processing node, and choose the associated configuration as constructed above.

In this method, the shortest paths can be found by computing the shortest path tree from the source s and another shortest path tree converging to the destination

d. Using Dijkstra's algorithm, the time complexity of the method is $O(|V|\log(|V|) + |E| + |R|)$. While this implies that the described method is viable in the specific case which involves only one processing step, there are issues with generalizing the method to handle more processing steps. Consider another unicast session, which now requires two processing steps. Let us denote the sets of nodes for the processing steps as R_1 and R_2. By applying the same method, we need to account for each pair of processing nodes in $R_1 \times R_2$ to configure the session. This accounting requires $|R_1| + 2$ shortest path trees to be computed to obtain all the shortest paths required for the configurations. In addition, $|R_1| \times |R_2|$ comparisons are needed to find the least cost configuration. If there are k steps, we need to compute and compare the cost associated with each choice for the processing nodes, where the number of choices is $O(|R_1| \times |R_2| \times ... \times |R_k|)$. In the worse case, the comparison takes $\Omega(n^k)$ time.

2.3.1 Layered networks for single step processing

In this section, we introduce a better alternative for solving the unicast session configuration problem. This method takes the problem given in the network graph into a different space, where the problem is solved as a conventional shortest path problem. Then, the result is brought back into the original network graph to obtain the final solution.

We illustrate the method by focusing on the transformation that converts the network graph into a new graph space called a "layered network". Let us reconsider the unicast session with a single processing step, where R is the set of processing nodes that are capable of the processing, given the network graph $G = (V, E)$. For this session, the new method, which we will call the *layered network* method, transforms the original network graph into a "two layer" graph.

The layered network G' includes two copies of the network graph G. We refer to one copy as layer 0 and the other copy as layer 1. Also, for each node v in G, we denote the copy of the node in layer 0 as v_0 and the copy in layer 1 as v_1. The cost

of each edge in layer 0, say (u_0, v_0), is set to be the product of the unit cost of the original edge (u, v) in the network graph and the bandwidth requirement $b(s, t_1)$, i.e., $c(u, v) \times b(s, t_1)$. Similarly, the cost of each edge (u_1, v_1) is set to be $c(u, v) \times b(t_1, d)$.

Then, we complete the layered network G' by adding an inter-layer edge (r_0, r_1) for each processing node r in R. Here, the cost of (r_0, r_1) is set to the product of the unit cost of the node r and the processing requirement $p(t_1)$, i.e., $c(r) \times p(t_1)$. Figure 2.6 shows an example of session graphs with processing type t_1, $b(s, t_1) =$

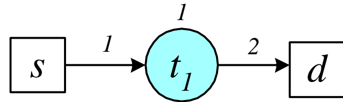
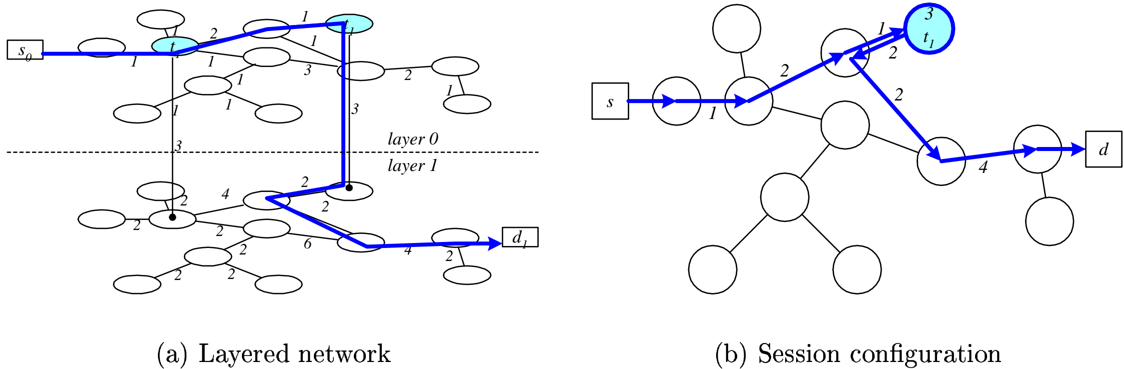


Figure 2.6: Session graph with a single processing step

$1, b(t_1, d) = 2$, and $p(t_1) = 1$. For this session graph, we transform the original graph in Figure 2.3 into a layered graph shown in Figure 2.7(a).

Given the new graph G', the layered network method computes the least cost path from the node s_0, the copy of the source in layer 0, to the node d_1, the copy of the destination in layer 1. Note that G' only carries edge costs, so shortest path algorithms can be applied directly. Figure 2.7(a) also shows the least cost path in the layered network.



(a) Layered network

(b) Session configuration

Figure 2.7: Session configuration using a layered network

For the final solution to the unicast session configuration problem, the least cost path in G' is mapped back to the network graph G as follows. For each regular

edge involved in the path, we "project" it to the original edge in G. Similarly for each inter-layer edge in the path, we "project" it to the original processing node in G and mark it as the designated node for the processing requested in the session graph. The projection yields a legitimate configuration connecting the two terminals and containing the processing node on the path. The projected configuration of the least cost path in Figure 2.7(a) is highlighted in Figure 2.7(b) using thick lines. We claim that this configuration gives the least cost and therefore is the optimal solution.

To prove our claim, let us assume that there is another configuration in the network graph with a smaller cost. We can map this configuration back to a path in the layered network by simply reversing the "projection" procedure. Now, this path has the same end points as the least cost path while having the same cost as the configuration from which it is mapped. However, it has a smaller cost than the least cost path, which is a logical contradiction. Hence, our claim holds.

The layered network method has three steps: construct the layered network, compute the least cost path in the layered network, and project the path back to the original network. The first step can be implemented with $O(|V| + |E| + |R|)$ by iterating on the set of nodes and edges. The second step can be implemented to run in $O(|V| \log |V| + |E| + |R|)$ using Dijkstra's shortest path algorithm. Finally, the projection can be done in time that is linear to the number of edges in the path, which is $O(|V'|)$. Here, the dominant part is the shortest path computation. The comparison method introduced earlier has the same asymptotic time complexity, and may slightly outperform the layered network method in a real implementation. However, the layered network method scales better for multiple steps, as we discuss in the next section.

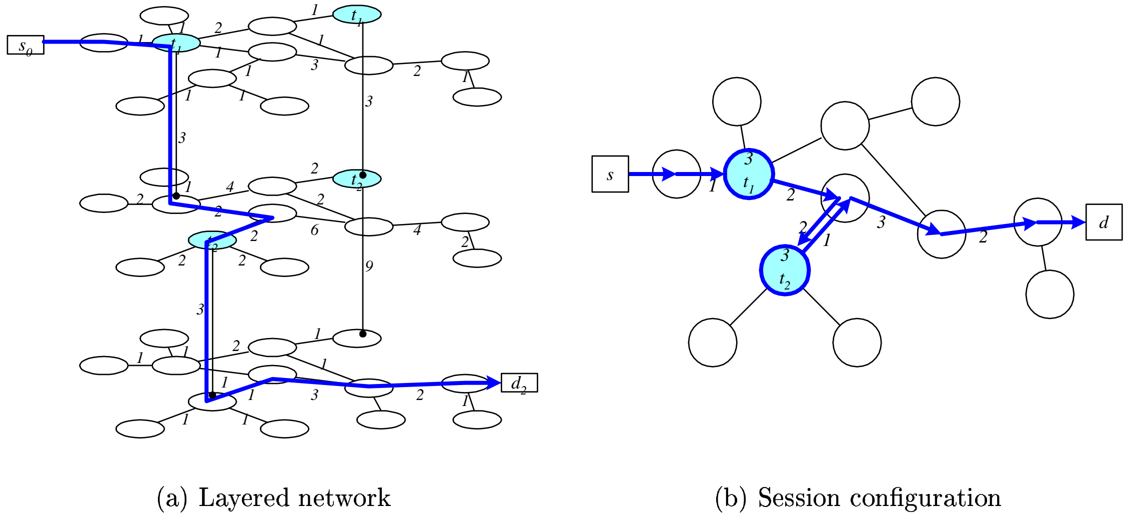
2.3.2 Using layered networks for multiple processing steps

In this section, we generalize the layered network method for an arbitrary number of processing steps. Consider a unicast session that involves two terminals s and d and processing with k consecutive steps of types $t_1, t_2, ..., t_k$, where the bandwidth

requirements are $b(s, t_1), b(t_1, t_2), ..., b(t_k, d)$, and the processing capacity requirements are $p(t_1), p(t_2), ..., p(t_k)$. For each step i, let us denote the set of processing nodes capable of the step as R_i.

We build the layered network in a similar way as single step processing. For k steps, we make $k+1$ copies of the original network, where the copies are denoted layer 0 through layer k. For each link (u, v), we apply the scaled cost $c(u, v) \times b(t_i, t_{i+1})$ to its copy in layer i, where $t_0 = s$ and $t_{k+1} = d$. We also add inter-layer edges between every two consecutive layers as follows. Between layer $i-1$ and layer i, we add the edge (r_{i-1}, r_i) for each processing node $r \in R_i$, and apply the scaled cost $c(r) \times p(t_i)$.

The intuition in this formulation is that any path from layer 0 to layer k is forced to include k inter-layer edges, where the ith inter-layer edge corresponds to one of the processing nodes capable of the ith step. Then, the projection of such a path to the original graph forms a path that passes through the k processing nodes in the given order.



(a) Layered network (b) Session configuration

Figure 2.8: Session configuration using a layered network for multiple processing steps

Therefore, for the given unicast session, we compute the least cost path from s_0 to d_k, where s and d are the source and the destination. The projection of the path on the network graph now forms a legitimate configuration for the session, connecting

the two terminals and containing the k processing nodes. Figure 2.8(a) shows an example of the layered network for a session with two processing steps t_1 and t_2. The bandwidth requirements are $b(s, t_1) = 1, b(t_1, t_2) = 2$, and $b(t_2, d) = 1$, and processing capacity requirements are $p(t_1) = 1$ and $p(t_2) = 3$. The least cost path is drawn with thick lines. The projected configuration of the path is given in Figure 2.8(b).

In fact, the projection of the least cost path is the least cost configuration, which can be proved using a argument similar to the one given for single step processing in Section 2.3.1. Therefore, the layered network method solves the optimal session configuration problem for unicast sessions with an arbitrary number of processing steps.

Again, the time complexity is dominated by the shortest path computation. The layered network contains $(k+1)|V|$ nodes and $(k+1)|E| + \sum_i |R_i|$ edges. The least cost path in the graph can be found in $O((k+1)(|V|\log|V| + |E|) + \sum_i |R_i|)$. In order to find the least cost configuration, particularly as k grows larger, the layered network method outperforms the comparison method. Unlike the layered method, the comparison method must account for all possible ways to select processing nodes and to configure the given session using them.

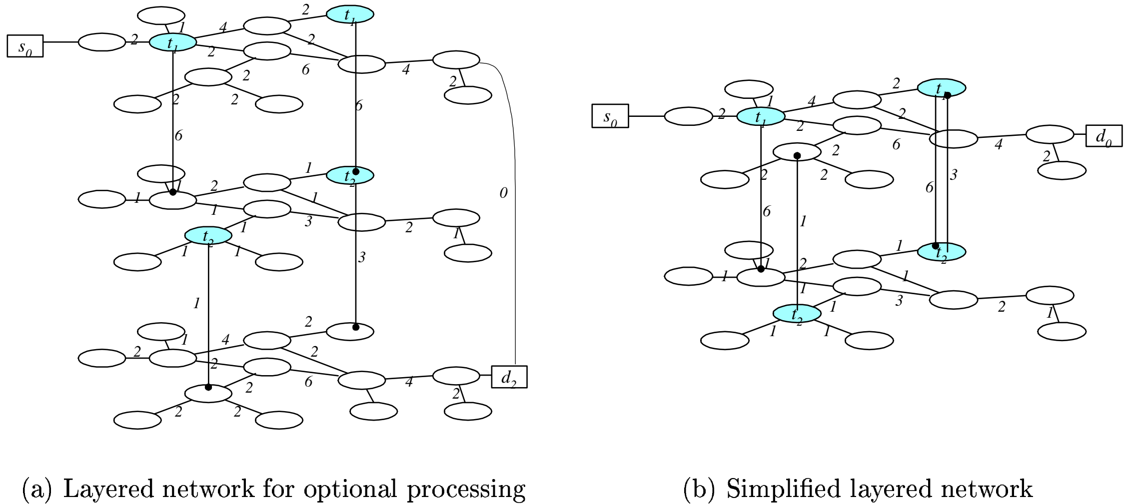
Furthermore, for any given value of k, the layered network method takes no more than about k times the time it takes to compute the regular least cost path in the network graph. Thus, it certainly is feasible for dynamic session configurations. In the next three sections, we consider a series of realistic application examples and demonstrate how the layered network method can be applied to them.

2.4 Applications with optional processing

Some network applications provide services that are not necessary for correct data transmission, but which can improve the performance or quality of the connection.

These optional processing steps might decrease the transmission cost to some destination nodes, but not necessarily to all. We now extend our method to handle such cases.

For concreteness, we use a simple example of a compression/decompression application. The processing for compression and decompression incurs a cost, but the intermediate data stream has a lower bandwidth ($b(t_c, t_d) < b(s, t_c)$), where t_c and t_d represent the processing types for compression and decompression. This lower bandwidth yields lower transmission costs. Thus, for long-distance transmissions the processing overhead is worthwhile, while for short distances, the cost of the added processing may exceed the benefit. The cost/benefit decision may be made by using the method of the previous section. To make the compression and decompression processing optional, we make the destination d shown in the last layer accessible directly from the first layer: we add an edge from the access node of d in layer 0 to d in the last layer. This edge is assigned a cost of zero. Figure 2.9(a) shows the layered network. Note that when the bandwidth of the decompressed data stream matches



(a) Layered network for optional processing

(b) Simplified layered network

Figure 2.9: Layered networks for optional compression/decompression

that of the original, uncompressed data stream, we can actually use a slightly simpler layered network with just two layers, and with edges (u_0, u_1) for all vertices $u \in R_1$

and edges (v_1, v_0) for all vertices $v \in R_2$. The edges within layer 1 are scaled down by the compression factor, as are the edges from layer 1 to layer 0. Figure 2.9(b) shows the simplified layered network.

The method can be extended to configuring sessions where different processing stages are optional. However, when the effects on the bandwidth of the data stream are more complex than in the simple compression/decompression example, a more complex layered graph may be required. These more general cases can be solved using layered graphs that have the first layer connected to multiple columns of layers. Each column contains some subset of the layers for the complete processing, and eventually connects to the destination d below the last layer of each column. The general form of such a graph is illustrated in Figure 2.10. The columns of layers connected from the source s and to the destination d represent possible choices of processing sequences.

2.5 Congestion Control Processing

Application-specific congestion control [26] is often cited as a good example application for active networking. The idea is that an application-specific module could modify the application data stream dynamically in response to network congestion, in a way that minimizes the impact on the application (for example, a video congestion control module might preferentially discard high frequency information, to reduce the subjective impact of the lost information).

For this type of application, the modules should be installed at nodes preceding those links that are most likely to be subject to congestion, but can be omitted from links where congestion is unlikely to occur. If the application is configured to use several congested links, the congestion control module will need to be installed at each of these links. If it is configured to use only uncongested links, then no congestion control modules need to be installed. If a path using several congested links is much shorter than a path that uses no congested links, it may be preferred. We want

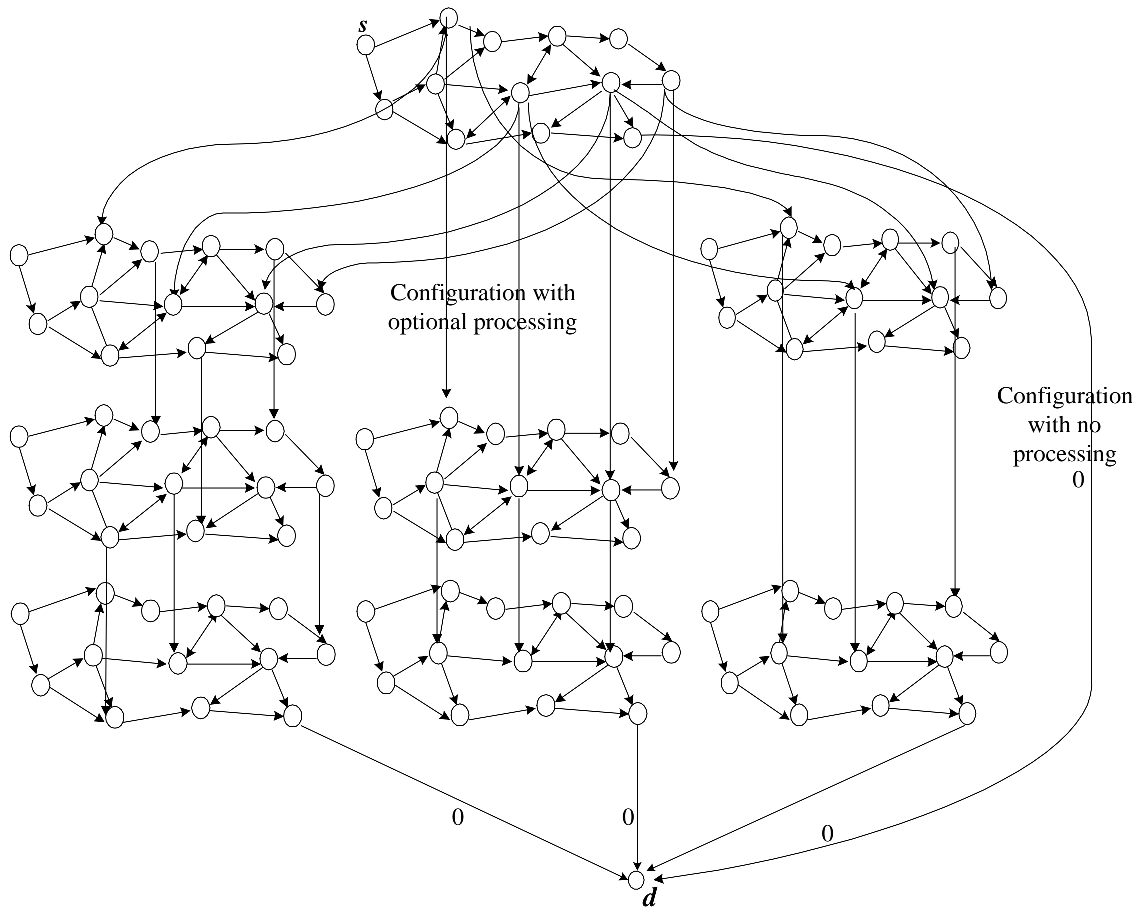


Figure 2.10: Transformed Network for Optional Processing

to formulate the problem so that we can make the best overall choice of a path, considering both the cost of the links and the cost associated with the congestion control (this may include both a processing cost component and a "cost" for the impact of congestion on the application). We can accomplish this simply by modifying the costs of all congested links to reflect the added cost of coping with congestion at those links, and then search for a shortest path, using the modified costs.

The problem is defined formally as follows. The network is represented by a graph $G = (V, E)$ and we let $L \subseteq E$ denote the set of congested links. Each edge (u, v) in the graph has an associated cost $c(u, v)$, and each congested edge has an additional cost $c'(u, v)$. Given a source s and a destination d, our objective is to find

a least-cost path from s to d. The cost of a path includes the cost of its links, and for congested links we include both c and c' in the sum.

2.6 Configuring single source multicast sessions

In this section, we discuss the configuration of multicast sessions, which involve a single source and multiple receivers. The configuration for such a session should provide a set of link resources that connect the source with each receiver in the session. In the particular form of a multicast, the link resources may be shared among data flows terminating at different receivers, and the configuration often forms a tree shape rooted at the source.

We begin by considering the intermediate processing applied to the data flows of such multicast sessions. As an example, consider a video transfer application that provides a single-source multicast session where the video data coming from the source is compressed to reduce transmission cost and then decompressed before it gets to each of the receivers. Figure 2.11 shows a session graph for this example.

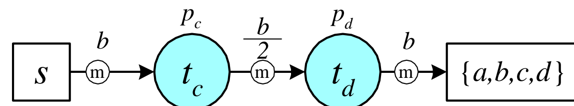
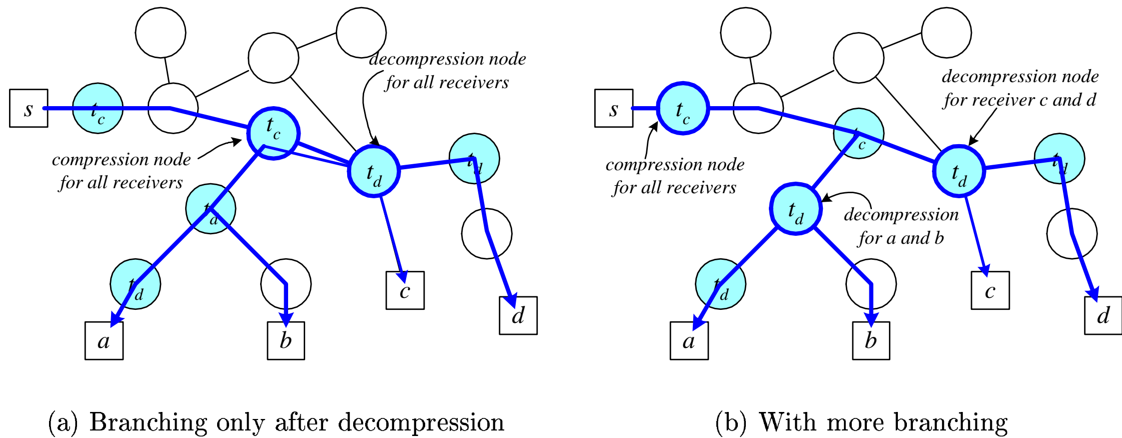


Figure 2.11: Multicast session graph for a video transfer application

The graph resembles the session graph of the unicast session with two processing steps introduced earlier, with the destination node now representing all receivers in the multicast. The edges in the session graph are of a special type to reflect the fact that link resources can be shared among multiple data flows terminating at different receivers. The session graph also shows the changes in the bandwidth requirements when the compression ratio is 50%. Possible configurations of the session are shown in Figure 2.12 with thick lines, each of which forms a tree with different branching points and different choices of processing nodes. The optimal configuration of the session is the one that has the least cost among all possible configurations.



(a) Branching only after decompression

(b) With more branching

Figure 2.12: Multicast session configurations

One way to configure the session is to configure a separate unicast session for each receiver. However, in doing so, we lose the opportunity to share resources raising the overall cost. Instead, we can directly find a multicast configuration in the layered network, which is constructed in the same way as for unicast sessions. Figure 2.13 shows the layered network. To configure the session, we find a multicast tree rooted at s_0 and terminating at a_2, b_2, c_2 and d_2 in the layered network. Then, we project it to the original network graph in the same way as for unicast sessions. The projection corresponds to a proper configuration for the multicast session with the processing applied to all data flows. The multicast tree shown in Figure 2.13 corresponds to Figure 2.12(b).

Therefore, by finding a configuration in the layered network, we configure an equivalent multicast session with processing in the network graph. The layered network method again lets us hide the processing requirement with the graph transformation and lets us solve the problem as a conventional multicast configuration.

Finding the least cost configuration is more complex for multicast sessions. In fact, the multicast routing problem is equivalent to the $\mathcal{NP}$-hard Steiner tree problem in graphs. The Steiner tree problem has been studied extensively for undirected graphs, and there are several approximation methods for it, with the best method known having a worst-case performance ratio of 1.55 [27]. However, those methods

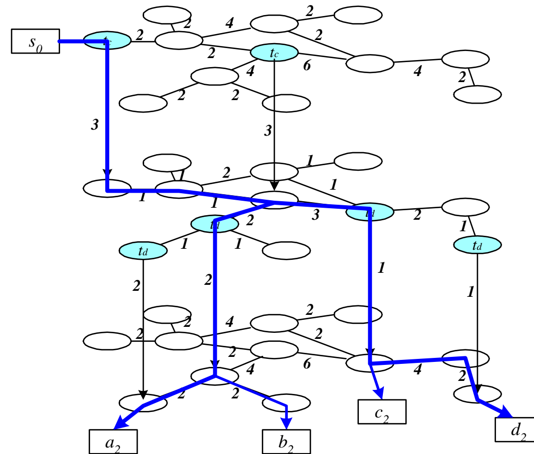


Figure 2.13: Layered network for multicast

cannot be generalized for directed graphs. Only in recent years has the directed Steiner tree problem been studied, and few approximation methods exist. The method presented in [28] gives the ratio of 2Ψ in time $O(k^2b + |E|)$, where k is the number of receivers, Ψ is a measure of the asymmetry of the optimal Steiner tree, and $b < |V|$ is a tunable variable in the algorithm. Another set of methods, described in [29], gives the ratio of $i(i-1)k^{1/i}$ in time $O(|V|^i k^{2i})$, given any $i > 1$.

Unfortunately, the results for directed graphs are not appropriate for dynamic session configurations, since their time complexity is excessive. We suggest two simple heuristic methods which attempt to find a good configuration. Given a multicast group which is composed of a source and multiple receivers, the methods work as described below.

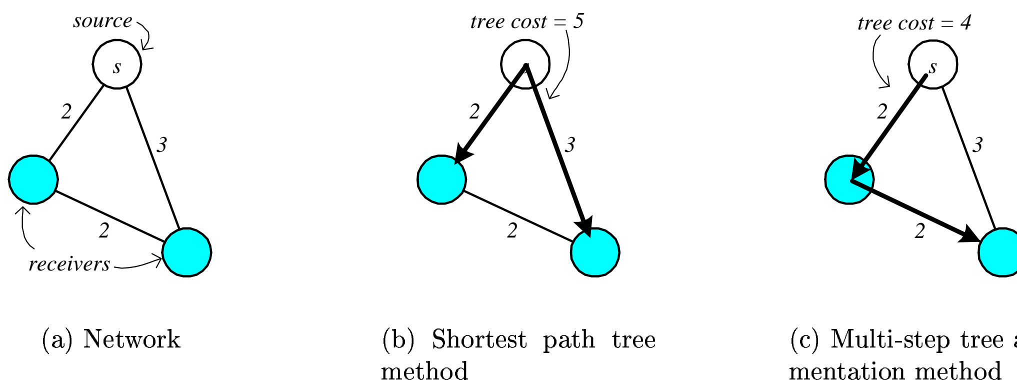
- The **shortest path tree method** first finds the shortest path tree in the layered network rooted at the source. It then extracts a subtree from the shortest path tree so that the subtree contains only the paths that lead to the receivers.

- The **multi-step tree augmentation method** incrementally constructs a multicast tree T, starting with $T = (\{s\}, \{\})$ and a set M, where s is the source and M contains the receivers. It repeats the following step until the set M is empty. Find the receiver $d \in M$ that is nearest to T. The distance from T to d is measured as the shortest

path between any node in T to d. Then, augment the tree T to include the shortest path from T to the selected receiver d, and remove d from M.

We can view these two heuristics as extreme cases of a more general algorithm. The general algorithm is a modification of Dijkstra's algorithm. After steps $k, 2k, 3k, ...$ of Dijkstra's algorithm, we reset the distance variable $d(v)$ to zero, for every vertex that is on any of the paths that connect the source to the receivers in the partial tree constructed so far. For each of the "boundary vertices" v that is connected to any vertex in the constructed path tree, we also let $d(v)$ be the length of the shortest edge connecting v to some vertex with distance value 0. The two heuristics correspond to setting $k = n$ and $k = 1$, respectively. Intermediate values of k yield algorithms of varying time complexity and varying solution quality.

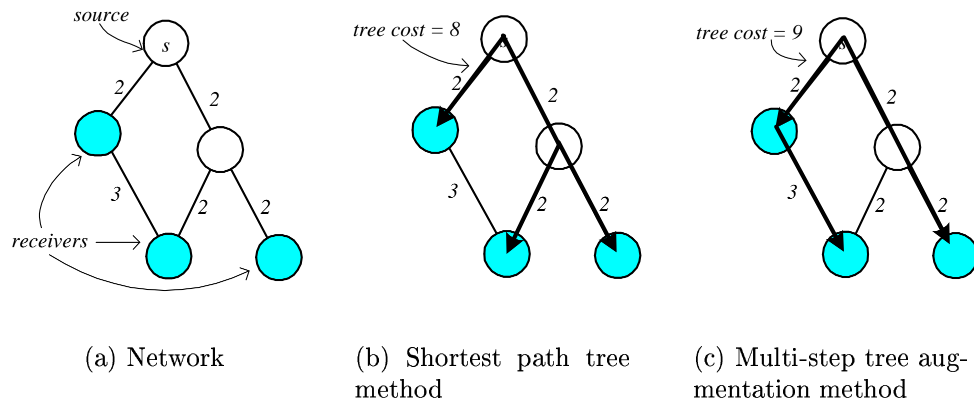
As an upper bound on the costs of multicast trees, we use the sum of the costs of the shortest paths that connect the source to the receivers. This sum represents the cost of configuring multicast sessions simply by independent unicast configurations. By comparing multicast tree costs to the upper bound, we measure the benefit of resource sharing among multiple receivers.



(a) Network (b) Shortest path tree method (c) Multi-step tree augmentation method

Figure 2.14: Comparison of the tree costs resulting from the heuristic methods, where the result of the shortest path tree method costs less than the multi-step tree augmentation

While the two heuristic methods take advantage of resources shared among different receivers in reducing the total cost of multicast trees, it is unclear which of



(a) Network

(b) Shortest path tree method

(c) Multi-step tree augmentation method

Figure 2.15: Comparison of the tree costs resulting from the heuristic methods, where the result of the multi-step tree augmentation costs less than the shortest path tree method

the two method reduces cost more. In fact, neither of them is always better than the other, as shown by the examples in Figure 2.14 and Figure 2.15. Figure 2.14(a) shows a network and a single-source multicast group to be connected. Figure 2.14(b) and Figure 2.14(c) show the multicast trees obtained by the two heuristics. Here, the tree obtained by the shortest path tree method has a smaller cost. On the other hand, Figure 2.15 shows the other case, where the tree obtained by the multi-step tree augmentation method has a smaller cost.

To evaluate the heuristic methods, we performed a simulation study on a network spanning the 50 largest metropolitan areas in the United States. Figure 2.16 shows the network with processing nodes highlighted by triangles. In this network, we consider a multicast application whose sessions are composed of a source and multiple receivers. Here, the source always sends a video stream compressed by 10:1, and the video must be decompressed at a processing node so that receivers can directly display the video. Sessions of this type must be configured on a tree that is rooted at the source and reaches all receivers, and also contains at least one processing node for decompression on each of the source-to-receiver paths in the tree. Our simulation results include approximately 2 million such multicast session configuration attempts.

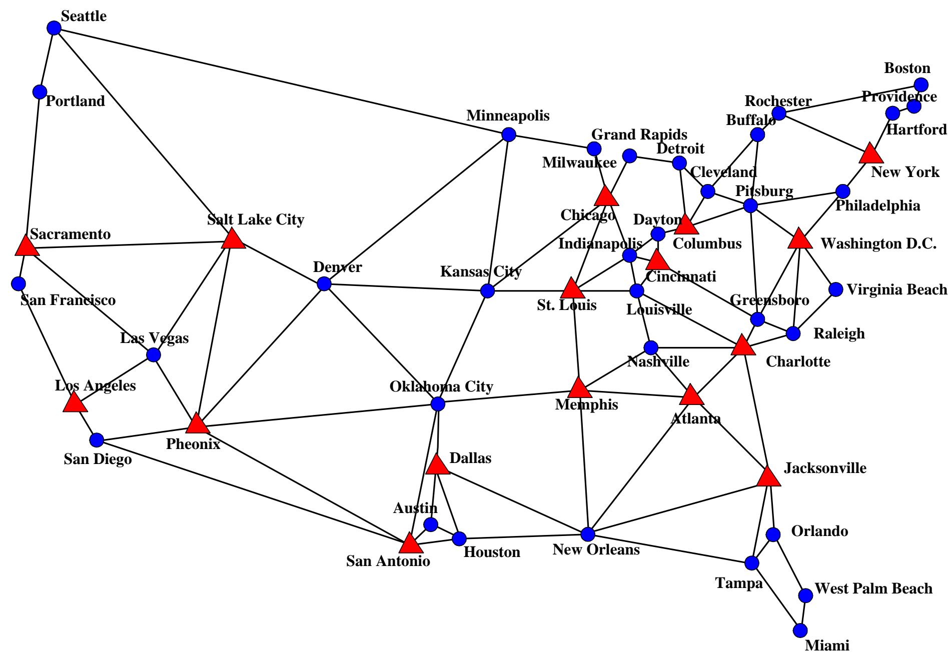
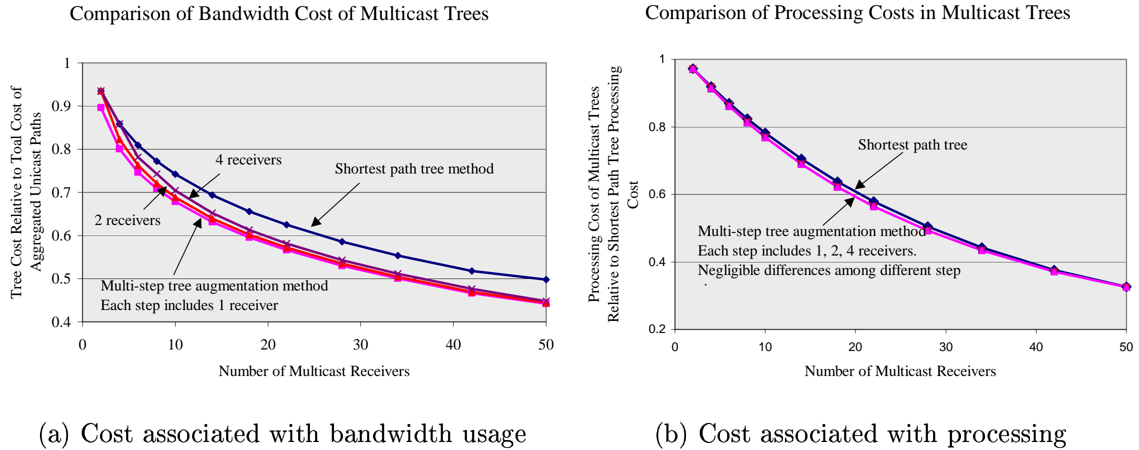


Figure 2.16: Metro 50 Network

In each attempt, we randomly selected a source node and receiver nodes in the network, and applied the heuristic methods to the layered graph constructed from the network given in Figure 2.16. For simplicity, we focused on finding a multicast tree that only minimizes the cost associated with bandwidth usage by not considering the cost associated with processing. This was done by assigning zero costs to inter-layer edges in the layered graph. The decompression processing, however, still affects the multicast tree construction because of its bandwidth expansion. It is preferred to place the processing nodes closer to the receivers to minimize the bandwidth usage while maximizing the bandwidth sharing among the different paths that reach the receivers.

Figure 2.17(a) shows the costs for bandwidth usage in the multicast trees constructed by the shortest path tree method and three variants of the multi-step tree augmentation method, each of which includes 1, 2, and 4 receivers at each step. These costs are shown in relation to the upper bound when the number of receivers is varied from 2 to 50. Remember that given a source and multiple receivers we refer



(a) Cost associated with bandwidth usage

(b) Cost associated with processing

Figure 2.17: Comparison of costs for resources in multicast trees

to the sum of the costs of individual shortest paths that lead to the receivers as the upper bound.

Overall, all heuristic methods show significant cost reductions, by 30% when the number of receivers is 18, and by 50% at maximum. The result implies that more resources are shared among source-to-receiver paths as the number of receivers grows. Among the heuristic methods, when the number of receivers is between 6 and 50, the multi-step tree augmentation method shows approximately 5% more cost saving than the shortest path tree method. Note that source-to-receivers paths are required to include at least one of the processing resources, which are limited to one third of the entire nodes in the given network. Because of this restriction, there is a smaller chance for a receiver to have multiple options to be connected to the tree. Consequently, the multi-step tree augmentation method is less likely to find paths that are different from the shortest paths. This notion explains the relatively small difference of 5% observed in this simulation. For the similar reason, there are only negligible differences among the variants of the multi-step tree augmentation method,

While processing costs are not directly considered in optimizing multicast trees, we still measured the costs associated with processing in the same simulation simply by counting the number of times that processing nodes are used in each multicast tree.

These processing costs are shown in Figure 2.17(b) in relation to the total processing costs of individual unicast paths. Note that because each unicast configuration uses one processing node (for decompression), r processing nodes are needed in total when r is the number of receivers. All methods present significant savings in processing usage compared to the unicast configuration option showing approximately 42% cost reduction with 22 receivers. The cost reduction increases as the number of receivers grows for more processing nodes can be shared among different source-to-receiver paths.

Some applications might have restrictions on path lengths, for example, to avoid extreme transmission delays. We considered the diameters of multicast trees for such applications. In our simulation, the diameter of a tree was the length of the longest source-to-receiver path assuming the geometric distance for the length of each link. Because the multi-step tree augmentation method might choose a path that takes less cost to reach the currently constructed tree than the source, source-to-receiver paths provided in such multicast trees can be longer than the shortest paths. Figure 2.18 compares the diameters of trees computed by the multi-step tree

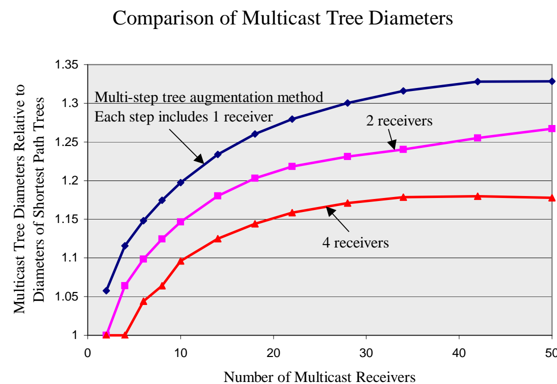


Figure 2.18: Comparison of diameters of multicast trees

augmentation method to the diameters of the shortest path trees. The original multi-step tree augmentation which includes 1 receiver at each step shows up to 33% increase in diameters. Naturally, as more receivers are considered at each step, the difference is reduced, with 18% increase at maximum when 4 receivers are included at each

step. To restrict the diameters of multicast trees while minimizing their resource (bandwidth and processing) usage, the multi-step tree augmentation method can be used by tuning the parameter that sets the number of receivers considered at each step.

2.7 Related Work

Extensible networks expand the scope of network resources by allowing applications to use processing resources at routers in addition to the bandwidth resources in links. The Darwin project [30] focused on the management of this broader set of network resources. One of the core mechanisms in the Darwin system is a resource or service broker called Xena, which discovers and selects the resources that are necessary and (near) optimal for service requests from applications. Once the resources are identified, a signaling mechanism called Beagle conveys signaling messages for resource reservation. Darwin also contains a mechanism that manages and adapts the resource usage at each resource, which is based on a Java code module called a delegate.

Among the mechanisms in Darwin, Xena implements the algorithm to determine the resources to be configured for each service request of an application or an application session. While Xena expresses the algorithm as a 0-1 integer programming problem to solve more general forms of application sessions, our session configuration described earlier in this chapter provides a more efficient algorithm for the most popular subsets of session forms and may be combined with Xena for better performance.

Other relevant work includes routing frameworks and protocols. While we have emphasized fundamental algorithmic issues in configuring resources, implementation strategies need be discussed in order to apply our approach. Current routing frameworks that deliver resource configuration services to application sessions follow the link state routing principle. Link state routing [11] requires each node to maintain at least a partial view of the network state, which is composed of the states of all links in the network. When there is a change in the status of a link, a link state

advertisement (LSA) is flooded throughout the network so that every node can update its view of the current network status and compute its routes to other nodes. In other words, such routing frameworks provide a distributed resource configuration service that allows configuration decisions to be made by multiple nodes in a cooperative fashion. While the same can be done in a centralized fashion by assigning the task to a single server, the distributed system is more suitable for larger networks and is more reliable and robust in general. Such distributed systems must include a component that distributes information about network resource availability and a component that uses that information to make configuration decisions with respect to specific sessions.

The ATM *Private Network-Network Interface* protocol (PNNI) [14] is an example of a distributed resource allocation system that solves a similar problem. PNNI can be viewed as two protocols, a link-state protocol that distributes information about network resource availability and a signalling protocol that uses this information to make virtual circuit routing decisions. In the case of PNNI, the route from a source to a destination is selected by the switch connected to the source, using stored information about the network topology and resource availability. The protocol then passes the selected route to other switches along the path. They, in turn, make local resource reservations and propagate signalling messages along the path. During this process, if an attempt to make a local resource reservation fails, a new path may be computed by the switch at the point where the reservation failed, allowing the path setup process to continue. To make the approach scalable to very large networks, the PNNI protocol aggregates information about sections of the network, allowing switches to have complete knowledge of the portion of the network that is close to them and more summary knowledge of distant portions of the network.

The general approach taken by the PNNI protocol can be extended to handle configuration of sessions requiring intermediate processing. The state information distributed by the routing protocol must be expanded to include information about

processing resources available at various locations in the network. Using this information, a path can be computed by the router connected to the source of a unicast session, and then forwarded in a signalling message to successive routers on the path to the destination, allowing local resource reservations to be made as the signalling message proceeds to the destination. Of course, as with the basic PNNI protocols, the selected paths may not be globally optimal, since initial path selections may be based on summary information about distant portions of the network. This is nothing new in network routing, where optimality of path selection must generally be sacrificed for the sake of scalability.

Other approaches are possible as well. In particular, other link state protocols, such as Open Shortest Path First (OSPF) [13], can be used to distribute state information, and other signalling protocols can be used to select paths and make the required resource reservations.

Chapter 3

Resource Configuration in Capacity Constrained Networks

In this chapter, we study the configuration of sessions when the network has hard limits on the resource capacities to be consumed by application sessions. For instance, a video transfer application may require a fixed bandwidth available on the links in order to achieve desirable video quality for its interactive sessions, while some link resources may not have enough capacity to provide the required bandwidth. Figure 3.1 shows an example session of a video transfer application in which the sender sends 5 Mb/s of video stream compressed by 10:1 to a receiver, that is incapable of decompressing the video. For such a receiver, the network provides processing nodes that can handle the decompression, as shown in the figure. Given a uniform cost for using any link in the network, Figure 3.1 shows in dotted lines the least-cost path configuration computed by the layered graph method. A valid configuration must contain at least 5 Mb/s on the links that are used to connect the source to the processing node; it must contain 50 Mb/s on the links that are used to connect the processing node to the receiver because the decompression expands the data bandwidth by 10 times. In the figure, we show the bandwidth available at each link in units of Mb/s. Note, however, that the least-cost path does not provide enough capacity. Meanwhile, the other configuration, shown by solid lines, satisfies

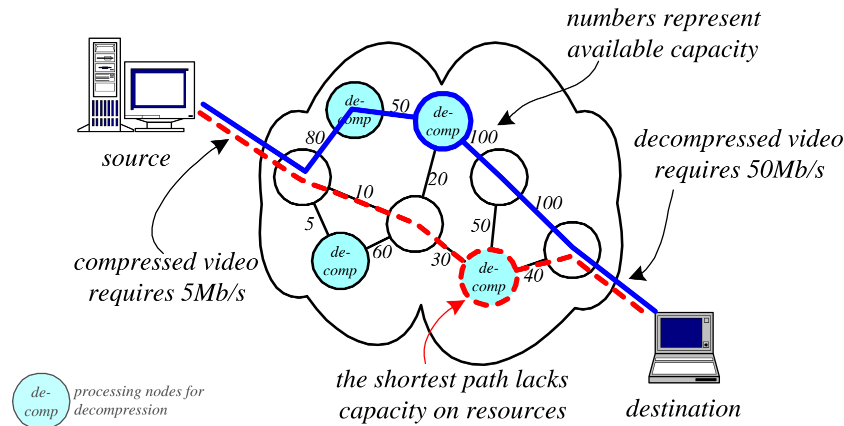


Figure 3.1: Video transfer application with bandwidth requirements

the bandwidth requirement and therefore is a valid configuration. Likewise, processing nodes may also be required to have a certain amount of processing capacity in order to handle the target processing, while some nodes lack the required capacity.

In order to guarantee resource availability, the available capacity must be considered, as well as the cost of each resource. In the session configuration problem given in Problem 2.2.1, however, we did not include capacity limits in the network graph, and considered every resource in configuring sessions regardless of the available capacity. To provide resources with adequate capacity, we generalize the resource configuration problem, and discuss methods for the generalized problem in this chapter.

3.1 Generalized Resource Configuration Problem Redefined

In order to explicitly consider the capacity issue, we redefine the session configuration problem with the available capacity at each resource specified in the network graph. The problem statement is given in Problem 3.1.1.

Problem 3.1.1 *Session Configuration Problem for Capacity Constrained Networks*

Given: *A directed session graph* $G_s = (V_s, E_s)$, *with a type* $t(u)$ *and a capacity requirement* $p(u)$ *for each vertex* $u \in V_s$, *and a bandwidth requirement* $b(u, v)$ *for each edge* $(u, v) \in E_s$.

Also, a directed network graph $G = (V, E)$ *with a type set* $T(u)$ *and processing capacity* $P(u)$ *for each vertex* $u \in V$, *and available bandwidth* $B(u, v)$ *for each edge* $(u, v) \in E$.

In addition, the unit costs, $c(u)$ *and* $c(u, v)$, *are given for each vertex and edge as a positive integer value.*

Find: *A location function* $l : V_s \to V$ *and a routing function* $r : E_s \to 2^E$ *that satisfy*

$$\forall u \in V_s, \quad t(u) \in T(l(u)) \tag{3.1}$$

$$\forall (u, v) \in E_s, \quad r(u, v) \text{ is a simple path in } G \text{ from } l(u) \text{ to } l(v) \tag{3.2}$$

$$\forall x \in V, \quad \sum_{\substack{u \in V_s : \\ x = l(u)}} p(u) \leq P(x) \tag{3.3}$$

$$\forall (x, y) \in E, \quad \sum_{\substack{(u,v) \in E_s : \\ (x,y) \in r(u,v)}} b(u, v) \leq B(x, y) \tag{3.4}$$

and that minimize the cost $C_{l,r}$ *where*

$$C_{l,r} = \sum\nolimits_{u \in V_s} c(l(u))p(u) + \sum\nolimits_{(u,v) \in E_s} c(r(u, v))b(u, v) \tag{3.5}$$

This problem extends Problem 2.2.1 further by adding the capacity constraint of each resource in the network graph, while keeping the objective of finding the optimal location and routing functions. Conditions (3.3) and (3.4) state that the available capacity should be sufficient to accommodate the total capacity consumption at each

resource. More specifically, Condition (3.3) states that the available capacity at each processing node must be greater than or equal to the total capacity required at the node by the given session. Similarly, Condition (3.4) states that the available bandwidth at each link must be greater than or equal to the total bandwidth required at the link by the session.

Recall that a new variable has been added to the network graph for each resource to express the available capacity, P for a processing node and B for a link. We refer to this network model as a capacity-constrained network. Given the new network model and the capacity requirement, we now reconsider the optimal configuration problem, focusing on unicast sessions.

First, configuring sessions that do not require intermediate processing can be done simply by eliminating the links that lack the required bandwidth from the network graph and finding the least cost path in the "reduced" network. Because the "reduced" network contains only links with enough capacity for the bandwidth requirement, and no link is used more than once (in the least cost path), we can guarantee that the optimal configuration computed in this network will always satisfy the capacity requirement.

On the other hand, because a single resource may be used multiple times in a selected configuration, the same strategy may not always work for sessions that do require intermediate processing. An example is given in Figure 3.2, where a session

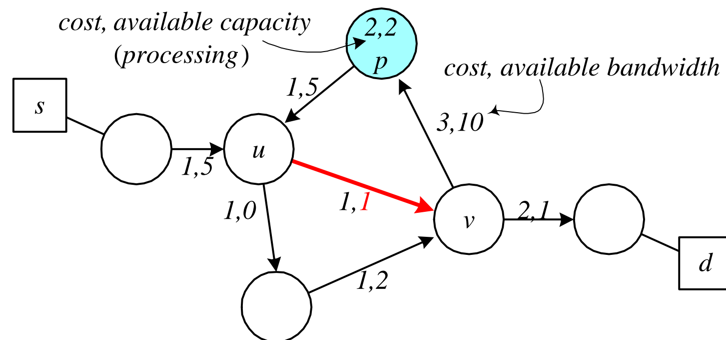
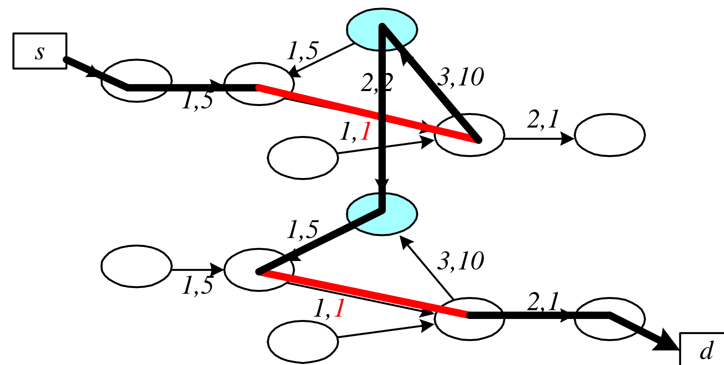


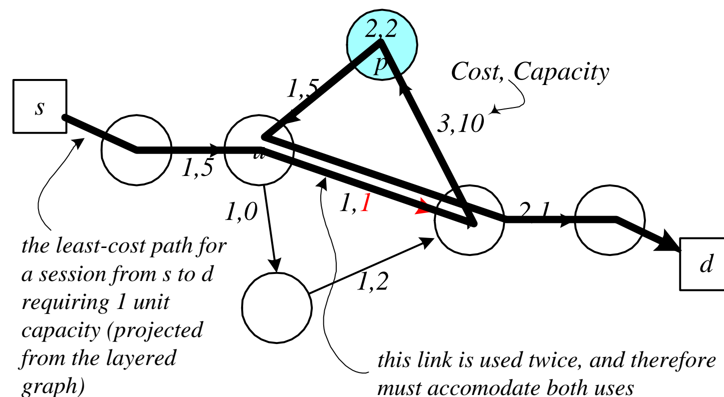
Figure 3.2: Network graph with a cost and capacity specified for each resource

from s to d is to be configured with one processing step. Here, the shaded node is

the only processing node capable of the processing, and each resource is associated with two values, the first being its cost and the second being the available capacity. When the required capacity is 1 unit, we can modify the network by eliminating the links with less than 1 unit of available capacity. Figure 3.3(a) shows both the layered graph composed from this reduced network and the least-cost path. However, this



(a) Layered graph and the least-cost path



(b) Projected configuration

Figure 3.3: Session configuration in capacity constrained networks

least-cost path is projected to a configuration in the original network, which overuses a resource. Figure 3.3(b) shows that the projected configuration uses the link (u, v) twice, consuming 2 units of bandwidth, while the available bandwidth at the link is only 1 unit. While the session can be configured on this path if it requires ≤ 0.5

units of bandwidth, there is no possible path otherwise, because the total bandwidth requirement exceeds the available bandwidth of any path from s to d. To find a valid configuration, for each resource a decision has to be made whether to exclude the resource from consideration according to its available capacity and the capacity requirement. However, it is unclear how to foresee the non-uniform resource usage as described earlier without examining all possible paths.

These difficulties arises because the general problem of configuring sessions with capacity constraints and processing requirements is intractable. Consider a complete network, $G = (V, E)$, where every node except s and d is capable of any type of processing, and each has 1 unit of available capacity. Now, consider a session from s to d which requires $|V| - 2$ processing steps. Any feasible configuration to this problem must pass through all the intermediate nodes, and thus it provides a solution to the well-known Hamiltonian path problem [31], which is known to be $\mathcal{NP}$-complete.

Given the intrinsic intractability of the problem, we turn next to a study of heuristic algorithms. In the next two sections, we introduce two heuristic methods for the optimal session configuration problem in a capacity-constrained network. We focus on unicast sessions that specify a processing requirement for each step and a bandwidth requirement for each path segment between two consecutive steps. Note that the bandwidth requirement can differ on different path segments, since processing steps may expand or reduce the amount of data. As discussed in Chapter 2, the costs in the different layers are scaled to account for such effects. Our heuristics extend the layered graph method to prevent resources from being used beyond their current capacity.

3.2 Heuristics: Selective resource consideration

The first heuristic is really a collection of similar algorithms that we refer to as the selective inclusion algorithms. Each modifies the layered graph to prevent links from

being over-used and then finds a shortest path in the modified graph. The algorithms differ in the way they modify the layered graph.

- The **Strict inclusion method** includes an edge in the layered graph only if it has enough capacity so that it cannot be over-used, even if it is selected for use in all layers. This policy applies to both intra-layer and inter-layer edges. Since different processing steps may require different amounts of processing capacity, we include a given edge as an inter-layer edge only if the sum of the capacities required for all the processing steps is no larger than the available capacity of the processing node represented by the inter-layer edges. Similarly, we include a given intra-layer edge only if its capacity is no smaller than the sum of the bandwidth requirements for all path segments. Once the modified layered network is constructed, a shortest path from the source to the destination is computed. If none exists, the session configuration attempt is rejected.

- The **Loose inclusion method** includes an edge in all layers if it has sufficient capacity to be used in any one of the layers. If, after a path is determined, the path is found to over-use an edge, the path is discarded and the session configuration attempt is rejected.

- The **Permissive loose inclusion method** is not intended as a practical algorithm, but is used in our simulation study to provide a nominal bound on the performance of the other algorithms. It works like the loose inclusion method, except that it never rejects the path that is found, even if the path over-uses an edge. In this way, the permissive loose inclusion method provides a lower bound on the blocking probability of session configuration attempts given any method. Here, the blocking probability is the rate of unsuccessful session configuration attempts to the total number of attempts.

- The **Random inclusion method** includes edges in a set of selected layers, for which the total capacity requirement is no larger than the edge capacity. For each edge, the layers are selected randomly and independently. Once the modified layered network

is constructed, the shortest path search is performed. If successful, the session is configured using that path.

- The **Consecutive inclusion method** first selects a layer at random and then goes through the remaining layers in consecutive order, adding the edge to each layer in which the addition cannot violate the capacity constraint.

The selective inclusion methods are very simple to implement and, according to our simulation results given later in this chapter, can perform reasonably well when the session resource requirements are much smaller than the capacities of the links and processing nodes Note that resource over-uses are most likely to occur when resource requirements take a significant portion of the remaining capacity, for example 60%. This finding implies that the selective inclusion methods do not handle resource over-use well.

3.3 Heuristics: capacity tracking

Our second heuristic is somewhat more complex but can perform well, even when session resource requirements are relatively large. The algorithm is an extension to Dijkstra's shortest path algorithm, and is called the **capacity tracking** algorithm. We start with a brief review of Dijkstra's shortest path algorithm.

Given a graph, and a source node s, Dijkstra's algorithm computes a shortest path tree rooted at s. Initially, the tree contains just s. The algorithm maintains a set of boundary vertices, S, which includes all nodes v that are connected to some vertex u in the partial tree constructed so far, by a directed edge (u, v). At the start of the algorithm, S contains the nodes v, for which there is an edge of the form (s, v). The algorithm also maintains, for each vertex v, a tentative distance $d(v)$, which is the length of the shortest path from s to v that has been found so far. It also maintains a tentative parent $p(v)$, which is the predecessor of v in a path from s of length $d(v)$. The quantities $d(v)$ and $p(v)$ are not defined for nodes that are neither in the tree nor in S.

At each step, Dijkstra's algorithm selects a node v in S for which $d(v)$ is minimum, and adds it to the tree. It then examines each of the outgoing edges of v, i.e., each edge of the form (v, w). For each node w that is neither in the tree nor in S, it adds w to S, setting the tentative parent of w to v, i.e., $p(w) = v$, and setting the tentative distance of w to $d(v)$ plus the length of (v, w), i.e., $d(w) = d(v) + length(v, w)$. For each node w that is in S, it compares $d(w)$ to $d(v, w)$ plus the length of the edge (v, w), and if it finds that $d(w)$ is larger, it updates $d(w)$ and $p(w)$. If the set of boundary vertices is implemented using a Fibonacci heap [32], Dijkstra's algorithm runs in $O(m + n \log n)$ time, where n is the number of nodes in the graph, and m is the number of edges.

When Dijkstra's algorithm is applied to a layered graph, some of the paths in the shortest path tree may contain edges on different layers that correspond to the same link or node in the original network from which the layered network was constructed, leading to over-use of resources. To prevent this, we modify the basic processing step to include a check for over-used resources. In particular, when a node v_i is added to the tree (i denotes the layer in which the vertex appears), we consider edges of the form (v_i, w_i) and (v_i, v_{i+1}). Before processing an edge of the form (v_i, w_i) which belongs to layer i, we examine the path in the tree from s to v_i and add up the capacities required by all edges on the path that correspond to the original link (v, w). If this total capacity, plus the capacity that would be used by the edge (v_i, w_i), exceeds the available capacity of the link, then no action is taken with respect to that edge. Figure 3.4 shows an example of this case, where two intra-layer edges corresponding to the link (v, w) are considered in the shortest path tree rooted at the node s. The intra-layer edge considered in the second layer cannot be used because the available capacity (1 unit) has already been used in the first layer. The dotted lines show the part of the shortest path computed by the original Dijkstra's algorithm, which is not used by this heuristic method due to the insufficient capacity. Edges of the form (v_i, v_{i+1}), which are inter-layer edges, are handled similarly according to the available

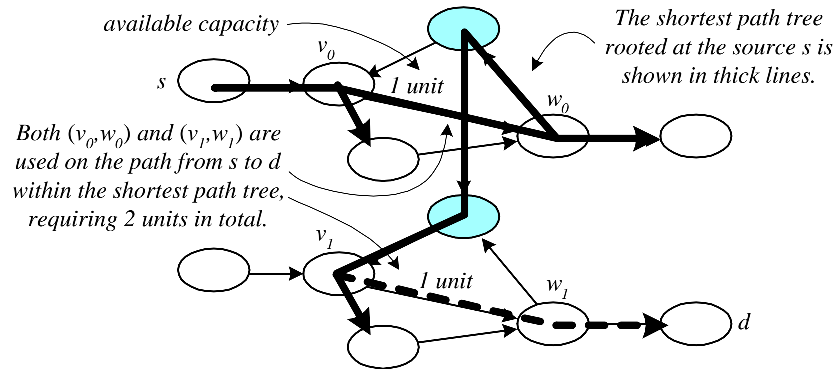


Figure 3.4: Shortest path tree in a layered graph

capacity of the processing nodes associated with these edges. We refer to this capacity checking procedure as capacity tracking.

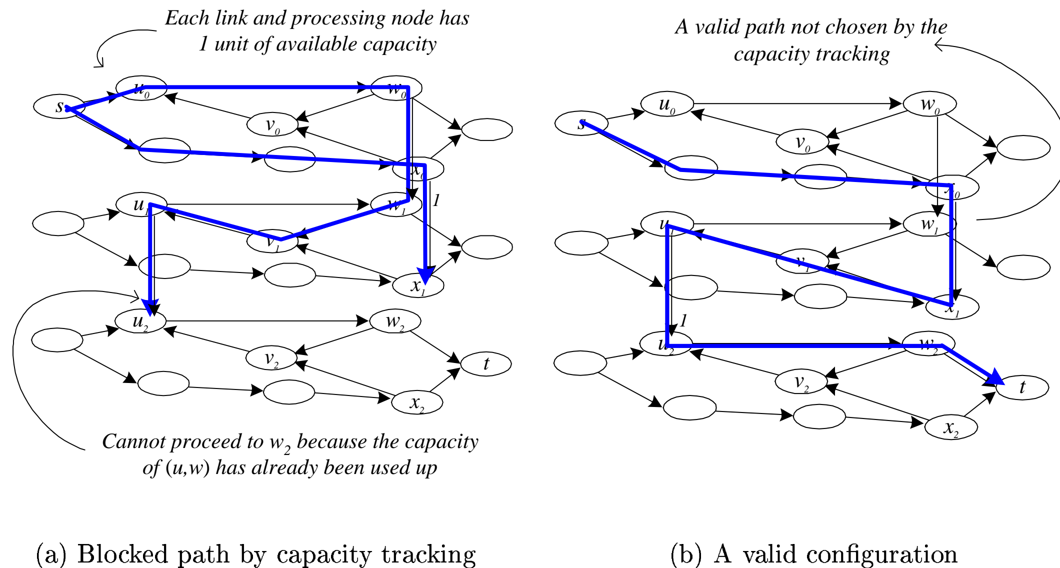
In the worst case, the extra time required by capacity tracking is $O((km)(kn))$, where m and n are the number of edges and nodes in the original network and k is the number of processing steps. This can be seen by noting that the checking procedure is invoked no more than $k(m+n)$ times, and each execution requires that we traverse a path with no more than $kn-1$ edges.

The running time can be improved by maintaining an additional variable $\kappa(v_i)$ for each vertex in the partial tree constructed so far. If u_i is the tentative parent of v_i, i.e., $u_i = p(v_i)$, then $\kappa(v_i)$ denotes the sum of the capacities required from all edges on the tree path from s to v_i that are copies of the link (u, v) in the original network graph. Similarly, if $v_{i-1} = p(v_i)$, then $\kappa(v_i)$ is the sum of the capacities required from all edges on the tree path from s to v_i that correspond to the processing node v in the original network graph. Using these additional variables, we can terminate the capacity tracking search from a node v_i back to s early, reducing the time taken for capacity tracking to $O(kmn)$.

In practice, the extra time required by capacity tracking is much smaller than the worst-case analysis suggests, because networks are designed to have small diameters, which means that the paths in the shortest path tree generally have far fewer than kn edges. If we let D denote the maximum number of edges in a path from the

root to a vertex in the shortest path tree, then the extra time required by capacity tracking is $O(kmD)$. Even this result over-states the time required by capacity tracking in practice. As will be seen later, running time measurements in more realistic situations show that capacity tracking takes less than twice the time required by the simpler heuristics.

Capacity tracking ensures that paths found by the algorithm do not overuse any resources. However, since the problem is $\mathcal{NP}$-hard, we cannot expect it to always find a valid path, even when a path exists. Consider the example shown in Figure 3.5(a). If each link in the original network graph has one unit of capacity, and



(a) Blocked path by capacity tracking

(b) A valid configuration

Figure 3.5: Blocked path in capacity tracking

the session requires one unit of capacity on each edge of the selected path, it can fail to find a path, as shown in part (a) of the figure. The bold edges are the edges that form the shortest path tree at the time the path search terminates. Note that there is no way to extend the tree further, since the only edge leaving vertex u_2 has already been used in the top layer, and hence cannot be used again. On the other hand, there is a path that could be used for this session, as shown in part (b).

3.4 Simulation Results

We performed a set of simulations for the session configuration problem to evaluate the heuristic methods discussed so far. In the simulations, we considered the following four different network topologies.

- Torus: This network is based on a grid of 64 nodes in which every node has an outgoing edge to each of its four neighbors – north, south, east and west – along the grid lines. The nodes at edges of the square grid also have links that "wrap around" to the corresponding node at the opposite edge, resulting in a torus topology. Figure 3.6 shows the network topology. Nodes that can perform processing are shown as triangles.

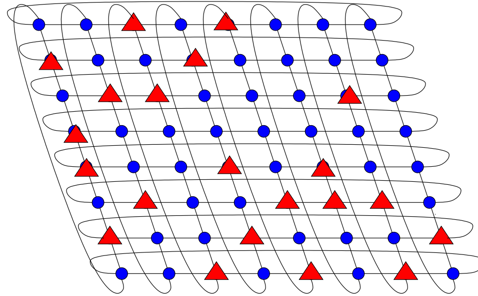


Figure 3.6: Torus network

- Random: This network is a random regular network with 64 nodes, each having 4 incident edges. We build the network starting with a random degree-bounded tree that spans all 64 nodes, then we expand the network by adding edges randomly until every node has exactly four incident edges.

  In both networks, every link has the same capacity and the same cost, and one third of the nodes are randomly designated as processing nodes, with the ability to perform processing. All processing nodes have the same capacity.

- Metro 20: This network is a more realistic network configuration, spanning the 20 largest metropolitan areas in the United States. The network topology is shown in

Figure 3.7. Nodes that are capable of performing processing are shown as triangles. Link costs are set equal to the physical distance between the nodes they connect,

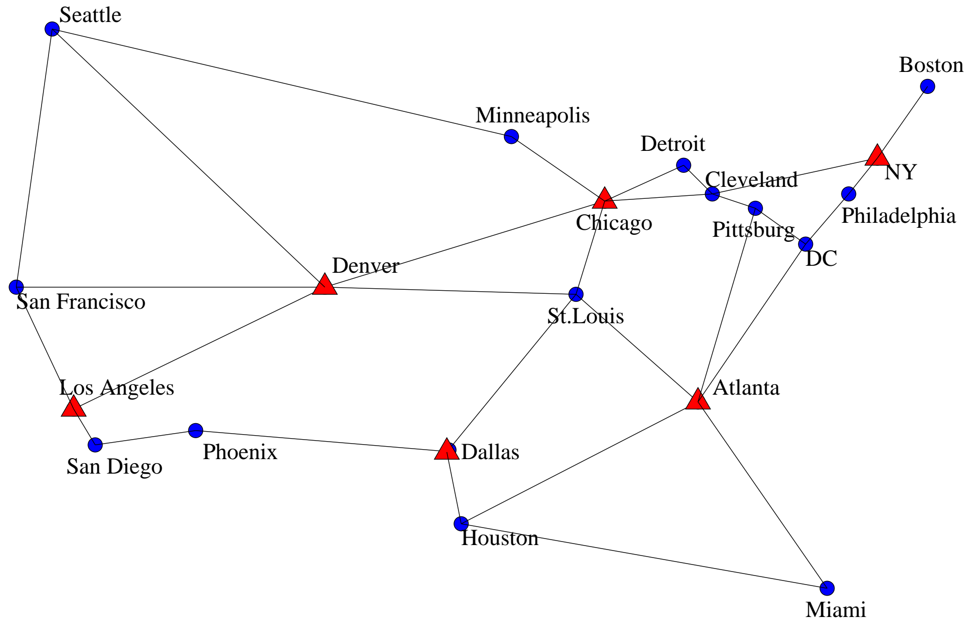


Figure 3.7: Metro 20 Network

reflecting the higher cost associated with links spanning greater distances. The link capacities are selected to be large enough to handle the anticipated traffic. The link dimensioning procedure used for this purpose is taken from [17], which describes a constraint-based network design methodology and an interactive network design tool that implements it.

We constrain the traffic in two ways. First, the total traffic entering and leaving a node is chosen to be proportional to the population of the metropolitan area represented by that node. Next, for each node u, we constrain its traffic to every other node in proportion to the populations of the metropolitan areas represented by the other nodes. Specifically, if δ_v is the fraction of the population outside node u that is associated with node v, then we limit the traffic between u and v to be no more than $1.3\delta_v$ times the total traffic entering and leaving node u. The factor of 1.3 was chosen to allow for some flexibility in the distribution of traffic, reflecting the natural variations that occur in network traffic.

Given these traffic assumptions and a *default path* joining each pair of vertices, link dimensions can be computed using linear programming. The resulting link capacities guarantee that any traffic pattern satisfying the traffic constraints can be carried if the traffic is routed along the default paths. The default path between a pair of vertices is a shortest path containing at least one processing node, and can be found using a two layer network. The processing nodes along each default path are dimensioned to handle the worst-case traffic load allowed by the traffic constraints. When performing the simulations, we do not constrain the traffic to use just the default paths, but the link dimensions are chosen under the assumption that the default paths are used.

- Metro 50: This network, a larger version of the Metro 20 network, has a node for each of the fifty largest metropolitan areas in US. The topology is shown in Figure 3.8. The links and processing nodes are dimensioned in the same way as in the Metro 20.

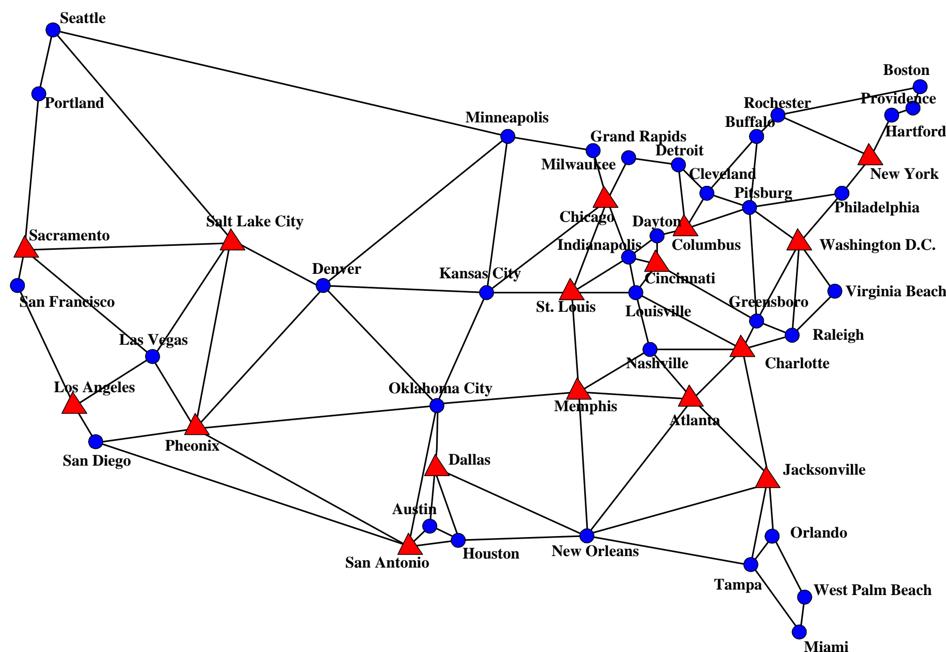


Figure 3.8: Metro 50 Network

While the Torus and Random are not particularly realistic network configurations, they provide a more “neutral” context for evaluating the session configuration

algorithms than the somewhat idiosyncratic network topologies that arise from real world considerations. By considering a variety of different networks, we hope to avoid drawing conclusions that may be attributable purely to special properties of a particular network.

There are several configuration parameters that affect the simulation results.

- *Density of processing nodes* (P): The density of processing nodes is just the ratio of the number of nodes that can perform processing to the total number. In the results reported here, $P = \frac{1}{3}$. The processing nodes were randomly selected for the Torus and Random topologies, and were configured for the Metro 20 and Metro 50 as shown in Figure 3.7 and Figure 3.8, where processing nodes are drawn as triangles.

- *Session capacity requirement* (BW_s): The capacity that an individual session uses at each link and processing node, BW_s, is set to 3% of the average link capacity.

- *Number of steps* (N_{steps}): The number of processing steps that a session requires.

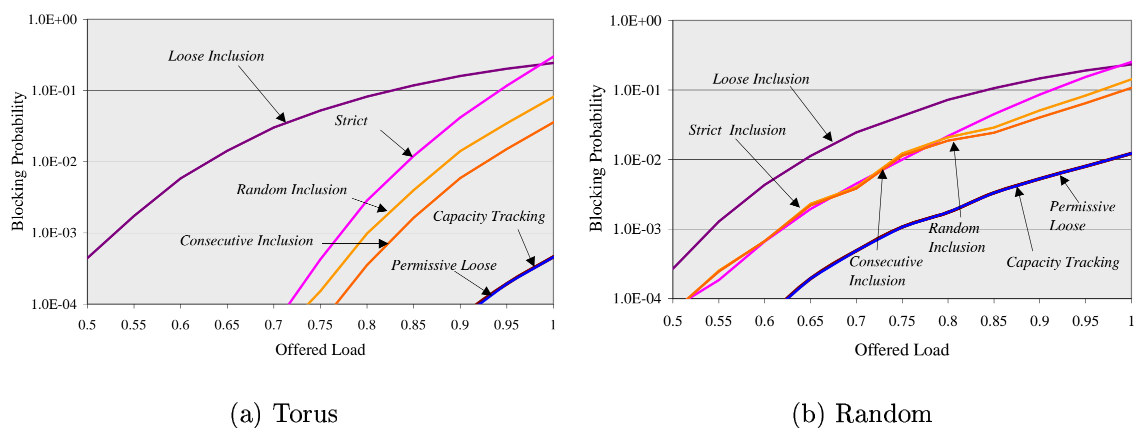
- *Offered load at links* (O_l): The average offered background traffic level on each link. The simulation was done by generating background traffic levels independently at each link and node, then attempting to connect random pairs of nodes. This procedure was repeated multiple times to produce the reported results. Each simulation run included over 2.5 million session setup attempts. The background traffic was generated using an $M/M/k/0$ queueing model (k servers and zero length queues, where k is the ratio of link capacity to session bandwidth).

- *Offered load at processing nodes* (O_p): The average offered background traffic level at each processing node. For the results reported here, the offered load at the processing nodes is the same as the offered load on the links.

The selection of the end nodes of the sessions was done completely randomly for the Random topology. For the Torus, the selected node pairs were restricted to be exactly four hops apart. For the Metro 20 and Metro 50, the selection of the

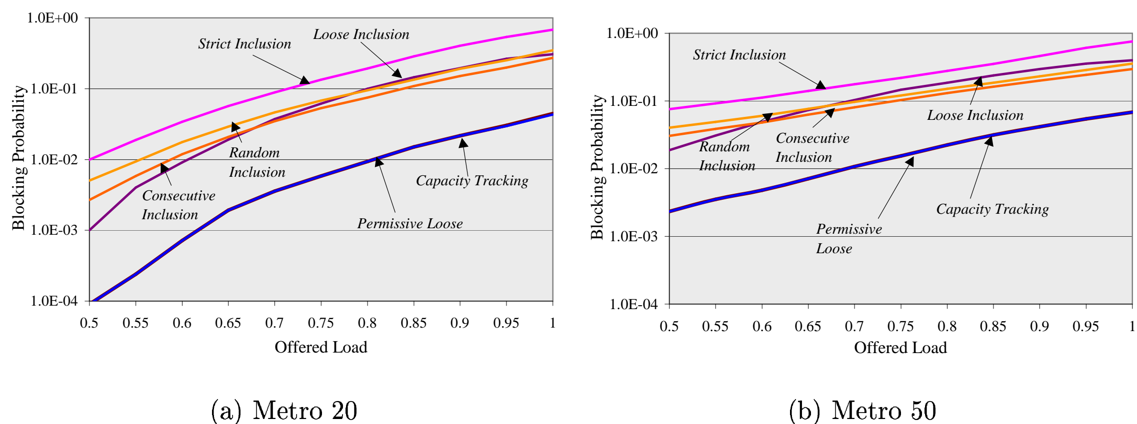
end nodes was weighted by the populations of the cities, reflecting the higher traffic volumes expected in larger cities.

Our primary performance metric is the blocking probability, which is the percentage of session configuration attempts that were unsuccessful. Figures 3.9 and



(a) Torus (b) Random

Figure 3.9: Performance of heuristics for session configuration



(a) Metro 20 (b) Metro 50

Figure 3.10: Performance of heuristics for session configuration, the metropolitan area networks

3.10 show the blocking probabilities for the various heuristics as a function of the offered load. The plots also show the blocking probability when paths are constrained to use the default path. Recall that the default paths were used in the dimensioning process, so this restriction is worth considering as a point of comparison. In general,

however, the lack of routing flexibility implied by this policy results in higher blocking probabilities than with the other algorithms.

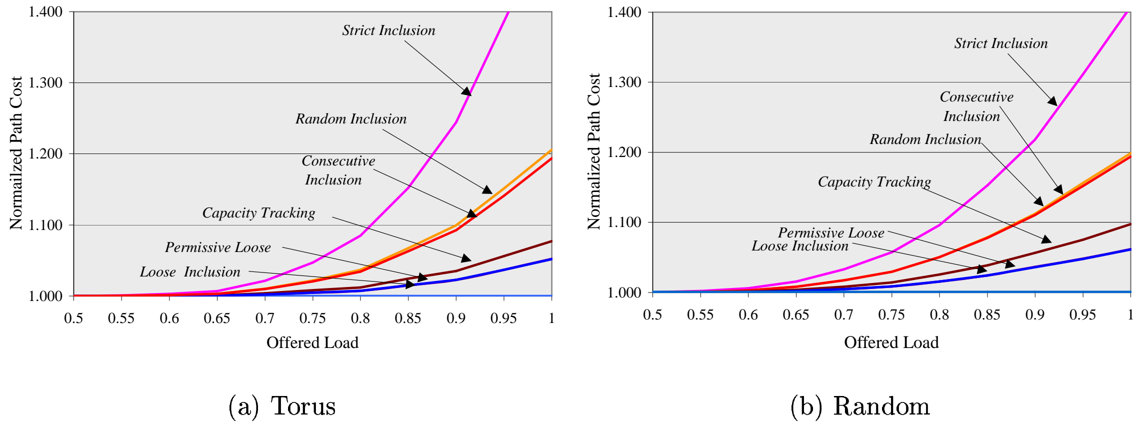
For all four networks, capacity tracking outperformed the heuristics that use selective inclusion. Also, note that capacity tracking generally performs almost as well as the permissive loose inclusion method, which is included as an idealized bound on algorithm performance.

For the Torus topology simulation, every heuristic method except loose inclusion results in blocking probability less than 1% for load up to 80%. (See Figure 3.9(a).) Note that Torus has many paths between selected end nodes, and therefore, the heuristics that avoid overusing resources by ignoring some edges in the layered graph still have a good chance of finding valid paths in the reduced graph. On the other hand, loose inclusion does relatively poorly, apparently because it often selects paths that over-use resources (primarily processing nodes).

For the Random topology simulation, all the better algorithms experience a higher blocking probability than for the Torus. The explanation appears to be the variety of paths available between endpoint pairs in the Torus and the limited separation between endpoints in the Torus simulation. With the Random topology, endpoints were simply selected at random, so many pairs are likely to be further apart than the four hops that constrained the choice of endpoint pairs in the Torus simulation. In Random simulation, there also tend to be fewer good “second-choice” paths when the preferred path is not available.

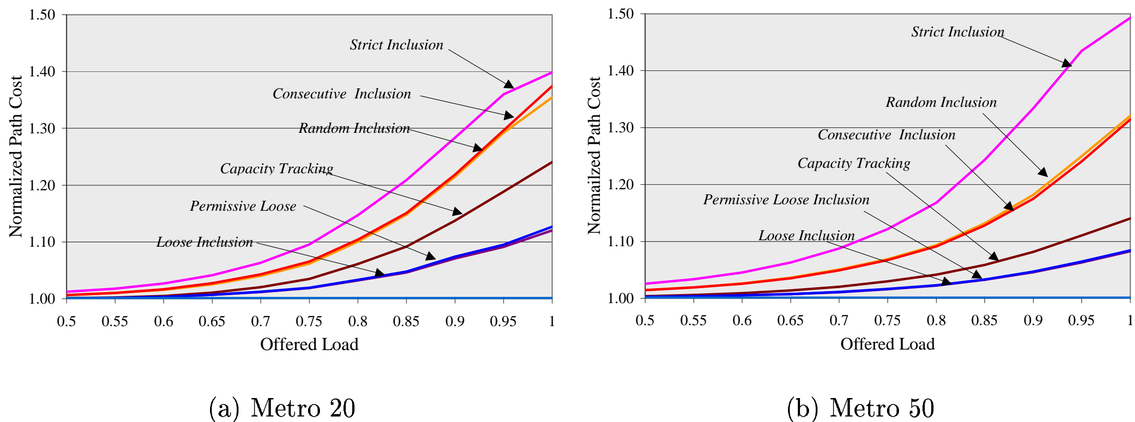
For the more realistic Metro 20 and Metro 50 networks, blocking probabilities are generally higher. For the Metro 20, we note that many sessions must take “detours” to pass through processing nodes. For example, consider sessions between Pittsburgh and DC or Seattle and Minneapolis. When the default path is too busy to accommodate sessions, the “second-choice” paths typically require even longer detours. With the Torus, on the other hand, the second and third choices are often no worse than the default. For the Metro 50, the detours required to reach processing nodes are generally smaller, but the number of hops required between endpoints tends

to be larger; for example, there are 11 hops in the shortest path from New York to Los Angeles. Note that with capacity tracking blocking probabilities of less than 1% are obtained for offered loads of more than 75%.



(a) Torus (b) Random

Figure 3.11: Configuration Cost

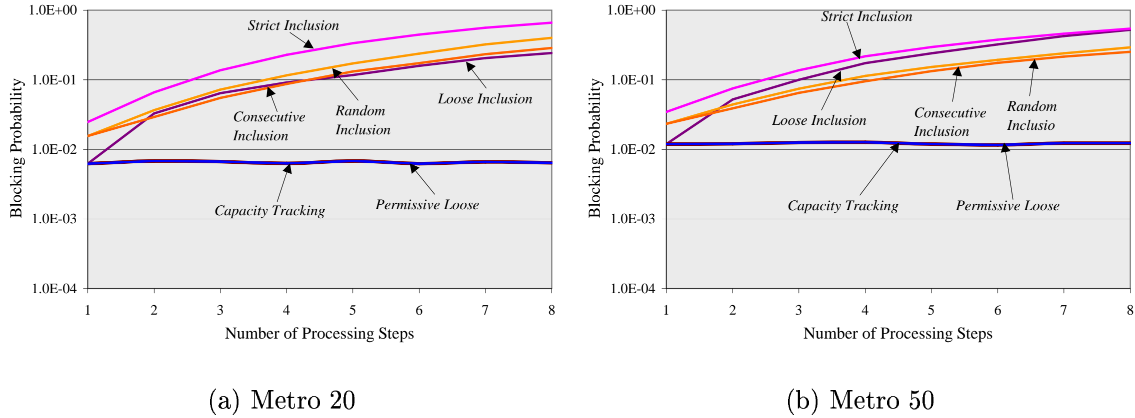


(a) Metro 20 (b) Metro 50

Figure 3.12: Configuration Costs of the metroarea networks

We also measured the cost of the successful configurations. In Figures 3.11 and 3.12, we show the configuration cost from all heuristics relative to the cost of the default shortest path, which is a lower bound. All heuristics provide nearly optimal costs at low loads, but deviate significantly at higher loads. The paths produced using capacity tracking generally stay within 5 to 10% of the lower bound up to loads

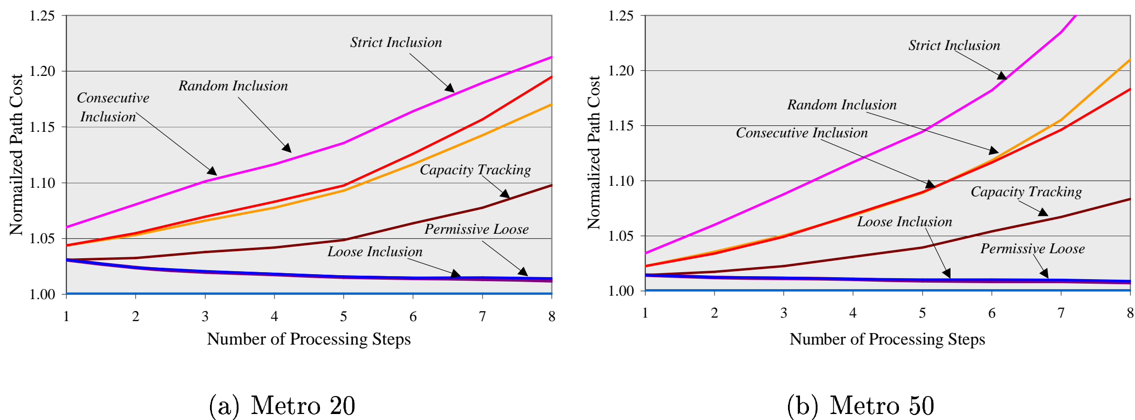
of 95%. For the Metro 20 network, the cost rises to about 20% more than the lower bound at a load of 95%.



(a) Metro 20 (b) Metro 50

Figure 3.13: Blocking rates at 75% traffic load

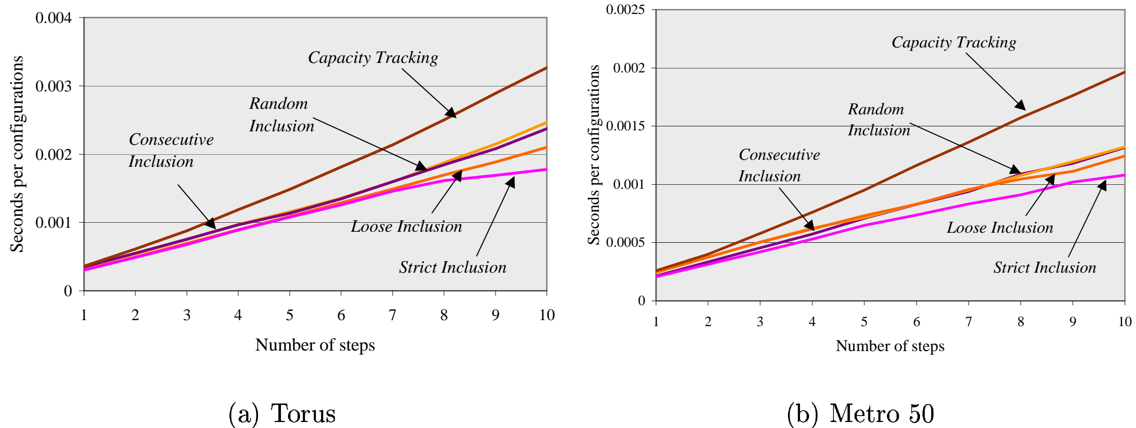
In another set of simulations, we varied the number of processing steps while fixing the offered load at 75%. Figure 3.13 shows the effect of this on blocking probability for the Metro 20 and Metro 50. As we increase the number of processing steps, the heuristics based on selective inclusion have more sessions blocked due to incorrect heuristic choices in selecting layers, while capacity tracking experiences no increase



(a) Metro 20 (b) Metro 50

Figure 3.14: Configuration cost at 75% traffic load

in blocking. Figure 3.14 shows the effect of increasing the number of processing steps on the path quality.



(a) Torus (b) Metro 50

Figure 3.15: Time requirements for session configurations

Lastly, we measured the average time required for session configuration by the different algorithms. Figure 3.15 shows the results for the Torus and Metro 50. For all algorithms, we varied the number of steps from 1 to 10. As can be seen, the algorithms based on selective link inclusions are the fastest. On the other hand, capacity tracking remains reasonably competitive, with a computational cost less than twice that of the best selective inclusion algorithm when ten processing steps are performed. Considering that sessions are likely to have far fewer than 10 steps in the vast majority of applications, the superior blocking probability achieved with capacity tracking more than compensates for the extra computational time.

Chapter 4

Designing Extensible Networks

A well designed network is crucial to delivering performance guarantees. This chapter discusses the problem of designing extensible networks, with the goal of finding a least-cost network in which resources can always accommodate anticipated traffic load. Designing networks is a well-known complex problem that has been studied extensively for conventional networks, where application-specific processing is limited only to end systems. Extensible networks, as discussed in Chapters 2 and 3, add more dynamics to operating networks by allowing processing customized for individual applications, not only at end systems but also at intermediate network nodes. Supporting customized processing at intermediate network nodes opens up a new, flexible way to provide advanced services to individual applications. Traffic streams from each application can be handled uniquely at the network nodes, as specified by the application. Designing such extensible networks is, however, complicated by this customized processing. First, adequate processing resources must be provided at network nodes to support applications that require customized processing. Second, link resources must be provided in consideration of customized processing, because such processing, when applied to traffic streams at intermediate nodes, can alter the link bandwidth initially configured by applications. Our work extends and generalizes constraint-based network design methods developed for conventional networks [20] to address these issues in designing extensible networks.

4.1 Constraint-based Network Design

A good deal of research effort has been put into designing networks in order to accommodate anticipated traffic efficiently and effectively. Designing a network starts with describing the traffic anticipated in the network. In an earlier form of telecommunication network design, traffic was described exclusively by a traffic matrix [33], which tightly specified traffic between every pair of nodes in the network. Traffic matrices were used in designing connection-oriented networks, such as telephone networks, in which application sessions were predictable and uniform enough to be described by probability models. The goal of designing such networks was to achieve cost effectiveness while obtaining a low probability for session call blocking. Traffic matrices continued to be used in designing packet-switched networks, which focused on bounding the delay for packet transmissions or achieving efficient resource usage while maintaining cost-effectiveness.

In another form, the traffic description is given by the traffic load expected at each terminal node, such as the total traffic load that can terminate at a node (ingress constraint), or the total traffic load that can be initiated at a node (egress constraint). This traffic description has been used in access network design [34, 35], where terminals must subscribe to a core network to connect to any other terminals. The goal here is to find an efficient access network which connects all terminals to the core network while satisfying the worse-case traffic expectation. Similar constraints were also adopted later in the hose model [18] by Duffield, et al. in designing efficient virtual private networks that satisfy given traffic constraints. By using ingress/egress constraints rather than a traffic matrix, the hose model expressed correlation and aggregation among traffic flows that are associated with different sources or destinations. The hose model allows network designers to give only a loose specification of the traffic, when there is not enough certainty to allow specification of a traffic matrix.

Fingerhut [20] introduced a more general concept of traffic constraints in constraint-based network design. This concept includes not only the existing cases such as ingress/egress constraints or pairwise constraints as expressed in traffic matrices, but also more general cases where traffic can be constrained between any two sets of nodes. In our extensible network design, we adopt and generalize Fingerhut's methods for constraint-based network design.

The constraint-based design of networks starts with a set of network locations and a set of traffic constraints. A traffic constraint is given as an upper bound for the traffic that flows from one set of locations to another set of locations. The objective of the design problem is to find a least-cost network configuration in which the link capacities ensure that any traffic configuration allowed by the constraints can be handled. Here, the cost of a network configuration is defined as the sum of the costs associated with each resource, where the cost of each resource is linearly proportional to its capacity. This linear dependency of cost to capacity is appropriate for large backbone networks, where the bandwidth required between adjacent routers may be a multiple of the largest physical link bandwidth. It is less appropriate in smaller networks, which use a variety of physical link bandwidths, and where the cost per unit bandwidth can vary significantly. Other network design approaches do consider non-linear cost functions [34, 35, 36]. The constraint-based design method can be generalized to include non-linear cost models, at the cost of more complex algorithms. In this work, we limit ourselves to linear cost functions for both link and processing resources.

Link capacities, which directly affect the cost of a network, are highly dependent on the routing policy, the principle that determines how to route traffic. Routing policies control the usage of link resources by specifying how to compute end-to-end paths to be assigned to application traffic. Fingerhut [20] showed how to find the required link capacities, given a set of traffic constraints, a network topology, and a deterministic routing policy. Note that in a deterministic routing policy, a pre-determined path is used for routing traffic between two end points, regardless of

traffic load in the network. In practice, more flexible routing policies can be used in an attempt to adapt to changing situations and to achieve better performance. For example, when the default path lacks the bandwidth required by a given flow, the routing policy might provide an alternate path to route the flow without blocking. The deterministic routes considered in designing networks can be viewed as the preferred routes used by the network in the absence of competing traffic.

In the constraint-based design of conventional networks, given a set of traffic constraints and a routing policy, we must first determine the network topology. Then we determine which links of the topology to use for each traffic flow and how much capacity to assign to each link. While there are no efficient algorithms for determining the best topology for an arbitrary set of traffic constraints, for certain classes of traffic constraints, a good deal is known. For example, in networks where the traffic between pairs of end points is tightly constrained in a traffic matrix, the complete graph is the optimal topology. For networks where the only constraints are on the traffic originating from or terminating at a network node, as in egress or ingress constraints, the best star topology is provably no more than twice as expensive as the optimal topology [20] and is usually very close to a computed lower bound. In some situations, network designers prefer more restricted topologies over the optimal topology. For example, to make the routing process at network nodes simpler and more scalable, tree topologies are often required in designing virtual private networks [37]. Gupta et al. studied the network design problem with such topological restrictions, given egress and ingress constraints. They proposed an efficient polynomial-time algorithm for finding the best tree topology [21] when the constraints at each node are symmetric; in other words, the worst-case traffic coming in or going out of each node is the same. They also showed that determining the optimal topology is a hard problem for other cases. For those hard cases, Gupta et al. presented approximation algorithms [21] that utilize linear programming and a rounding technique proposed in [38].

The general problem of finding the optimal (least-cost) topology is, as implied earlier, a hard problem, given an arbitrary set of traffic constraints [20] which can

consist of worst-case traffic limits between any two sets of network nodes. Fortunately, Fingerhut proposed a method to compute a lower bound on the cost of any topology, given a set of constraints and a routing policy [20]. In the absence of an efficient algorithm that solves the general problem, this method makes it possible to evaluate any topology, by computing link capacities, and quickly comparing the cost of the design to the lower bound to determine how good it is. In this chapter, we show how to extend Fingerhut's methods to extensible networks. Before stating the network design problem formally, in the next section we give an informal introduction to the constraint-based design of extensible networks.

4.2 Introduction to Designing Extensible Networks

Suppose you have just been assigned the task of designing a new network for the Acme Corporation. Acme has offices in New York, Philadelphia, Cleveland, Detroit, and Indianapolis and wants a network that will link these locations and support the expected traffic among the different sites. There is one catch, however. No one in the company has any definite idea how much traffic there will be. After consulting with various "experts" at Acme, all you can conclude is that there will be at most 500 Mb/s of traffic entering and leaving the New York and Philadelphia locations, 300 Mb/s at Cleveland and Detroit, and 200 Mb/s at Indianapolis. Using this information, you want to find a network configuration that will provide enough capacity to handle any traffic pattern that does not violate these constraints on the total traffic entering and leaving each location, while providing the lowest overall cost.

Fortunately for you, the company has purchased the new Extensible Network Planner(XNP) software package that can assist you in finding the best configuration, so you start up the software and specify the locations of your five sites and the constraints on the total traffic at each site. You then specify the trial topology shown in the screen shot in Figure 4.1 and ask XNP to determine how much capacity is required on each of the links to satisfy the given traffic demands. The screen shot

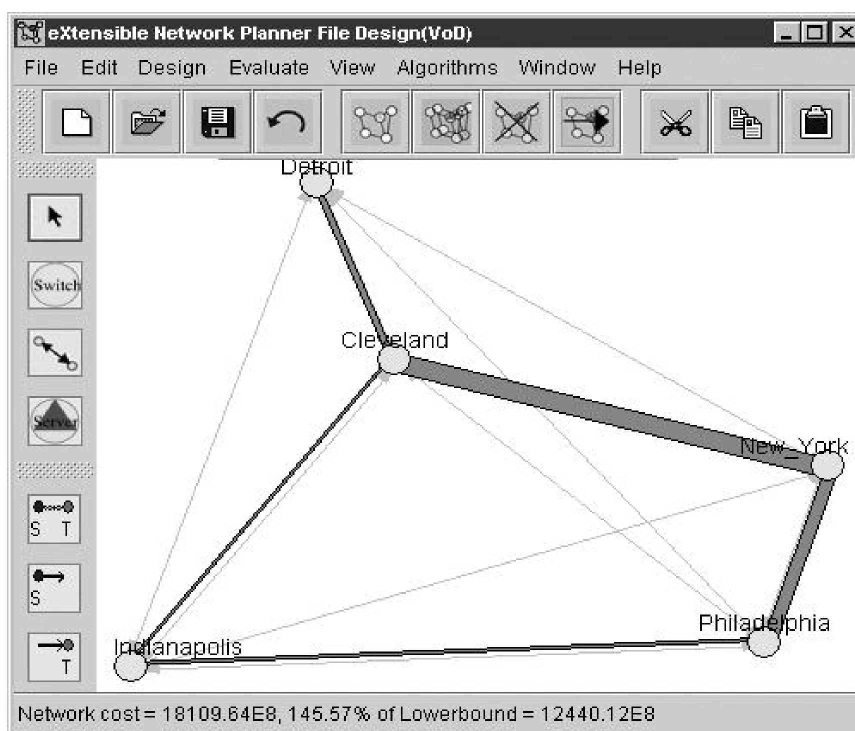


Figure 4.1: XNP snapshot for Acme corporation network

also shows, in dotted lines, all the links which you wish to consider as candidates for inclusion in the trial topology. (In this example, all links are candidates, but in more complex situations, a designer may want to limit the "design space" to a more restricted subset.)

XNP then executes one of its *link dimensioning algorithms* and assigns the resulting capacities to each of the given links. To determine the required link capacities, XNP must know how the network will route the traffic. By default, XNP assumes that traffic between any two locations always uses the least-cost path in the given network topology, where the cost of a path is the sum of the geographic distances spanned by the links in the path.

Table 4.1 shows the link capacities that are needed to accommodate any traffic pattern allowed by the constraints. For example, the link (Cleveland, New York) is assigned 800 Mb/s to accommodate the worst-case traffic, which includes 300 Mb/s

traffic from Detroit to New York, 300 Mb/s from Cleveland to Philadelphia, and 200 Mb/s from Indianapolis to Philadelphia.

Table 4.1: Resulting Capacities after Dimensioning

Link	Capacity
(Cleveland, New York)	800 Mb/s
(New York, Cleveland)	800 Mb/s
(Philadelphia, New York)	500 Mb/s
(New York, Philadelphia)	500 Mb/s
(Detroit, Cleveland)	300 Mb/s
(Cleveland, Detroit)	300 Mb/s
(Indianapolis, Cleveland)	200 Mb/s
(Cleveland, Indianapolis)	200 Mb/s
(Indianapolis, Philadelphia)	200 Mb/s
(Philadelphia, Indianapolis)	200 Mb/s

In Figure 4.1, the thickness of each link indicates its capacity as computed by XNP relative to others. XNP also calculates the total cost of the configuration. The default cost of a link is the product of its bandwidth in Mb/s and its length in miles. (The designer may specify a different cost per unit bandwidth for a link, in situations where the distance-proportional costs are inappropriate.) XNP also provides a lower bound on the cost of the best possible network for the given set of traffic constraints, so that you can estimate how close your network configuration is to an optimal configuration. (The lower bound is computed relative to the set of candidate links, so when all links are allowed, it is a lower bound on the cost of any network topology.) The network cost and the lower bound are shown at the bottom of the frame in Figure 4.1. The given network has a cost that is about 1.45 times the lower bound. You can refine the current network design by adding or removing links from the topology to seek a lower cost. Figure 4.2 shows a snapshot of a topology whose cost is about 1.03 times the lower bound.

Now, suppose that the company has decided to deploy extensible routers at its Philadelphia and Cleveland locations. Fortunately for you, the latest release of XNP also includes features to support extensible network design, so you designate

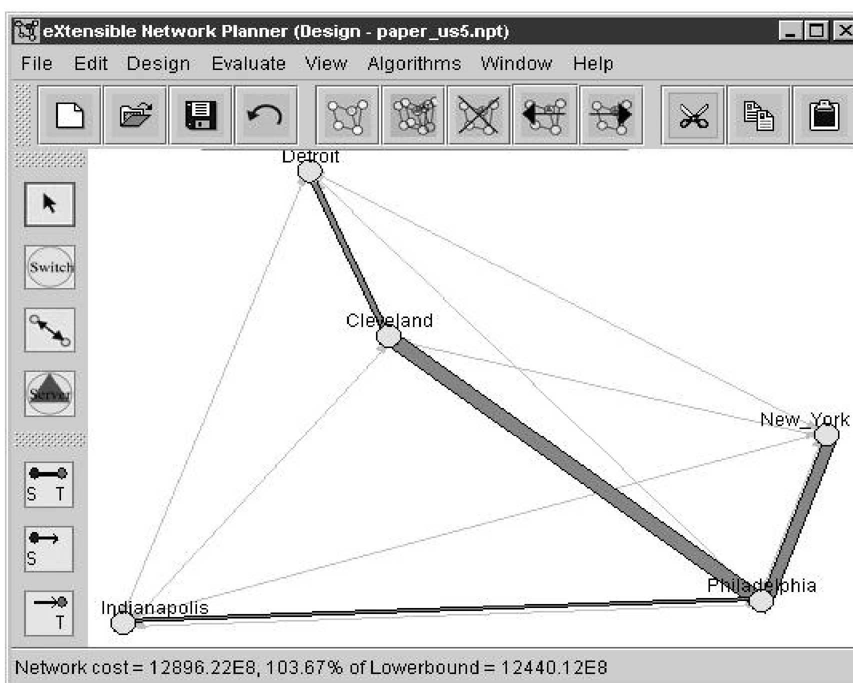


Figure 4.2: XNP snapshot with a lower cost design

the nodes at Philadelphia and Cleveland as extensible, causing XNP to highlight them as shown in Figure 4.3. The extensible routers at Philadelphia and Cleveland are to be used to support a video-on-demand application for employee education and corporate announcements. The video data is kept in compressed form on a server in the New York location and is to be decompressed by plugins running in the extensible routers. The network will automatically determine the best site to do the decompression, generally seeking to perform the decompression as close to the destination as possible. The compression algorithm reduces the bandwidth needed for the video data by a factor of 10, and the decompression plugin executes an average of 200 instructions for every byte of compressed video that it receives. For any traffic pattern that satisfies the same traffic constraints as in the previous situation, only 10% of the outgoing traffic from the New York site is expected to be compressed video traffic, and at most 50% of the traffic reaching any of the receiving sites is expected to be decompressed video traffic. Given this new information, you are asked to determine how much processing capacity is needed at each of the two sites and

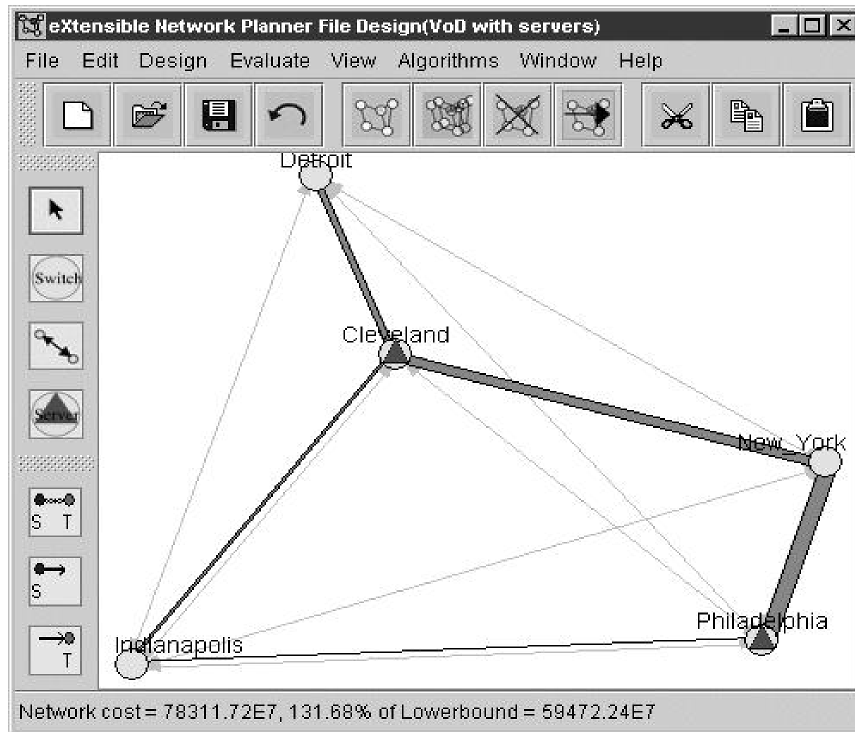


Figure 4.3: XNP snapshot for video-on-demand application

how much bandwidth is needed at each link, in order to carry the traffic for both the video-on-demand application and other applications that do not require intermediate processing.

To apply XNP to this new design problem, you must first describe the application characteristics to XNP. You define an *application format*, which specifies the amount of processing required by the intermediate processing step and the effect of the processing on the bandwidth of the data stream. In addition, you introduce new traffic constraints that bound the amount of video traffic that will be sent from the New York location and received at all locations. Given this new information, XNP determines the best way to route the video information for each destination. In particular, it determines that Philadelphia is the best place to perform decompression for users in Philadelphia and New York, and that Cleveland is the best place to perform decompression for users at the other sites. Based on this, it determines that to handle the peak video decompression traffic the extensible router at Cleveland must be able

to process 1 billion instructions per second and that the router at Philadelphia must be able to process 1.25 billion instructions per second. XNP also computes the new link capacities shown in Table Figure 4.2. Note that because the video application is asymmetric, the computed link capacities may also be asymmetric, even though in most practical situations, links are constrained to have the same capacity in both directions.

Table 4.2: Resource Capacities for Multiple Applications

Resource (Link or Processing node)	Capacity
(Cleveland, New_York)	800 Mb/s
(New_York, Cleveland)	800 Mb/s
(New_York, Philadelphia)	525 Mb/s
(Philadelphia, New_York)	750 Mb/s
(Cleveland, Detroit)	300 Mb/s
(Detroit, Cleveland)	300 Mb/s
(Cleveland, Indianapolis)	200 Mb/s
(Indianapolis, Cleveland)	200 Mb/s
(Philadelphia, Indianapolis)	200 Mb/s
(Indianapolis, Philadelphia)	200 Mb/s
Philadelphia	1.25 Gi/s
Cleveland	1 Gi/s

Using XNP, you have now designed networks for two varying situations and determined the capacity required at each individual resource. XNP was also helpful in choosing a desirable design, as it computed the lower bound on the network cost in each situation and provided a measure to compare the cost of each network design to the lower bound. In the next section, we introduce the formal basis for the constraint-based design methods developed in XNP. (We actually implemented XNP. The details are given in 5.)

4.3 Designing Extensible Networks

This section shows how to generalize constraint-based design methods, which were originally developed for conventional networks, for extensible networks. Several issues must be addressed in this generalization. First, and most fundamentally, we need a method for describing the resource usage of different applications or application classes, because extensible networks, unlike conventional networks, allow applications to involve customized processing at intermediate nodes in their sessions, and so require varying resource capacities along the session routes. We do this by defining *application formats* that express bandwidth and processing requirements associated with different components of an application session. Second, in addition to selecting a topology, we need to select network sites which will handle the intermediate processing. Third, we need to dimension not only the network links, but also the processing capacity, to ensure that all traffic demands can be met.

To incorporate these new issues, we redefine the constraint-based network design problem, using a set of application formats to be supported in the resulting network and a set of traffic constraints associated with the application formats. The objective is to find a least-cost network configuration where resources, including links and processing sites, have enough capacity to accommodate any application traffic pattern that satisfies the given traffic constraints. In the absence of efficient algorithms for finding the optimal topology, we take a pragmatic approach to the problem by providing tools that allow a network planner to quickly generate and evaluate different network topologies and processing sites. These tools automatically determine the link capacities and processing resources needed, as well as the overall cost. To assist a network planner in evaluating a candidate design, they also compute a lower bound on the cost of the best network for the given application formats and traffic constraints. To describe the design process for extensible networks, we start with some definitions.

4.3.1 Application Format

An *application format* describes characteristics of an application or class of applications that can involve an arbitrary number of processing tasks, or steps, to be performed sequentially on individual session routes. It consists of a pair of lists

$$[B = (b_0, b_1, ..., b_k), P = (p_1, p_2, ..., p_k)],$$

where b_i is the bandwidth needed on the path segment following processing step i (step 0 is the source) and p_i is the processing capacity (in instructions per second) for processing step i. For example, an application format for the video-on-demand application discussed in the previous section might be [(1 Mb/s,10 Mb/s),(25 Mi/s)]. For a slightly more complicated version of the video-on-demand application, which requires compression to be done at an intermediate node instead of at the source, the application format is [(10 Mb/s, 1 Mb/s,10 Mb/s),(20 Mi/s, 25 Mi/s)], provided that the compression requires 160 machine instructions per byte. For such a format f, let $|f|$ be the number of processing steps, let $f.b_i$ be element i of the first list, and let $f.p_i$ be element i of the second list. The above format is appropriate for one-way applications. We can extend the format for applications that involve symmetric communication in both directions, but to simplify the exposition, we omit this extension. We note that application formats can represent whole classes of applications that have similar characteristics. In particular, the relative magnitudes of the bandwidths and processing capacities are all that really matter for planning purposes. For example, the format [(1 b/s),()] can represent any application that does not require intermediate processing.

4.3.2 Traffic Constraints

The form of a traffic constraint should accommodate common cases, such as the *ingress and egress constraints* or the pairwise constraints introduced earlier in this chapter, as well as more complex constraints. The most general form for a traffic

constraint is a bound on the traffic passing between any two sets of nodes, using a specified subset of the application formats. To allow for formats that alter the bandwidth of a session, we define traffic constraints to be tuples of the form $(S, T, F, \alpha, \omega)$, where S and T are sets of nodes, and α and ω are non-negative numbers that limit the simultaneous traffic using application formats in a set of formats F, originating at nodes in S and terminating at nodes in T.

4.3.3 Traffic Configuration

A *traffic configuration* is a matrix $C = [c_{s,t,f}]$. The matrix entry $c_{s,t,f}$ specifies the number of sessions from a node s going to a node t using application format f. A traffic configuration C is *allowed* by a set M of traffic constraints if for all $(S_i, T_i, F_i, \alpha_i, \omega_i) \in M$,

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_0 c_{s,t,f} \leq \alpha_i$$

and

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_{|f|} c_{s,t,f} \leq \omega_i.$$

In the above conditions, for each constraint $(S_i, T_i, F_i, \alpha_i, \omega_i)$, we limit the total traffic originating at nodes in S_i, summed over all application formats f in F_i, all sources in S_i, and all destinations in T_i. Similarly, we limit the total traffic terminating at nodes in T_i, summed over all application formats f in F_i, all sources in S_i, and all destinations in T_i.

4.3.4 Routing Policy

A *routing policy* is a function $R(s, t, f)$ that specifies the path used for traffic originating at s, terminating at t, and using application format f. Specifically,

$$R(s, t, f) = [(u_1, u_2, ..., u_r), (u_{v_1}, u_{v_2}, ..., u_{v_{|f|}})],$$

where $(u_1, ..., u_r)$ is a path from s to t and $(u_{v_1}, ..., u_{v_{|f|}})$ is a sublist of $(u_1, ..., u_r)$, possibly with some nodes repeated, where $v_i \leq v_{i+1}$ and $1 \leq v_i \leq r$, and where every node in the second list is a processing node. For example, in the video-on-demand application of the previous section,

$$R(\text{New York}, \text{Detroit}) = [(\text{New York, Cleveland, Detroit}), (\text{Cleveland})].$$

This is interpreted as a path from New York to Detroit via Cleveland which is designated for processing. Note that the routing policy used in an actual network will typically be more flexible than the fixed routing policy assumed for network planning purposes. However, the routes used in network planning should correspond to the preferred routes in the real network. In a properly dimensioned network, the traffic will use these routes so long as no traffic constraints are violated. However, deviations from the planned routes may well happen if the network traffic exceeds the constraints; this occurs in real networks, despite the best efforts of network planners.

4.3.5 Load Factor

The *load factor* $w_{\ell,s,t,f}$ is the amount of bandwidth used on link ℓ by a session joining source s and destination t using application format f. The definition is complicated by the fact that a given link may be used more than once by a single application data stream that involves intermediate processing steps. Given a routing policy R that assigns a route

$$R(s,t,f) = [(u_1, u_2, ..., u_r), (u_{v_1}, u_{v_2}, ..., u_{v_{|f|}})]$$

to the pair of nodes s, t and the application format f, we define

$$w_{\ell,s,t,f} = \sum_{\substack{0 \leq i \leq |f| \\ \exists (u_j, u_{j+1}) = \ell \,\wedge\, v_i \leq j < v_{i+1}}} f.b_i,$$

where $v_0 = 1$ and $v_{|f|+1} = r$. This can be interpreted as the sum of the bandwidth required at link l by the path segments of the route from s to t for application format f. Similarly, we define the load factor for a processing node q as

$$w_{q,s,t,f} = \sum_{\substack{1 \le i \le |f| \\ \ni v_i = q}} f.p_i,$$

which is the sum of the capacities required by the processing nodes used by the route from s to t for application format f. We note that while the load factors are clearly dependent on the routing policy, we do not show this explicitly in our notation.

4.3.6 The Resource Dimensioning Problem

We can now define the resource dimensioning problem for a link ℓ. Given a set of application formats F, traffic constraints M, and a routing policy R, find a traffic configuration $C = [c_{s,t,f}]$ that maximizes

$$\lambda_\ell(C) = \sum_{s,t \in V} \sum_{f \in F} w_{\ell,s,t,f} c_{s,t,f}$$

while respecting the following inequalities:

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_0 c_{s,t,f} \le \alpha_i \tag{4.1}$$

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_{|f|} c_{s,t,f} \le \omega_i \tag{4.2}$$

for all $(S_i, T_i, F_i, \alpha_i, \omega_i) \in M$.

We can state it more succinctly as: find a C allowed by M that maximizes $\lambda_\ell(C)$. $\lambda_\ell(C)$ is the bandwidth needed at link l, given a traffic configuration C. Therefore, by finding the maximum possible value of $\lambda_\ell(C)$, we find the necessary and sufficient bandwidth for link l. Similarly, we can define the dimensioning problem for a processing site q as: find a C allowed by M that maximizes

$\lambda_q(C) = \sum_{s,t \in V} \sum_{f \in F} w_{q,s,t,f} c_{s,t,f}$. Stated in this way, it is clear that the resource dimensioning problem is a linear programming problem and can be solved effectively using an efficient LP solver.

4.3.7 Least-cost Routing Policy

This section elaborates further on routing policies. For a given network topology and set of processing sites, there is a basic routing policy that seeks to minimize the cost of the resources used by each session. This routing policy assigns a least-cost path to each tuple (s, t, f) comprising a source s, sink t, and application format f. Before we can determine a least cost path, we must first define what we mean by the cost of each resource. We assign a cost per unit bandwidth to each link and a cost per unit processing capacity to each processing node. These must be expressed in directly comparable units (e.g., dollars). The cost of using a given path for a given application session is the sum of the costs incurred at each link and processing node, taking into account the amount of bandwidth required on each link and the processing capacity used at each processing node. To find the best route from s to t for application format f, we must find a path from s to t with an appropriate set of processing nodes that minimizes the combined costs of the links and the nodes. To illustrate the issues that arise in this problem, consider a video application that, for efficient transmission, compresses the video data sent from a source at an extensible router and decompresses the compressed data before it reaches the destination. The preferred route would minimize the uncompressed data transmission in the network by selecting a router near the source to host the compression plugin and a router near the sink to host the decompression plugin.

We discussed this problem in Chapter 2 and 3, and presented an efficient method called the *layered network* method to address it. The *layered network* method solves the routing problem in extensible networks by reformulating the problem in another space, where it can be solved as a conventional shortest path problem. The resulting shortest path can then be mapped back to a route configured with links

and processing nodes in the original network. We briefly revisit the *layered network* method by showing how it computes the route from Detroit (Det) to New York (NY) for our example f =[(1 Mb/s,10 Mb/s),(25 Mi/sec)] in Figure 4.3, where two processing nodes are given at Cleveland (Cle) and Philadelphia (Phili) locations. Specifically, we focus on the transformation that converts the network to a new graph called the "layered network", which is shown in Figure 4.4. This layered network G'

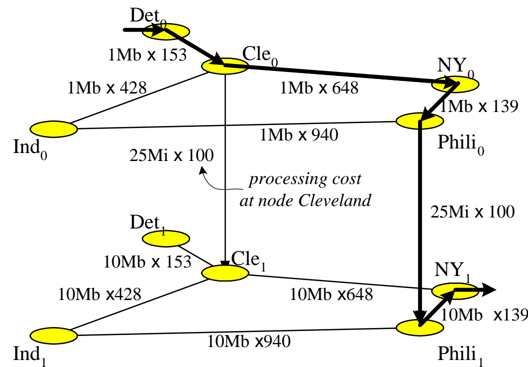


Figure 4.4: Layered network with least-cost path

includes two copies of the original network $G = (V, E)$. We refer to one copy as layer 0 and the other copy as layer 1. Also, for each node v in G, we denote the copy of the node in layer 0 as v_0 and the copy in layer 1 as v_1. We complete the layered network G' by adding an inter-layer link (r_0, r_1) for each processing node $r \in$ (Cle, Phili). For the purpose of routing, inter-layer links are assigned a cost equal to the cost of performing the decompression operation at that node. In other words, the cost of a inter-layer link is the product of the processing capacity required for the decompression algorithm and cost per unit processing capacity of the corresponding processing node. The links in the top layer are assigned a cost equal to the cost of carrying a compressed video stream (that is, the bandwidth of the compressed video stream times the cost per unit bandwidth of the link). The links in the bottom layer are assigned a cost equal to the cost of carrying a decompressed stream. Hence, the routing algorithm treats each link in the lower layer as being ten times more expensive than the corresponding link in the upper layer.

Given the layered network G', we compute the least cost path from the node Det_0, (the copy of Det in layer 0), to the node NY_1, (the copy of the destination node NY in layer 1). Note that G' only has link costs, so shortest path algorithms can be applied directly. Figure 4.4 shows the least cost path in the layered network. For the final solution to the problem, the least cost path in G' is mapped back to the original network G. For each regular link involved in the path, we "project" it to the original copy in G. Similarly for the inter-layer link in the path, we "project" it to the original processing node in G and mark it as the designated node for the requested processing. Figure 4.5 shows the projection. This projection yields the least-cost path from Det

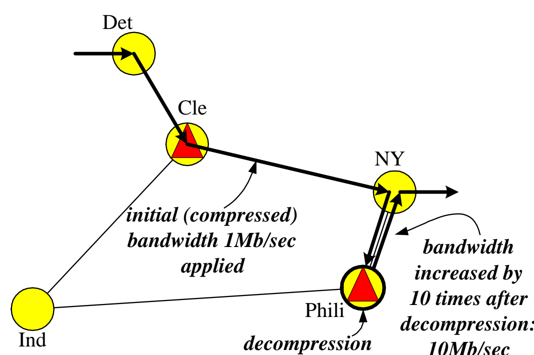


Figure 4.5: Least-cost route from Det to NY

to NY that includes an end-to-end path and a processing node. This configuration has the least cost among all such configurations, and therefore is the final solution. The *layered network* method can be generalized for an arbitrary number of processing steps, as detailed in Chapter 2.

Although other routing policies are certainly possible, we generally assume this *Least Cost Routing Policy*, for the purpose of network planning.

4.4 Resource Dimensioning using Flow Graphs

In the previous section, we showed that the problem of dimensioning resources in extensible networks could be formulated as a linear programming problem, making it amenable to solution using standard linear programming solvers. While this is a

viable approach, solving large linear programs can be very time-consuming, limiting its appeal in the context of an interactive network design tool. In this section, we introduce a different method for resource dimensioning, based on finding maximum flows, which can be significantly faster. While these flow based methods are not universally applicable, they do apply in some important common cases. Specifically, the flow based methods can be applied whenever every traffic constraint $(S, T, F, \alpha, \omega)$ satisfies these conditions:

$$S \text{ is a singleton and } T = V, \text{ or} \tag{4.3}$$

$$T \text{ is a singleton and } S = V, \text{ or} \tag{4.4}$$

$$S, T \text{ are both singletons,} \tag{4.5}$$

and there is a constant γ such that every application format f satisfies the condition

$$\frac{f.b_{|f|}}{f.b_0} = \gamma. \tag{4.6}$$

Conditions (4.3) and (4.4) describe *ingress/egress constraints*, and Condition (4.5) describes pairwise traffic constraints. Condition (4.6) gives a restriction on the bandwidth requirements such that the ratio of the terminating bandwidth requirement to the starting bandwidth requirement must be constant for all application formats considered.

Fingerhut [19] showed that the resource dimensioning problem for conventional networks can be solved under similar conditions, using the *max weight, max flow* method (equivalent to the *max cost, max flow* method), which is an extension of the well-known *max flow* problem. We extend the *max weight, max flow* method to extensible networks in this section.

The general max flow problem [32] consists of a directed graph with edges labeled with flow *capacities*, and two distinct vertices: the *source* and the *sink*. In

this graph, a flow is an assignment of a value to each edge, such that the value does not exceed the capacity of each edge, and the sum of the incoming flows equals the sum of the outgoing flows at each vertex, except at the *source* and the *sink*. The goal of the max flow problem is to find a flow in this graph which maximizes the incoming flow at the *sink*. The problem can be extended to include edge weights, leading to the *min weight, max flow* problem. Here, the weight of a flow on an edge is defined as the product of the flow and the edge weight, and the weight of a flow in the entire graph is defined as the sum of these products. In this work, we use a dual version of this problem called the *max weight, max flow* problem, in which the goal is to find a flow of maximum weight, where the weights are all non-negative.

To illustrate how the max weight, max flow method can be used for our dimensioning problem, consider the situation where a new video-on-demand application needs be introduced into the network given in Figure 4.5. In order to deploy the application properly, the network designer must find the extra capacity needed at each resource. As with the application discussed in Section 4.2, all video traffic is sent in a compressed format from New York, with a compression ratio of 10:1, and so has to be decompressed by extensible routers at the Philadelphia and Cleveland locations. The additional traffic expected for the new application is specified relative to the existing ingress/egress traffic, where the amount of ingress/egress traffic is restricted to 500 Mb/s at New York and Philadelphia, 300 Mb/s at Detroit and Cleveland, and 200 Mb/s at Indianapolis. The new video traffic is bounded so that at most 10% of the current outgoing traffic from New York is expected to be compressed video traffic, and at most an additional 50% of the current traffic to each site is expected to be decompressed video traffic. To make the problem more interesting, we also assume that at most 5% of the total traffic exchanged within the New York location is expected to be compressed video traffic. Note that compressed video from the video source in New York has to be routed through one of the extensible routers, the one in Philadelphia in this case, even when the receiver belongs to the same location, New York. Table 4.3 shows the traffic constraints corresponding to the traffic description,

Table 4.3: Traffic constraints associated with Figure 4.5 for resource dimensioning

S	T	α	ω
(NY)	V	50 Mb/s	∞ Mb/s
(Det)	V	0 Mb/s	0 Mb/s
(Cle)	V	0 Mb/s	0 Mb/s
(Ind)	V	0 Mb/s	0 Mb/s
(Phili)	V	0 Mb/s	0 Mb/s
V	(NY)	∞	250 Mb/s
V	(Det)	∞	150 Mb/s
V	(Cle)	∞	150 Mb/s
V	(Ind)	∞	100 Mb/s
V	(Phili)	∞	250 Mb/s
(NY)	(NY)	25 Mb/s	∞

NY : New York Det : Detroit
Cle : Cleveland Ind : Indianapolis
Phili : Philadelphia

where each row is a traffic constraint.

As described in the previous section, a route for each potential pair of end points, a source and a destination, can be computed by the *layered network* method. We can iteratively dimension the links and processing nodes using the route information. Let us select the link l =(Phili,NY) for example. First, we identify the pairs of

Table 4.4: Load factor of the pairs whose path is routed through (Phili, NY), relative to the initially configured session bandwidth

	NY	Det	Cle	Ind	Phili
NY	10	0	0	0	0
Det	10	0	0	0	0
Cle	10	0	0	0	0
Ind	10	0	0	0	0
Phili	10	1	1	0	0

terminal nodes whose route uses the link ℓ and compute the load factor of each pair relative to the initial session bandwidth as shown in Table 4.4. Recall that the load factor $w_{\ell,s,t,f}$ is the total bandwidth needed at ℓ to support a session configured from

s to t of the application f. The bandwidth increases by ten times after the intermediate processing, and therefore the load imposed by each session varies with respect to where it is used relative to the processing site. In the table, the cell [Ind][NY] contains the relative load factor $w_{\text{(Phili, NY),Ind,NY},f} \times \frac{1}{f.b_0} = \frac{10 \text{ Mb/s}}{1\text{Mb/s}} = 10$, because the link (Phili, NY) is used after the decompression processing at the Philadelphia site in the route from Indianapolis to New York. Meanwhile, the the relative load factor $w_{\text{(Phili, NY),Phili,Det},f} \times \frac{1}{f.b_0} = 1$, because the link (Phili, NY) is used before the processing occurs at the Cleveland site on the route from Philadelphia to Detroit.

We now proceed to find the capacity needed at the link (Phili, NY) to satisfy any traffic pattern that uses the load factors in Table 4.4 and is allowed by the constraints given in Table 4.3. We do this by configuring a flow graph so that any flow in the graph from the *source* vertex to the *sink* vertex corresponds to a valid traffic configuration that satisfies the traffic constraints. Figure 4.6 shows a flow graph, a *source* and a *sink*, and two columns of vertices. Each column contains a vertex for each node of the network in Figure 4.5. The figure includes an edge from the *source* to each vertex in the first column, and from each vertex in the second column to the *sink*. Between the columns, an edge is included from a vertex in the first column to a vertex in the second column only if the route between the terminal nodes corresponding to the vertices uses the link (Phili, NY), i.e., if the corresponding load factor in Table 4.4 has a non-zero entry. For example, the flow graph contains the edge (Det_s, NY_t) because the route between Detroit and New York uses the link (Phili, NY) with the relative load factor of 10.

We then apply the constraints in Table 4.3 to the flow bound of the edges of this graph. Let us first consider the edges from the *source* to the vertices in the first column, for instance, $(source, NY_s)$. We assume that the edge should accommodate all sessions configured from NY to any of the nodes in Figure 4.5, and assign the corresponding traffic limit to the flow bound of this edge. According to conditions

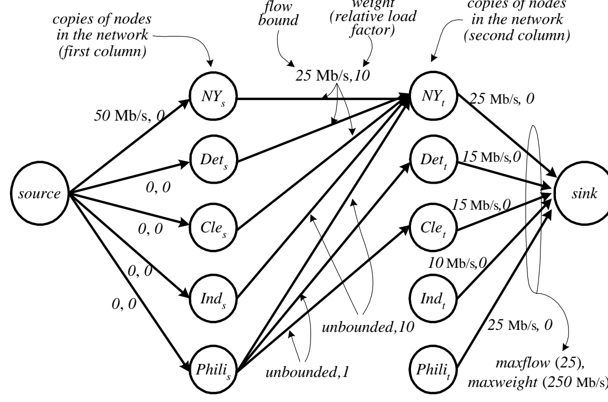


Figure 4.6: Flow graph used for dimensioning (Phili,NY)

(4.1) and (4.2) in Section 4.3, the value is given as follows:

$$\sum_{t \in V} f.b_0 c_{\mathrm{NY},t,f} \leq \min\left\{\alpha, \frac{\omega}{\gamma}\right\},$$

given ((NY), V, α, ω), where $\gamma = \frac{f.b_{|f|}}{f.b_0}$. Using the constraints in Table 4.3, we assign 50 Mb/s to $(source, NY_s)$ as shown in Figure 4.6.

Similarly, the edges from the vertices in the second column to the *sink* are assumed to accommodate the sessions configured to terminate at each node in the network. For instance, $(Cle_t, sink)$ should accommodate the sessions configured to terminate at Cle. The traffic limit in this case is computed as follows:

$$\sum_{s \in V} f.b_0 c_{s,\mathrm{Cle},f} \leq \min\left\{\infty, \frac{150 \text{ Mb/s}}{10}\right\} = 15\text{Mb/s},$$

given (V, (Cle), ∞, 150 Mb/s) in Table 4.3. We apply this value to the flow bound of $(Cle_t, sink)$.

Finally, the edges between the two columns are assigned the traffic limits between the corresponding pairs of end points. So, to (NY_s, NY_t), we assign the following traffic bound:

$$c_{\mathrm{NY,NY},f} \leq \min\left\{25 \text{ Mb/s}, \frac{\infty}{10}\right\} = 25 \text{ Mb/s},$$

given ((NY), (NY), 25, ∞) in Table 4.3. Note that the traffic patterns not specified in Table 4.3 are assumed to have an unlimited bound.

Now, any flow in this graph corresponds to a valid set of sessions which pass through the link (Phili, NY). Note that the resulting traffic is guaranteed to be allowed by the traffic constraints due to the bounds applied to the flow graph. Therefore, the maximum flow in this graph corresponds to a valid maximum traffic configuration that can be routed through the link. Considering that the sessions change their bandwidth along the routes, the actual traffic load on the link can be obtained by applying the relative load factors of the configured sessions, i.e., the bandwidth required at the link relative to the initially configured bandwidth. We apply the relative load factor of each pair of end nodes given in Table 4.4 to the weight of the corresponding edge between the two columns. For example, the load factor $w_{(\text{Phili,NY}),\text{Ind,NY},f} \times \frac{1}{f.b_0} = 10$; i.e., the cell [Ind][NY] in Table 4.4 is assigned to the weight of the edge (Ind_s, NY_t). Now, the product of the flow and the weight of this edge yields the actual load at the link (Phili, NY) imposed by the sessions configured from Ind to NY. The second value associated with each edge in Figure 4.6 is the weight. The edges adjacent to the *source* and the *sink* are assigned zero weight. Therefore, given a traffic configuration as a flow, the total weight of the flow corresponds to the total traffic load at the link (Phili, NY). By finding the maximum flow with the maximum weight, we obtain the worst-case traffic load, which is the capacity required at the link (Phili, NY).

We have just demonstrated how the *max weight, max flow* method solves a subset of the resource dimensioning problem efficiently. One of the most efficient algorithms for the *max weight, max flow* method, called "enhanced capacity scaling," is given in [39]. The time complexity of this algorithm is $O((k \log n)(n + n \log n))$, where k is the number of routes that use the link to be dimensioned and n is the number of nodes. If no route has more than h links, then the time needed to dimension all the links is $O\left(hn^3(\log n)^2\right)$. The most efficient algorithms for linear programming [40, 41, 42] have substantially higher worst-case complexities. Furthermore, the observed running time of existing LP solvers is substantially higher than the running

time of *max weight, max flow* algorithms, making these methods attractive in those situations where they can be applied.

4.5 Computing Lower Bounds on Network Cost

The methods discussed in the previous sections allow us to dimension resources in a network, given the network topology and a set of processing nodes. These methods, however, do not directly enable us to determine the best topology or set of processing nodes for a given set of traffic constraints and application formats. In this section, we show how to compute a lower bound on the cost of the best possible network for a given set of traffic constraints and application formats. This computation provides a useful benchmark for evaluating candidate designs.

The basic idea behind the lower bound method is simple and intuitive. It's based on the observation that some traffic configurations are inherently more expensive than others. In particular, traffic configurations in which most application sessions pass between sites that are geographically distant from one another are inherently more expensive than configurations in which most traffic is "local". Any network that can accommodate all configurations allowed by the given set of traffic constraints must, in particular, accommodate these more expensive traffic configurations. The idea then is to determine the *intrinsic cost of a most expensive traffic configuration* and use this as a lower bound on the cost of any network.

To define the intrinsic cost of a traffic configuration $C = [c_{s,t,f}]$, we must first define the intrinsic cost $\tau(s,t,f)$ of an application session from s to t using format f. We define $\tau(s,t,f)$ to be the cost of a least-cost route from s to t, assuming that the route can include links between any pair of nodes and that every node along the route can be a processing node. The cost of a route is the sum of the costs incurred at each link and node along the route, taking into account the bandwidth used on each link and the processing capacity used at each node. The intrinsic cost can be computed by constructing a layered graph for the given application format and finding the shortest

path from the copy of s in the first layer to the copy of t in the last layer. In this layered graph, there is a link between every pair of nodes within each level, and there are inter-layer links for every node.

Given τ, the intrinsic cost of a traffic configuration C is simply

$$\sum_{s,t \in V} \sum_{f \in F} c_{s,t,f} \tau(s,t,f).$$

We say that a traffic configuration C is the most expensive traffic configuration for a given set of traffic constraints M and application formats F if C maximizes

$$\sum_{s,t \in V} \sum_{f \in F} c_{s,t,f} \tau(s,t,f)$$

while respecting the following inequalities:

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_0 c_{s,t,f} \leq \alpha_i \tag{4.7}$$

$$\sum_{s \in S_i, t \in T_i, f \in F_i} f.b_{|f|} c_{s,t,f} \leq \omega_i \tag{4.8}$$

for all $(S_i, T_i, F_i, \alpha_i, \omega_i) \in M$. We can find the most expensive traffic configuration by solving the linear programming problem. Such a traffic configuration provides a lower bound on the cost of any network design that can handle all traffic configurations allowed by the given constraints.

In practice, a network designer may often choose to constrain the design space by excluding the possibility of direct links between some pairs of nodes and by limiting the set of nodes that are potential candidates for processing. For example, the direct link between two locations that are geometrically remote from each other might be omitted from consideration. The lower bound method can be easily extended to provide lower bounds relative to such restricted design spaces. The only thing that must be changed is the definition of $\tau(s,t,f)$. In the restricted design space, $\tau(s,t,f)$ is the cost of a least-cost route from s to t, assuming that the route can include only

those links in the design space and that processing can be done only at nodes that are designated as candidate processing nodes in the design space. $\tau(s,t,f)$ can be computed in the restricted design space by constructing a layered graph in which each layer includes only the links in the design space, and inter-layer links are provided only at the candidate processing nodes.

4.6 Computing Lower Bounds using Flow Graphs

This section presents a different way to compute the most expensive traffic configuration to obtain a lower bound, using the *max weight, max flow* method. As was shown for dimensioning resources in Section 4.4, this flow based method is applicable only when Conditions (4.9) (4.10) (4.11) and (4.12) are met.

The traffic constraint (S,T,F,α,ω) satisfies these conditions

$$S \text{ is a singleton and } T = V\text{, or} \tag{4.9}$$

$$T \text{ is a singleton and } S = V\text{, or} \tag{4.10}$$

$$S,T \text{ are both singletons,} \tag{4.11}$$

and there is a constant γ such that every application format f satisfies the condition

$$\frac{f.b_{|f|}}{f.b_0} = \gamma. \tag{4.12}$$

Finding a lower bound starts with creating a flow graph whose flow is equal to a valid traffic configuration in the complete network, which includes a direct link between every pair of nodes. The traffic configuration is considered in the design space, if that space is given, instead of the complete network. The goal is to compute the cost of the most expensive traffic configuration by finding the maximum flow that also has the maximum cost in this graph.

To illustrate this method, we take the same example as given in Section 4.4, a network composed of five locations in the northeastern United States, where a

video-on-demand application sends video traffic compressed by 10:1 from New York. The same traffic constraints are applied, where the total outgoing video traffic at New York is at most 50 Mb/s, the total incoming video traffic is at most 250 Mb/s, 150 Mb/s, 150 Mb/s, 100 Mb/s, 250 Mb/s each at New York, Detroit, Cleveland, Indianapolis, and Philadelphia. In addition, the total video traffic from New York that terminates at New York (at any terminal in the New York location) must be at most 25 Mb/s. The constraints are given in Table 4.5. Given the network and the

Table 4.5: Traffic constraints associated with Figure 4.5 for computing a lower bound

S	T	α	ω
(NY)	V	50 Mb/s	∞ Mb/s
(Det)	V	0 Mb/s	0 Mb/s
(Cle)	V	0 Mb/s	0 Mb/s
(Ind)	V	0 Mb/s	0 Mb/s
(Phili)	V	0 Mb/s	0 Mb/s
V	(NY)	∞	250 Mb/s
V	(Det)	∞	150 Mb/s
V	(Cle)	∞	150 Mb/s
V	(Ind)	∞	100 Mb/s
V	(Phili)	∞	250 Mb/s
(NY)	(NY)	25 Mb/s	∞

NY : New York Det : Detroit
Cle : Cleveland Ind : Indianapolis
Phili : Philadelphia

traffic constraints, we create a flow graph using a *source* and a *sink* and two columns of vertices. As shown Figure 4.7, each column contains a vertex for each node in the given network. This graph also includes an edge between every pair of nodes, one from the first column and the other from the second column.

In this flow graph, we assign a bound on the flow of each edge according to the traffic constraints in Table 4.5. The flow bounds are assigned so that any flow in the graph is a valid traffic configuration in the network. In Figure 4.7, the first five constraints in Table 4.5 are applied to the edges from the *source* to the vertices in the first column h. The next five constraints are applied to the edges from the vertices

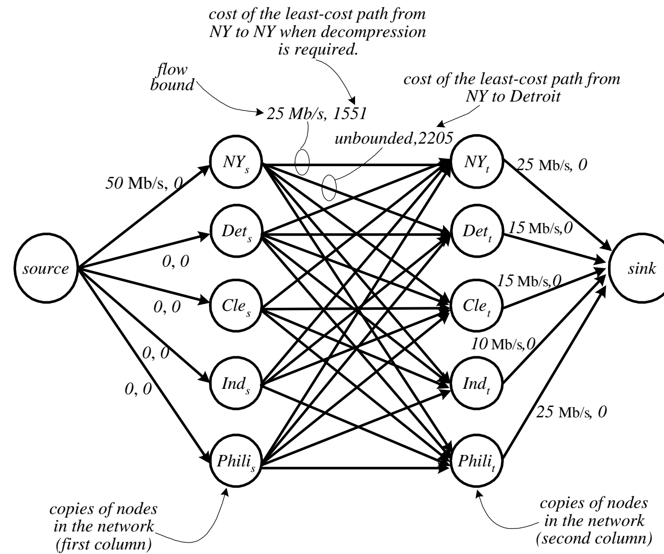


Figure 4.7: Flow graph used for computing a lower bound

in the second column to the *sink*, after considering γ, the ratio of the terminating bandwidth to the configured bandwidth. The pairwise constraint given in the last line of Table 4.5 is applied to the edge (NY_s, NY_t). Edges whose flow bounds are not specified by the constraints are assumed to have no bound on the flow. Consequently, given this specification of flow bounds, any flow in this graph is equivalent to a valid traffic configuration.

Now we apply $\tau(u, v, f)$ to the weight of each edge (u_s, v_t) between the two columns of the flow graph, where f is the application format of the video-on-demand application. Recall that $\tau(u, v, f)$ was defined in the previous section as the cost of the least-cost route from u to v using the application format f, assuming that the route can include links between any pair of nodes and that every node along the route can be a processing node. When a design space is given, the least-cost route is restricted to use only the links and the processing nodes given in the design space. The graph edges adjacent to the *source* and the *sink* are assigned 0 weight. Because of these weight assignments, the cost of any flow in the graph is now equal to the cost of configuring a valid traffic configuration. Consequently, by finding the maximum flow that also has the maximum cost, we can compute the most expensive traffic

configuration. The maximum flow with the maximum cost can be computed by the *max weight, max flow* method.

4.7 Discussion

This section briefly raises the issue of finding the optimal processing locations, which was not addressed in our earlier discussion. Finding a least-cost network configuration for conventional networks requires first determining an optimal topology that will result in the least cost and then determining the capacity required for each link included in the topology to compute the actual cost of the configuration. For extensible networks, we also need to select a set of network nodes that will handle the intermediate processing of the prospective applications, and to determine the capacity required for each of these processing nodes. Note that the required capacity of link resources and the cost associated with the resources are highly dependent on the locations of the processing nodes. Consider an application that requires intermediate processing between a source and a destination. An individual session of this application might take the shortest path between the two nodes if the path contains adequate processing site(s). Otherwise, the session must take an alternate path that provides the processing resources. The processing sites affect not only the link selection but also the bandwidth required on the selected links, because intermediate processing can alter the bandwidth of application sessions in-band by changing the data content. For both of these reasons, processing resources must be placed carefully to achieve the optimal (least-cost) network configuration. Finding the best locations for processing resources is, however, a hard problem. This processing location problem includes the K-median problem, which is known to be NP-hard [43]. Similar location problems and approximation algorithms have been studied in locating web caches [44, 45, 46], in placing servers for overlay multicast [47], and in other more general facility location problems [48, 49, 50, 38]. In spite of the hardness, it would still be worth investigating the problem of locating processing resources in extensible networks. Applications

requiring intermediate processing bear characteristics that have not been considered in the previous studies of location problems, and that can lead to effective heuristic algorithms. We leave this investigation to future work.

Chapter 5

Extensible Network Planner

This chapter describes the use of the Extensible Network Planner (XNP), a software tool for the constraint-based design of extensible networks. The conventional constraint-based network design [20] starts with a set of network nodes (for switches or routers) and a set of traffic constraints among the nodes. Its objective is to configure a least-cost network that accommodates any traffic pattern satisfying the given traffic constraints. We have generalized the constraint-based network design, originally developed for conventional networks, to handle the design extensible networks, where applications can execute processing customized for their individual goals, at intermediate network nodes. In Chapter 4, we addressed several issues regarding this generalization and presented methods that are appropriate for designing extensible networks. To provide a comprehensive network design tool for both extensible and conventional networks, we implemented the methods presented in Chapter 4 and incorporated them into a previously unpublished java-based software tool, *cappuccino* [22], developed for conventional networks.

XNP allows network designers to quickly create, configure and evaluate network designs by providing a convenient graphic-based interface and automated functions. Using XNP, network designers can generate a topology, describe prospective applications and their traffic patterns, compute the capacities of the resources in the

trial topology to accommodate the demands of those applications, and evaluate the resulting network configuration.

Section 5.1 describes how to obtain and start XNP. Section 5.2 shows by example how to use XNP to design extensible networks. Section 5.3 and 5.4 show how to file and maintain network configurations. Lastly, Section 5.5 briefly describes the algorithms implemented in XNP, that automate the design tasks that are described in Section 5.2.

5.1 Obtaining and Starting XNP

This section describes how to obtain and start XNP.

5.1.1 Installing XNP

The installation takes the following three steps:

- Install a Java virtual machine with version 1.4.x, which is required to run XNP as a java application. The Java virtual machine can be obtained by downloading JRE from **http://java.sun.com/j2se/1.4.1/download.html**.
- Download and unzip the file (**xnpt.tar.gz**) into the directory that you want to install the tool, from **http://www.arl.wustl.edu/~syc1/xnpt.tar.gz**.
- Edit the script file for XNP. For Windows, edit npt.bat so that the **JAVA_HOME** variable contains the directory where the Java VM is installed. By default, the variable is set to **c:\Program Files\java\j2re1.4.1_01**. The file, npt.bat, can be found under the tool's root directory, **$Your_Dir$\npt\npt.bat**.

5.1.2 Running XNP

Windows: run npt.bat from the tool's root directory ($Your_Dir$\npt\npt.bat)

Unix: run the shell script, $Your_Dir$/npt/npt.sh, from the tool's root directory.

Note that the current package contains a preliminary version of XNP. Some functions of the tool might not work properly. Installation and update information is available at **http://www.arl.wustl.edu/~syc1/npt.html**.

5.2 Basics of Extensible Network Planning

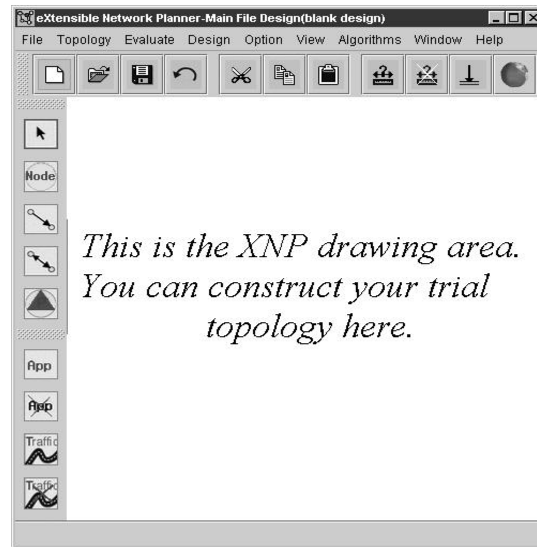
This section introduces how to use XNP to plan a network from scratch. There are five essential building blocks in planning a network in XNP. The tasks associated with the building blocks are as follows. 1) constructing a network topology in XNP by adding nodes, connecting nodes using links, or adding processing capability to nodes, 2) describing the resource usage of prospective applications, 3) specifying the anticipated traffic demands of those applications, 4) dimensioning the network topology by determining adequate capacities for individual network resources to satisfy the traffic demands of the applications, 5) and evaluating the resulting network configuration. This section will guide you through the step-by-step process of each task.

5.2.1 Constructing a network

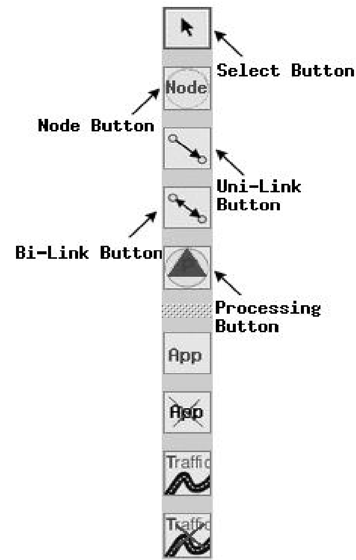
To construct a network, first start XNP as described in Section 5.1. You can then create a trial topology in the blank drawing area of the XNP main window as shown in Figure 5.1(a). There are five modes in creating topologies in the drawing area: a mode for selecting objects, a mode for adding nodes, two modes for adding links, and a mode for enabling processing at nodes. You must be in a proper mode to add, remove, or edit a network object. The buttons shown on the left side of the XNP main window can be used to change the current mode. The buttons are also shown in Figure 5.1(b).

- Adding Network Nodes

  To add a node, you first click the node button to go to the mode for adding nodes. As you do so, the node button should be highlighted. To actually add a node in



(a) XNP blank snapshot



(b) Buttons

Figure 5.1: XNP main menu and buttons

the drawing area, click on the position where you want to add the node. A pop-up window will then appear as shown in Figure 5.2(a). In the pop-up window, specify the

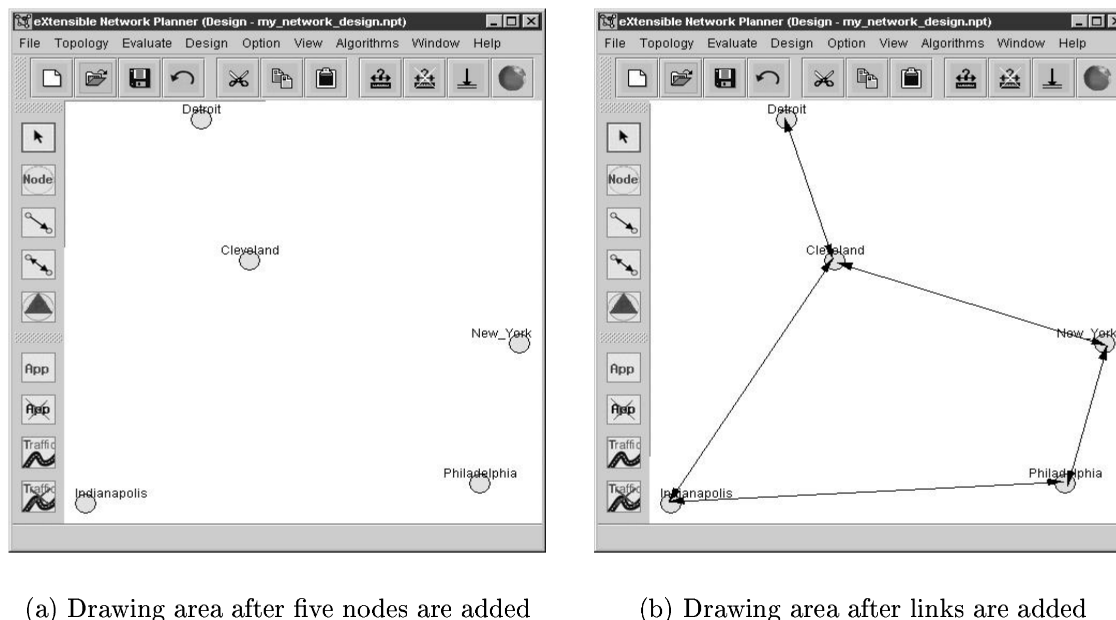
(a) Traffic expressed in b/s

(b) Traffic expressed using abbreviations

Figure 5.2: Example popup windows for adding a node

following information: the name of the node, the maximum traffic that can originate from the node, and the maximum traffic that can terminate at the node. The units of the traffic are assumed bits per second. The conventional abbreviations, i.e., K, M, and G for thousand, million, and billion, can also be used as in Figure 5.2(b).

Figure 5.3(a) shows the drawing area after nodes are added to five locations in the northeastern United States.



(a) Drawing area after five nodes are added

(b) Drawing area after links are added

Figure 5.3: XNP snap shots while generating a topology

- Adding Links

  XNP allows two types of links: bi-directional and uni-directional links. While uni-directional links can accommodate traffic in only one fixed direction, bi-directional links can do it in both directions.

  To change to the mode for adding a bi-directional link, you click the bi-link button shown in Figure 5.1(b). Then, to actually add a bi-directional link between the node u and the node v, you must click the position of the node u and then the position of the node v in the drawing area. After clicking the second node v, you provide the cost associated with this link in the pop-up window as shown in Figure 5.4. By default, XNP assumes that the cost associated with each link is linearly proportional to the geometric length of the link, and asks you to enter the cost for transmitting a bit of data on the link for one mile. You can change this setting to have a uniform cost for all links, or to assign each link a customized (user-defined) value. Once the link cost

Figure 5.4: Popup window for entering a link cost

is provided, a link with arrows on both ends will appear between the two nodes. The uni-directional link from u to v can be added in the same way, and the actual link will appear with an arrow directed toward the node v.

Figure 5.3(b) shows the drawing area after bi-directional links are added among the five nodes.

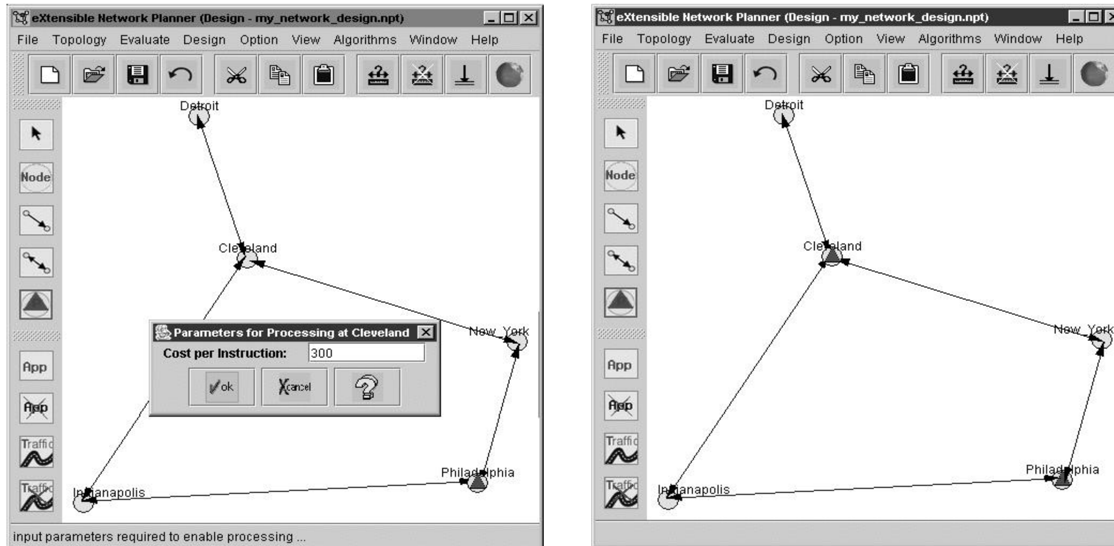
- Adding Processing Capability

  First to go to the mode for enabling processing at nodes, click the processing button. As you do so, the processing button should be highlighted. To actually enable intermediate processing at a node, first click the node and then specify the cost associated with processing, i.e., the cost for executing one machine instruction at the node. Figure 5.5(a) shows an example of the pop-up menu. As you confirm the information, the node will be highlighted by a triangle. Figure 5.5(b) shows the drawing area after processing capability is added to two nodes, Cleveland and Philadelphia.

To view or to edit more detailed information of an individual link or node, you can click on its position using the right mouse button. XNP also provides standard functions for editing the current topology such as delete, copy and paste. Figure 5.6 shows the corresponding buttons. In order to edit the topology, you must be in the mode for selecting objects. Click the select button to change to this mode.

Now, to delete an object, a node or a link, from the drawing area, click on the location of the object and then click the delete button. To copy an object, click on the location of the object and then click the copy button. To paste an object that has been copied, click on the location where you want to place the object.

Once a topology is created by adding network objects such as nodes or links, and by providing processing capability, you can now proceed to compute the capacity



(a) Popup window for placing processing capability

(b) Topology with processing nodes

Figure 5.5: XNP snap shots for placing processing capability

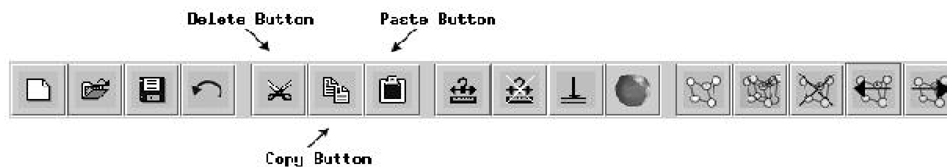


Figure 5.6: Buttons for editing network configurations

required at each of those network objects. As discussed in Chapter 4, the required resource capacity is highly dependent on the resource usage of applications and the actual traffic expected by the applications. In order to correctly determine the capacities, you must first provide XNP with this application and traffic information. The buttons shown in Figure 5.7 are used in adding and deleting such information.

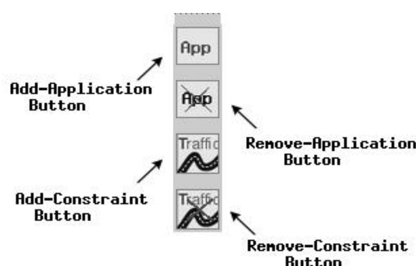


Figure 5.7: Buttons for applications and traffic constraints

5.2.2 Application formats

You can specify the resource usage of an application by adding an application format. An application format specifies the number of processing steps required by an application, the number of machine instructions required for each processing step, and the bandwidth requirement of each path segment between two consecutive processing steps.

- Adding Application Formats

  To add an application format, click the add-application button shown in Figure 5.7. Then, a pop-up window will appear as shown in Figure 5.8.

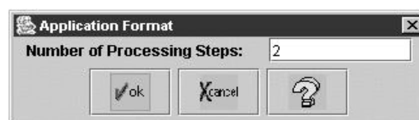
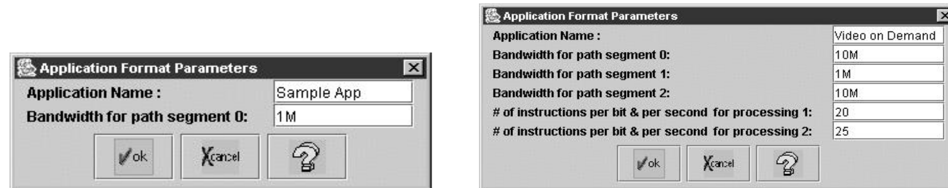


Figure 5.8: Popup window for adding an application format

If zero is entered for the number of processing steps, the application format is assumed to describe a conventional application, which does not involve any intermediate processing. For such an application, another pop-up window will appear as shown in Figure 5.9(a), where you must then provide the name of the application and the bandwidth requirement. If a non-zero value, say n, is entered for the number of processing steps in Figure 5.8, the second pop-up window will ask you to provide the name of the application, the bandwidth requirement for each of $n+1$ path segments, and the number of machine instructions required for each of the n processing steps.

Figure 5.9(b) shows an example of the pop-up window for an application with two processing steps.



(a) No processing required (b) Processing required

Figure 5.9: Popup window for describing an application format

- Removing Application Formats

  To remove an application format that has been added, click the remove-application button. You will get a pop-up menu that lists all application formats. You can choose the application formats you want to remove from this list. Figure 5.10 shows the pop-up window that displays two applications.

Figure 5.10: Popup window for removing application formats

5.2.3 Describing traffic expectations

Given a list of application formats to support in the network, you now specify the traffic expected from those applications. XNP provides three ways to specify traffic expectations:

- by an upper bound on the traffic that can flow between two selected sets of nodes,
- by an upper bound on the traffic that can flow from a selected set of nodes (sources) to any nodes in the network,

- by an upper bound on the traffic that can flow from any nodes in the network to a selected set of nodes (destinations).

  All three descriptions, which are called “traffic constraints”, provide worst-case estimates of traffic that can flow among network nodes. You also specify the applications associated with each traffic constraint.

- Adding a traffic constraint
  To add a traffic constraint, click the add-constraint button. Then, you can select

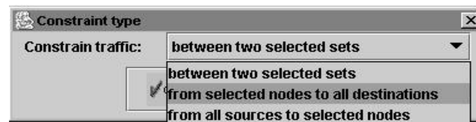


Figure 5.11: Popup window for selecting a traffic constraint type

one of the three options in a pop-up menu as shown in Figure 5.11. When an option is chosen, another pop-up window will appear requesting more information about the new traffic constraint.

If you choose the first option, you must first select a list of source nodes and a list of destination nodes, and then specify the maximum (worst-case) traffic that can originate from the sources and the maximum (worst-case) traffic that can terminate at the destinations. The maximum traffic originating from the source nodes can be given in proportion to the total outgoing traffic at the source nodes, and similarly the maximum traffic terminating at the destination nodes can be given in proportion to the total incoming traffic at the destination nodes. Note that you have already specified the worst-case outgoing and incoming traffic at the nodes when you create them earlier. The values for the maximum traffic can also be given simply in bits per seconds.

In the same menu, you can select or un-select applications to specify the applications to which this new traffic constraint will be applied. Additionally, you can add this traffic constraint into a constraint group. The default option for the constraint group

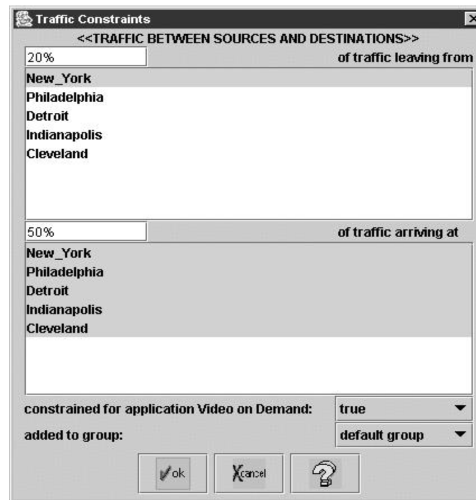


Figure 5.12: Popup window for adding traffic constraint between two sets of nodes

is the “default group”. Such grouping makes it easier for you to later display or remove traffic constraints. Figure 5.12 shows an example of the pop-up menu for the first constraint type.

If you choose the second option, you only need to select a list of nodes for the source set. The destination set will be fixed to include all nodes in the network. The remaining information can be filled in the similar way as for the first constraint type. Figure 5.13 shows an example of the pop-up menu for the second constraint type.

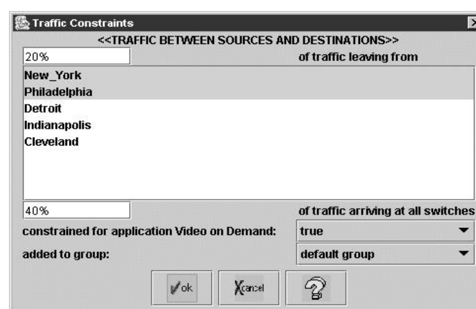


Figure 5.13: Popup window for adding traffic constraint when the destinations include all nodes

Finally, if you choose the third option, you only need to select a list of nodes for the destination set. The source set will be fixed to include all nodes in the network. Figure 5.14 shows an example of the pop-up menu for the third constraint type.

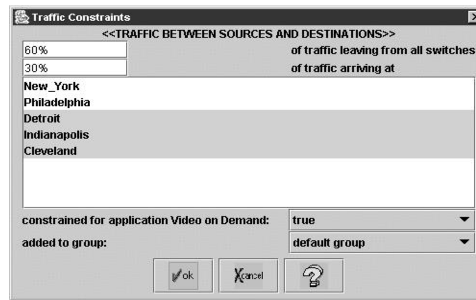


Figure 5.14: Popup window for adding traffic constraint when the sources include all nodes

- Removing traffic constraints

  There are two options for removing traffic constraints. You can remove traffic constraints either individually or by group. Click the remove-constraint button shown in Figure 5.7 to choose one of the two options for removing traffic constraints in a pop-up menu shown in Figure 5.15.

Figure 5.15: Popup window for removing traffic constraints

Using this interface, you can conveniently remove a group of traffic constraints or individual constraints.

5.2.4 Resource dimensioning

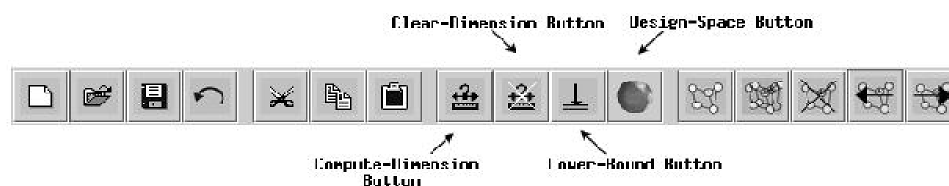
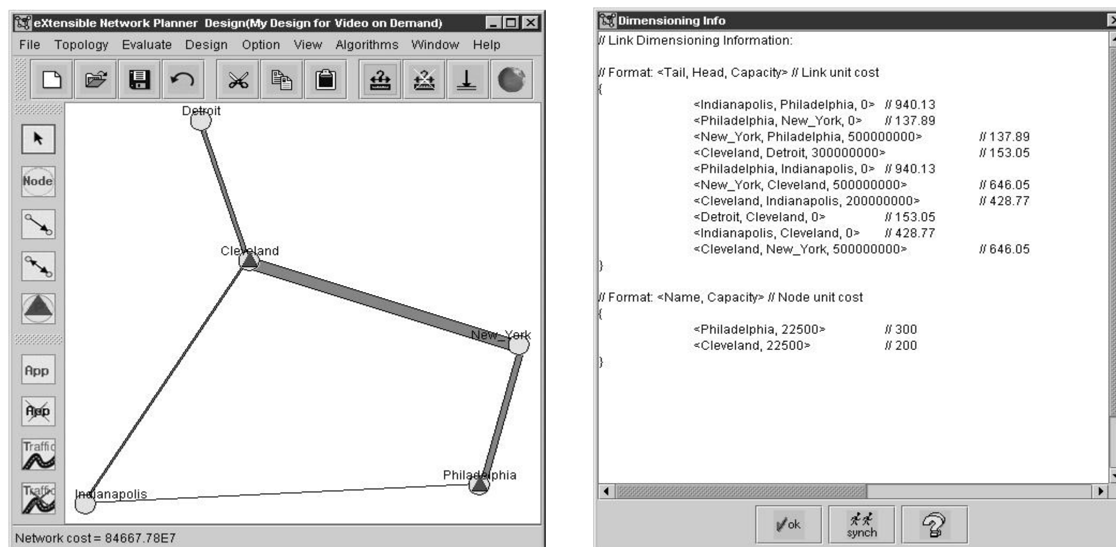


Figure 5.16: Buttons for evaluating network configurations

XNP is now ready to compute capacities required for links and processing resources. Click the compute-dimension button shown in Figure 5.16 to initiate the computation. XNP will iterate through all links and processing nodes to compute the capacities for them. Depending on the size of the network, the number of prospective applications, and the number of traffic constraints, the computation time might vary.

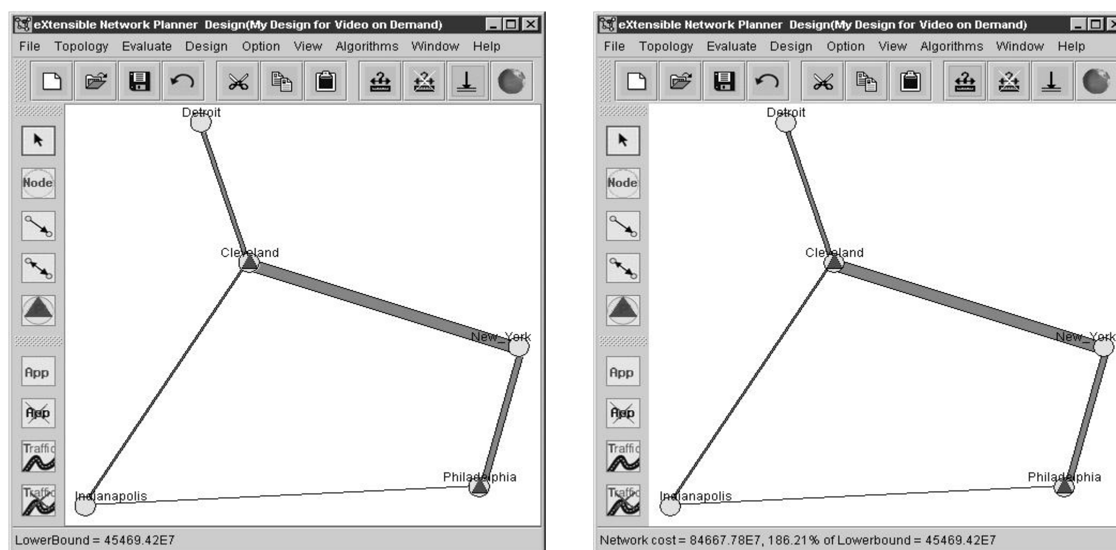


(a) Topology with dimensioning information

(b) Popup window displaying dimensioning information

Figure 5.17: XNP snap shots for dimensioning information

Once the dimensioning computation is finished, XNP redraws the topology to show the differences in resource capacities. Figure 5.17(a) shows the redrawn topology where the thickness of each link shows its capacity relative to other links' capacities. The dimensioning result can also be displayed on a separate pop-up window in more detail as shown in Figure 5.17(b). The figure lists every link and processing node with the computed capacity. XNP also computes the total cost associated with this network configuration, and shows the cost below the drawing area.



(a) Lower bound

(b) Cost relative to the lower bound

Figure 5.18: XNP snap shots for lower bound information

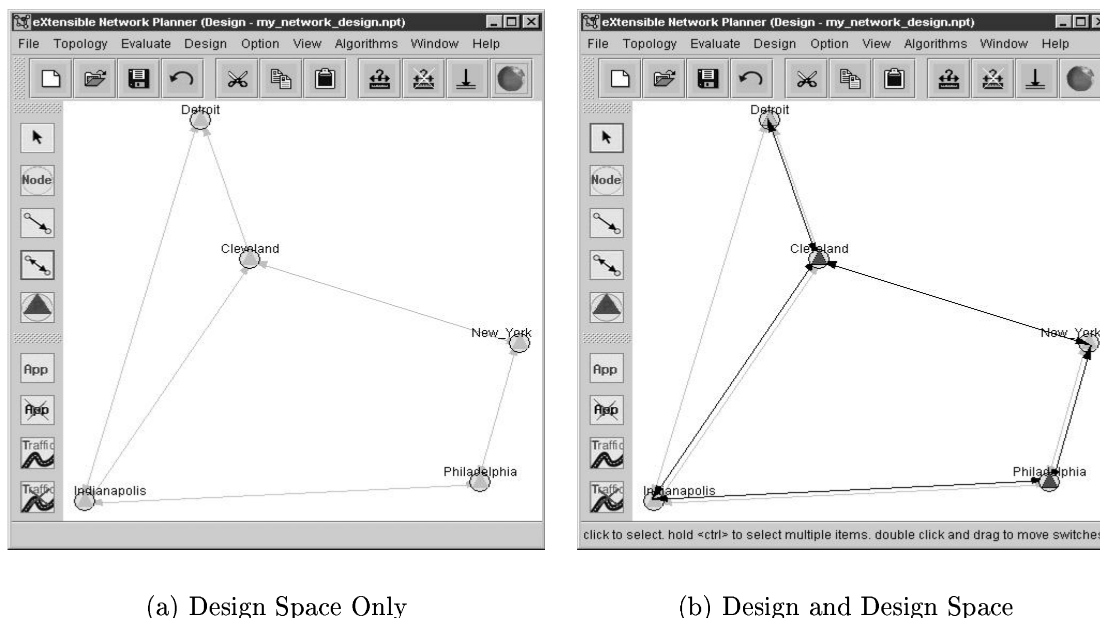
5.2.5 Evaluating network configurations

Once you obtain the dimensioning information for a network configuration, you can evaluate the network configuration by computing a lower bound on the cost of the best network configuration. The lower bound can vary depending on the “design space”, the candidate links and processing nodes that are allowed to be used in the network configuration. Note that we previously assumed that the “design space” included all nodes and all directed links. How to modify the “design space” will be detailed in the next section.

Meanwhile, to obtain a lower bound, click the lower-bound button shown in Figure 5.16. XNP will then compute a lower bound based on the given design space, and the result will be written below the drawing area as shown in Figure 5.18(a). You can click the compute-dimensioning button again to obtain the cost of the network relative to this lower bound, as shown in Figure 5.18(b). This relationship gives a convenient measure for estimating the quality of your network configuration. By adjusting options for computing a lower bound and therefore by considering your

design restrictions for using resources, you can obtain a more accurate and useful measure.

5.2.6 Design Space



(a) Design Space Only

(b) Design and Design Space

Figure 5.19: Two other views in XNP

The design space defines the set of resources that can potentially be used in your network configuration. In our earlier illustrations, we assumed that all direct links can be used in constructing a network topology. We also assumed that all nodes can be assigned processing capability. In reality, you might want to restrict this potential resource usage. For example, you might want to avoid placing a link between two very remote locations, or might want to avoid assigning processing capability to certain nodes for maintenance reasons. XNP allows you to express these restrictions by the "design space" and to limit the choices of resources within this space. Naturally, you would create your design space prior to constructing a trial topology.

To better assist you in creating a design space and a trial topology, XNP provides three different ways to view the drawing area. You can view the trial topology

only, the trial topology and the design space together, or the design space only. By clicking the design-space button shown in Figure 5.16, you can change to a different view. This allows you to have the view appropriate for the current task at any point of the design process. In the previous figures, you have seen the views with the trial topology only. Figure 5.19 shows the examples of the two other views. Note that candidate links included in the design space are drawn by lines with a lighter color. Similarly, candidate processing locations are highlighted by a triangle also with a lighter color.

Now, in order to modify the default design space, you use the design-space button to view the design space only. To include or exclude a link in the design space, use the cut, copy, and paste buttons. To enable or disable potential processing capability at a node, use the processing button.

5.3 Design Operations

XNP allows you to manage multiple designs at a time, so that you can try different trial topologies and compare them with each other conveniently. Figure 5.20 shows the buttons used for creating, deleting and browsing multiple designs.

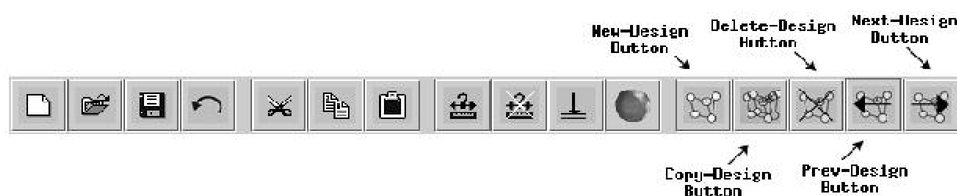


Figure 5.20: Buttons for managing designs

- Creating a design

  To create a blank design, click the new-design button. XNP will add a new design to the list of current designs, and display the blank drawing area for the new design.

- Browsing designs

  XNP keeps the current designs as a list, and allows you to browse the list. You

can use the two buttons, the prev-design button and the next-design button to do so. By clicking the prev-design button, you can go to the previous design in the list. The drawing area will then display the prev-design. Similarly, by clicking the next-design button, you can go to the next design in the list. The drawing area will then display the next-design.

- Deleting a design

  To delete an existing design, first go to the design you want to delete, and click the delete-design button.

5.4 File Operations

XNP allows you to store the current designs in a file. Figure 5.21 shows the buttons used for the file operations.

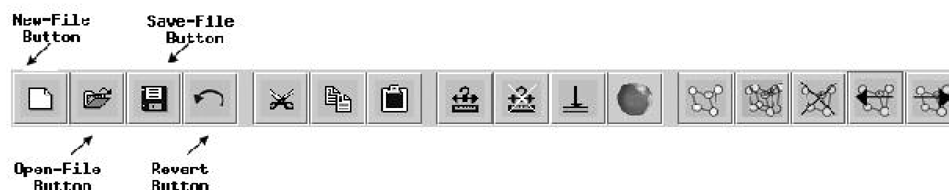


Figure 5.21: Buttons for file operations

- Create a new file

  To create a new file, click the new-file button. It will create a file with a new design. The drawing area will display the blank drawing area for the design.

- Open a file

  To open an existing file, click the open-file button. It will prompt you with a pop-up menu where you can select a file from the current directory. The file has to be in the XNP file format. By default, the pop-up menu will display only the files of the XNP file format.

- Save a file

  To save the current list of designs as a file, click the save-file button. If no file name has been given, it will prompt you with a pop-up menu in which you can enter or select the name of the file that the designs will be saved to.

- Revert to a previously saved file

  In case you want to ignore the change you have made to the current designs, you can revert to the previously saved file. XNP will discard the changes that have been made since you saved last time, and re-display the unmodified version.

5.5 Algorithms implemented in XNP

This section describes the algorithms that have been implemented in XNP. These algorithms help network designers by automating various tasks related to generating topologies, modifying topologies, or describing traffic constraints, that are otherwise time consuming.

5.5.1 Generating and modifying network topologies

XNP provides the following nine algorithms to generate and modify network topologies. All of these algorithms are applied to the current view. In other words, if the current view displays the trial topology or both the trial topology and the design space, the algorithms will modify the trial topology. If the current view displays the design space only, the algorithms will modify the design space. Any modification, particularly addition, to the trial topology is restricted by the design space. Only the links that exist in the design space can be added to the trial topology, and only the nodes that are specified as candidate processing nodes in the design space can be activated as processing nodes in the trial topology.

- Complete network

  This algorithm simply creates a complete topology that connects every pair of

nodes. When it modifies the trial topology, it simply adds every link defined in the design space to the trial topology.

- Delaunay triangulation

  This algorithm creates the Delaunay triangulation [51] defined by the locations of the nodes in the current view.

- Delaunay triangulation with trimmed links

  This algorithm is a variation of the Delaunay triangulation. It first creates the Delaunay triangulation, and then eliminates relatively "long" links whose end points are connected by alternative paths that are not excessively "longer" than the direct links. You can control this link elimination process by specifying a threshold on the ratio of the length of the direct link to the length of the shortest alternative path.

- Link complement

  This algorithm simply adds direct links that are not currently drawn, and deletes the existing links. Again, if the current view displays the trial topology, a link can be added to the trial topology only if it is in the design space.

- Minimum spanning tree

  This algorithm creates a minimum spanning tree for the current set of nodes. If the current view displays the trial topology, the minimum spanning tree is restricted to use only the links that are in the design space.

- Random link adder

  This algorithm randomly adds links to the current view. You must specify the number of links to be added.

- Random node adder

  This algorithm randomly adds nodes to the current view. You must specify the number of nodes to be added.

- Star network

  This algorithm creates a star-shaped topology where every node but a "center" node has a unique link to the center. You must click a node you want to set as the center before running this algorithm. Again, if the current view displays the trial topology, the star topology is restricted to use only the links that are in the design space.

- Symmetric link adder

  This algorithm makes the current topology bidirectional by adding a link with the opposite direction for each of the existing links.

5.5.2 Placing network nodes

- Geographic planar projection

  This algorithm performs the planar projection [52]. Given a list of coordinates for locations on the globe, the planar projection projects these coordinates onto a flat surface touching the globe. Our implementation assumes that the surface contacts the globe at the South Pole and that each coordinate is composed of a latitude and a longitude.

  You must supply the coordinates by either entering them directly to XNP or reading them from a file. You can also supply the traffic estimate of each location in addition to its coordinate. This algorithm will automatically apply this traffic estimate to the maximum incoming and outgoing traffic of the location.

- Grid network

  This algorithm creates a $N \times M$ grid network. You can set N and M to any positive numbers.

- Torus network

  This algorithm creates a $N \times M$ torus network, a grid network augmented with

links that wrap around the network to connect each node on an edge of the grid to the node on the opposite side.

5.5.3 Generating traffic constraints

These algorithms generate a set of traffic constraints according to the policies specified below. Both policies were studied in the experimental study of constraint-based network design [17].

- Localized traffic constraints

  In these constraints, amount of traffic a node sends to another node is a (decreasing) function of the distance between the two nodes. Similarly, the amount of traffic a node receives from another node is also a (decreasing) function of the distance between the two nodes.

- Proportioned Pairwise traffic constraints

  For any two nodes u and v, let $f(u,v) = \frac{\omega_v}{\sum \omega_x}$, $g(u,v) = \frac{\alpha_u}{\sum \alpha_x}$, where α_x is the maximum outgoing traffic at node x, and ω_x is the maximum incoming traffic at node x. These constraints restrict the simultaneous traffic from the node u to the node v by limiting it to $\mu_s = c \times f(u,v)\alpha_u$ when it leaves u, and also by limiting it to $\mu_t = c \times g(u,v)\omega_v$ when it terminates at v. Here, c is a constant called the relaxation factor. When c is large enough that $\mu_s(u,v) \geq \alpha_u$ and $\mu_t(u,v) \geq \omega_v$, these constraints are neglected. You must provide the relaxation factor before running the algorithm.

Chapter 6

Extensible Network Design Using XNP

6.1 Experimenting with Various Design Choices

Now please recall the Acme company introduced in Section 4.2 of Chapter 4, which had a plan to build a network connecting its five locations in the northeastern United States. The network was to support a video on-demand application for employee education and corporate announcements. Using the video on-demand application, Acme wanted to allow a source node to send compressed video data, an intermediate router node to decompress the video, and a destination node to receive the resulting uncompressed video data. In Chapter 4, we briefly introduced how to handle the network design for Acme using our tool, the Extensible Network Planner (XNP), and elaborated on the methodologies behind XNP. In this chapter, we demonstrate our network design approach in more practical situations that involve larger sets of geographical locations. Assume that Acme has expanded its business to include twenty branch locations, each of which is located at one of the twenty most populous metropolitan areas in the United States. Now, the task is to design and dimension the network so that it supports the on-demand video application for a new set of video sources while providing decompression for all receivers. To investigate the

impact of the new application format and processing locations on our network design more closely, we assume that resources have already been provisioned for conventional applications and focus our design problem on providing resources for the on-demand video application.

This time, the compressed video data is kept on four servers in the New York, Chicago, Los Angeles, and Atlanta locations. The compression factor is 10, and the decompression algorithm executes approximately 200 instructions per byte of the compressed video. Every video server maintains copies of all videos, and therefore, viewers are naturally assumed to receive video transmissions from the geographically nearest source. For example, video receivers in San Diego or Los Angeles will receive video transmitted from the source in Los Angeles only. Table 6.1 shows how video sources and receivers are associated. Each row contains a video source and all locations that will receive video transmissions from the source.

Table 6.1: Sources and their receiver locations for the US network

Source location	Receiver locations
New York	Pittsburgh, Washington D.C., Philadelphia, Boston, New York
Chicago	Minneapolis, St. Louis, Detroit, Cleveland, Chicago
Los Angeles	Seattle, Denver, San Francisco, San Diego, Phoenix, Los Angeles
Atlanta	Dallas, Houston, Miami, Atlanta

Given the association between video servers and receiver locations, it is anticipated that ingress/egress traffic is proportional to the population at each location. Figure 6.1 shows an example of the popup window for restricting the total ingress and egress traffic at Minneapolis. It is further anticipated that only 10% of the outgoing traffic from the video sources will be on-demand compressed video traffic and at most 50% of the traffic reaching any of the other sites is expected to be decompressed video traffic. Figure 6.2(a) shows the popup window for restricting the outgoing compressed video traffic from the video source at New York. This constraint includes in

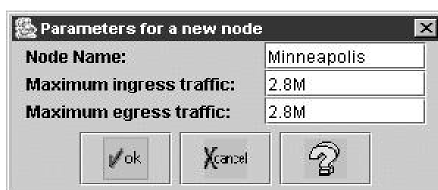
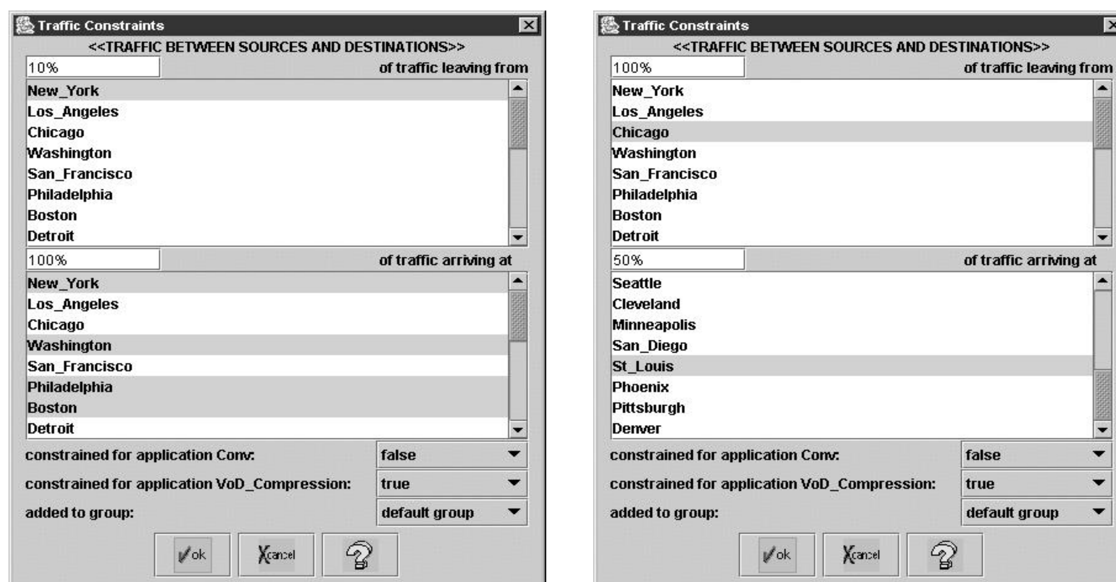


Figure 6.1: Traffic constraints for maximum incoming/outgoing traffic at Minneapolis, proportional to the population

the destination set only the locations that will receive video from the New York location, as specified in Table 6.1. Figure 6.2(b) shows the popup window for restricting the incoming (decompressed) video traffic at the St. Louis location. Because the St. Louis location receives video transmission always from Chicago, this constraint includes only the Chicago location in the source set.



(a) For outgoing video traffic at New York

(b) For incoming video traffic at Chicago

Figure 6.2: Traffic constraints for the video on-demand application

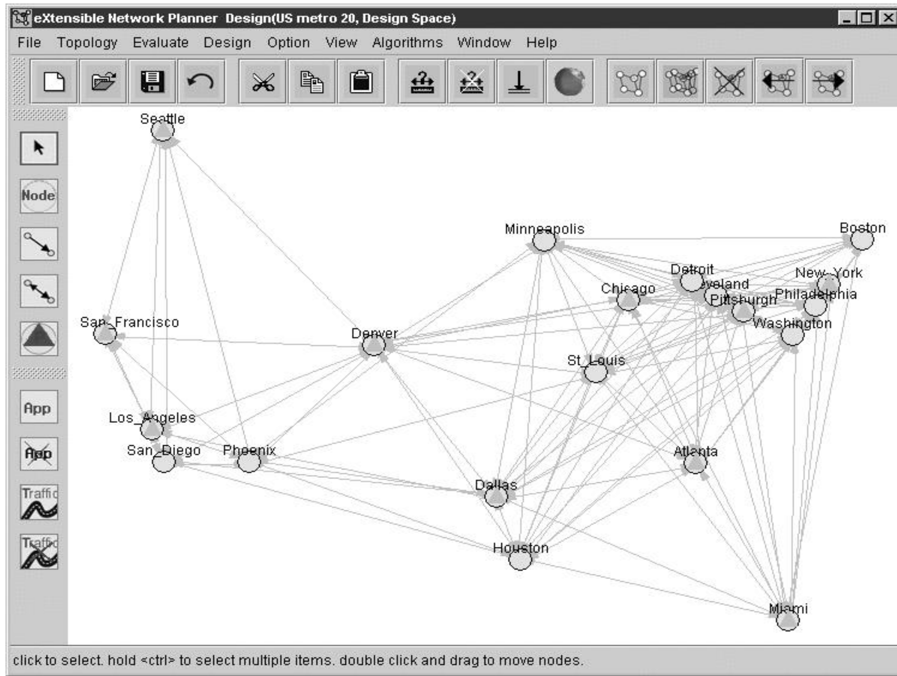
We also consider that videos can be also at any of the twenty locations, and distributed to the four sources to be stored, but far less frequently than the compressed on-demand video transmissions. We assume that there is no intermediate processing involved in these transmissions, which therefore are routed as for a conventional

application. Such traffic is anticipated to be at most 3% of the total egress traffic at each node, and at most 3% of the total ingress traffic at each of the video sources. These traffic constraints can be added in a similar way as shown in Figures 6.2(a) and 6.2(b).

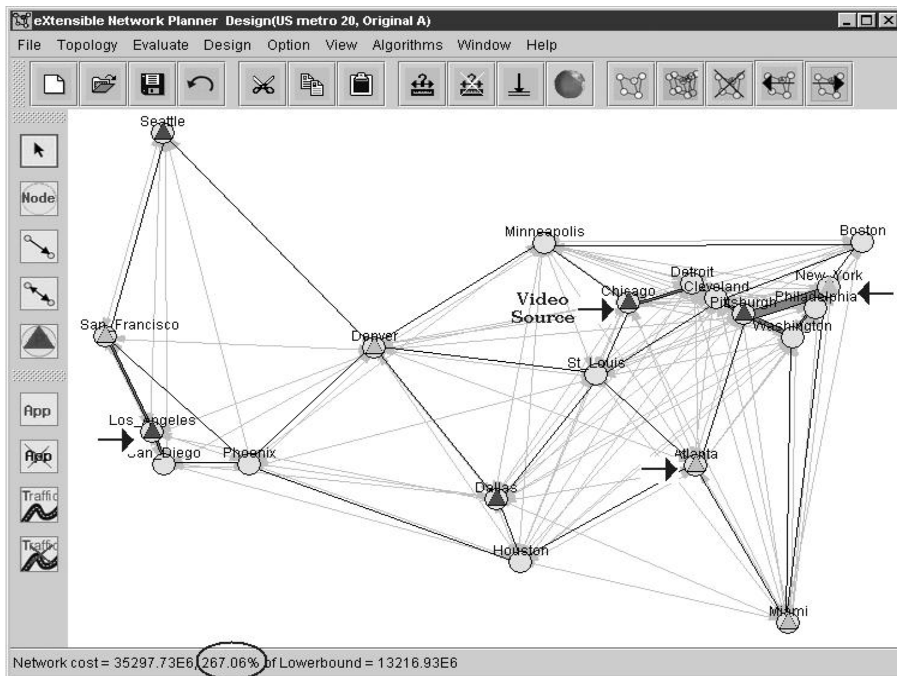
We restricted the design space by defining a set of potential decompression nodes and a set of potential links. First, the nodes were selected based on geographic location and population. This selection was guided by the observation that the design cost could be reduced by performing decompression at the destinations that are more populous and therefore contain more video viewers. It was also considered that the design cost could be reduced by designating decompression processing to locations where otherwise no potential processing locations are available in their vicinities. The selected locations are Los Angeles, New York, Chicago, Miami, Atlanta, Dallas, Pittsburgh, Denver, Seattle, and San Francisco. We also avoided extremely long links in our design by considering only those links whose lengths are at most half of the network's diameter (the maximum distance between any two locations). Figure 6.3(a) shows the design space in XNP, where potential links and processing locations are drawn in light gray.

We have implemented in XNP a number of algorithms that automate the topology generation so that users can quickly configure and compare different network designs. This feature was used to generate a range of solutions for our example design, which are compared below.

- **Hand-crafted**: The handcrafted topology is shown in Figure 6.3(b).
- **MST**: This topology forms the minimum spanning tree on the twenty locations, using the links in the design space shown in Figure 6.4(a).
- **Delaunay**: This topology is created from performing the Delaunay Triangulation [51] over the coordinates of the twenty locations. Figure 6.4(b) shows the resulting topology.

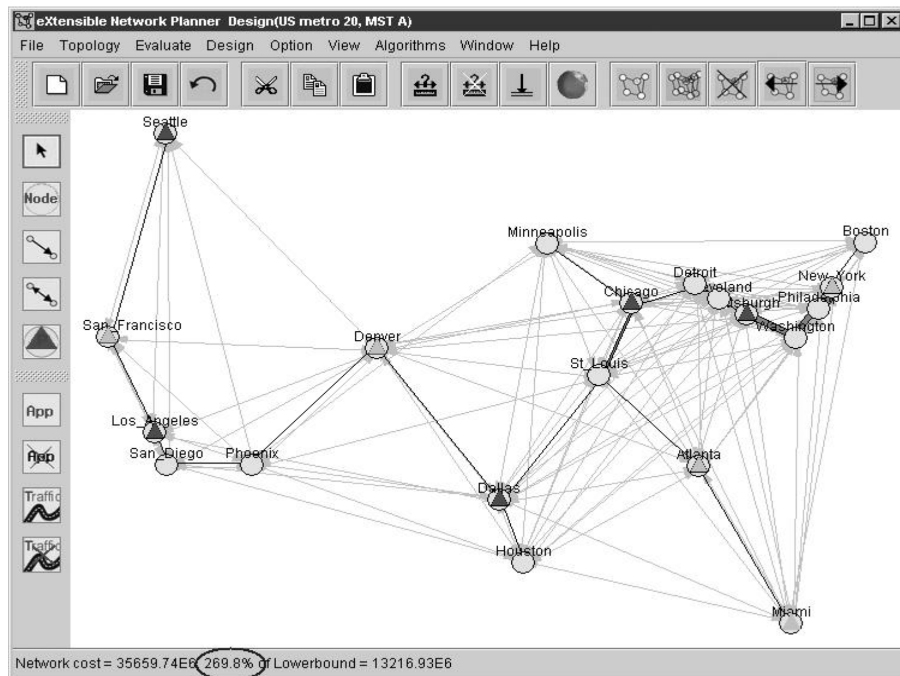


(a) Design space

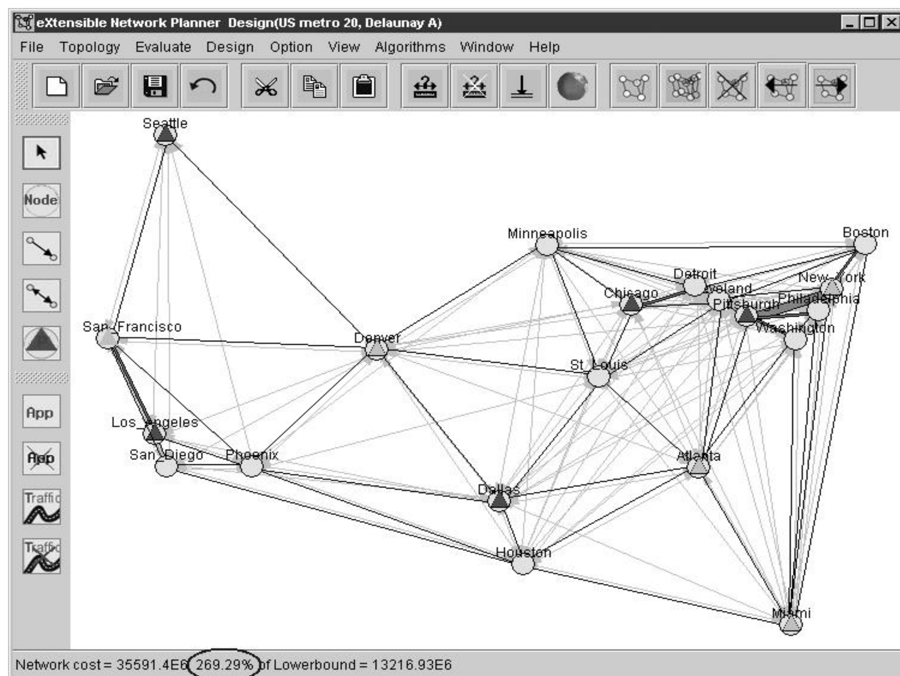


(b) Hand-crafted topology

Figure 6.3: Networks for the 20 largest metropolitan areas for US

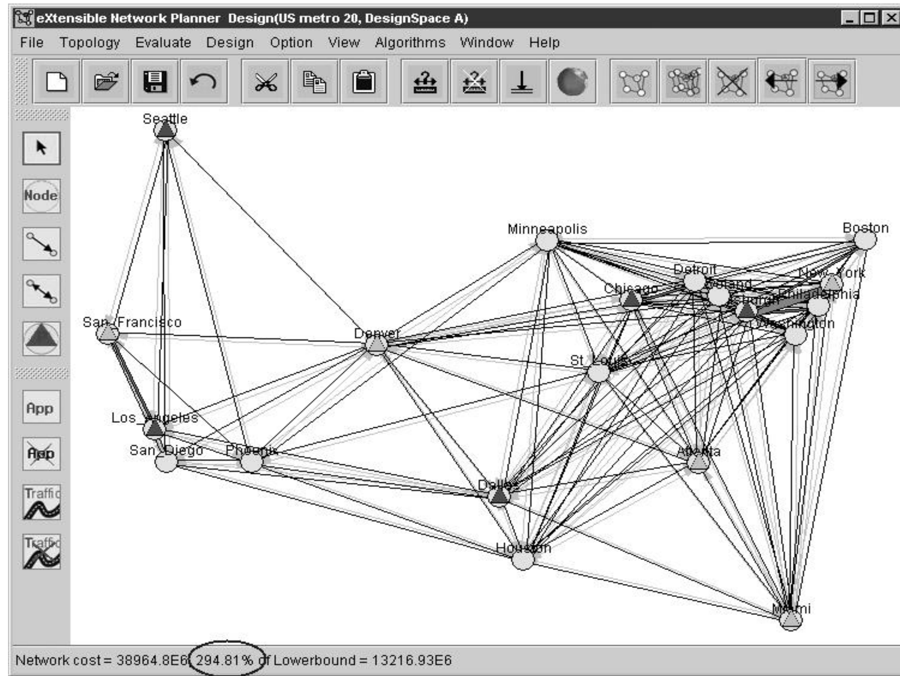


(a) MST topology

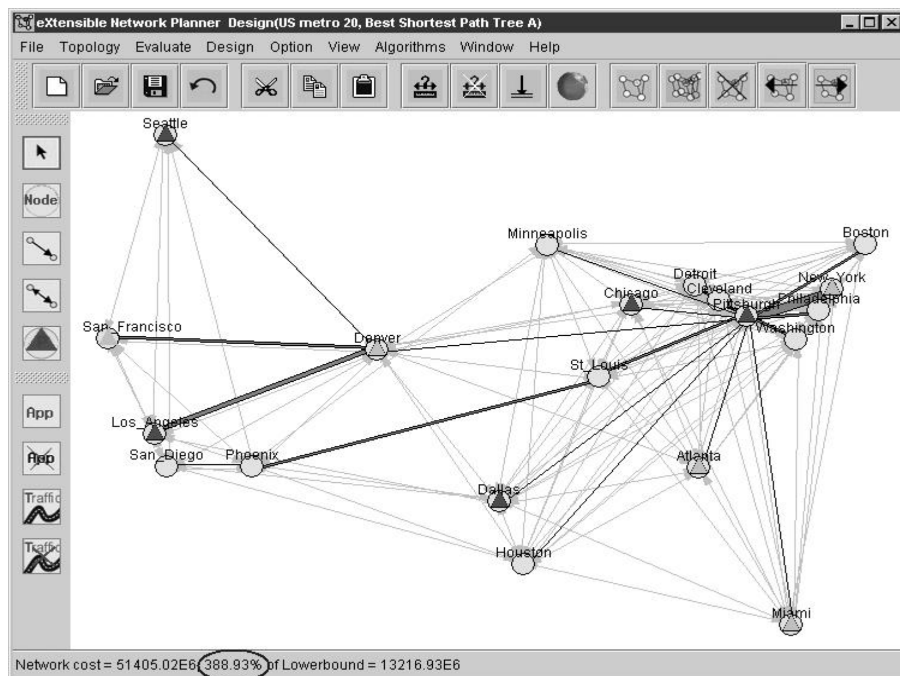


(b) Delaunay topology

Figure 6.4: Networks for the 20 largest metropolitan areas for US



(a) All-links topology



(b) Best-SPT topology

Figure 6.5: Networks for the 20 largest metropolitan areas for US

- **All-links**: This topology uses all links provided by the design space, as shown in Figure 6.5(a).

- **Best-SPT**: This topology, as shown in Figure 6.5(b), induces the least network cost among all shortest path tree topologies. Here, a shortest path tree topology is a subset of the design space and forms a shortest path tree rooted at a fixed node. In Figure 6.5(b), the root is located at Pittsburgh.

Topology generation algorithms, including the ones in this list, were described in Section 5.5 of Chapter 5.

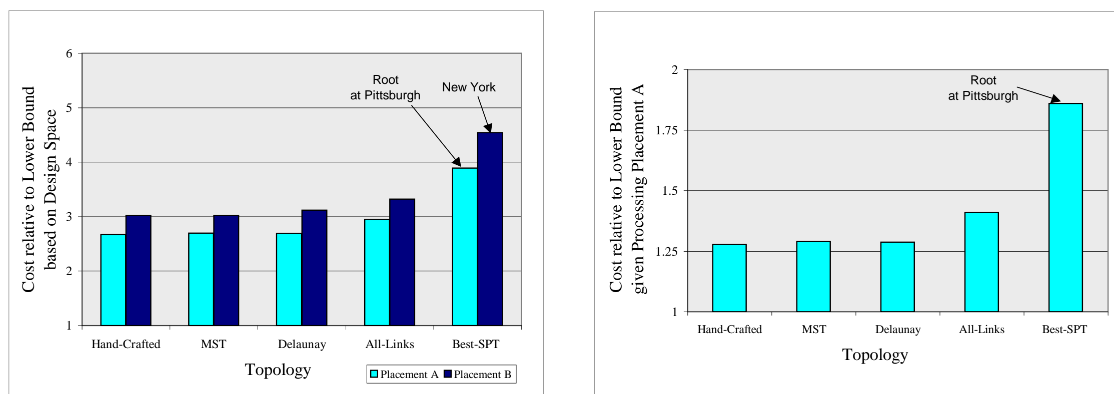
Because of the decompression required for the video traffic, processing locations have to be designated. In our experiments, we consider the following two sets of five sites that are dispersed over the network.

- **Placement A**: Pittsburgh, Dallas, Chicago, Los Angeles, Seattle

- **Placement B**: New York, Atlanta, Miami, Denver, San Francisco

Placement A was applied to all the topologies in Figures 6.3, 6.4, and 6.5 where the processing locations are highlighted by a triangle.

As one may want to consider different, or better, topologies and processing placements, our goal here is to demonstrate how users can design and evaluate extensible networks using our method. Using XNP, we repeated the following steps: build a topology, designate processing nodes, and dimension the resources. The trial designs in Figure 6.3, 6.4, and 6.5 are displayed with links dimensioned according to the given traffic constraints. On the bottom of each figure, XNP shows the cost of each design and the lower bound cost, given the potential links and processing locations of the design space. For simplicity, we considered only the cost of links in this experiment by setting the costs at processing nodes to zero. The cost at each link was computed as the product of the capacity and the length of the link.



(a) Design costs relative to the lower bound

(b) Design costs relative to the lower bound with fixed processing placement A

Figure 6.6: Cost comparison for US topologies

Figure 6.6(a) compares the costs resulting from all the choices of topology and processing placement, relative to the lower bound computed for the design space. Given a topology, one placement policy provides a lower cost than the other, possibly by providing better (smaller-cost) accesses to processing nodes. This result indicates that the combination of placement A and the hand-crafted topology is preferred as the least cost design, while the costs of all three designs–the hand-crafted, MST, and Delaunay topologies–lie within a close range. Such choices are, however, specific to the given traffic constraints, which allow more resource sharing among traffic bound to and from various terminals. Details of this issue have been studied by Ma *et al.* [17] In search of a better design, one may take one of the designs, and modify it by adding links, removing links, adding more processing locations, or removing current processing locations. Consider Figure 6.6(b), which shows the comparison of the costs to the lower bound, which was computed assuming that the processing locations are fixed according to placement A and only the link selection can be varied in search of the best design. Here, all three topologies provide relatively low costs that are approximately 1.3 times the lower bound given placement A. This result indicates that further enhancement of the topology alone is limited and that different placement policies must be combined in search of better designs.

The lower bound and the cost comparison of the existing topologies can be useful guides for modifying the design. Ideally, the goal should be to modify the design so that its cost becomes closer to the lower bound and eventually reaches the minimum. As discussed in Chapter 4, finding the design with the minimum cost is a hard problem. However, many heuristic techniques can be applied to this process, which we leave to future work.

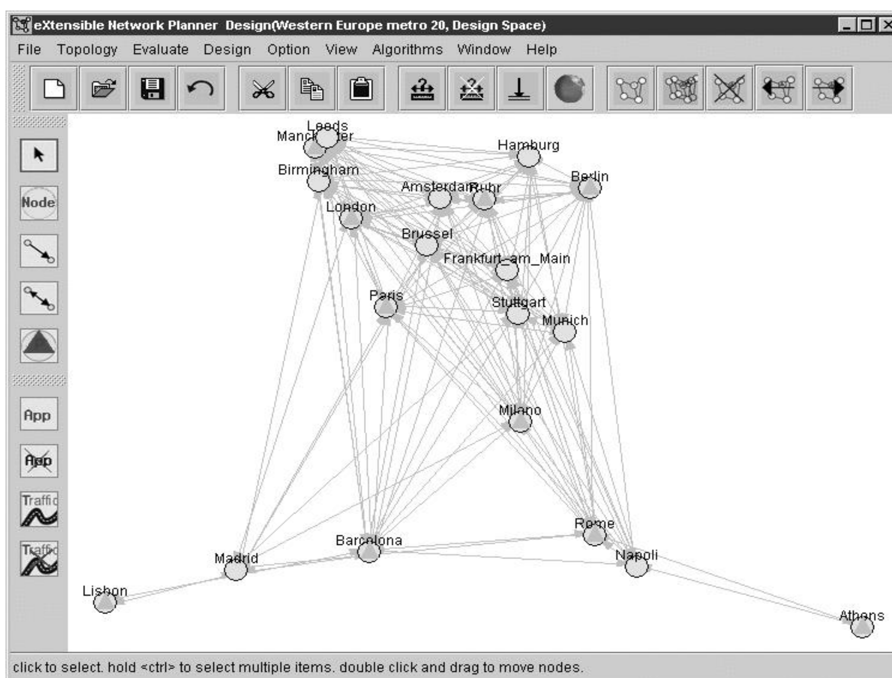
In another situation, the results are also affected by the geographical distribution of traffic, which depends heavily on the populations at the locations. We considered a different population distribution in Figure 6.7(a), which shows a design space comprising the twenty largest metropolitan areas in Western Europe. We applied the same application format and similar traffic constraints that limit the compressed video traffic to 10% of the outgoing traffic from the video sources and decompressed traffic to 50% of the traffic reaching any location, assuming video sources at London, Berlin, Madrid, and Rome. As in the previous case, each video source is designated to transmit compressed videos for a set of receivers, as shown in Table 6.2.

Table 6.2: Sources and their receiver locations for the Western Europe network

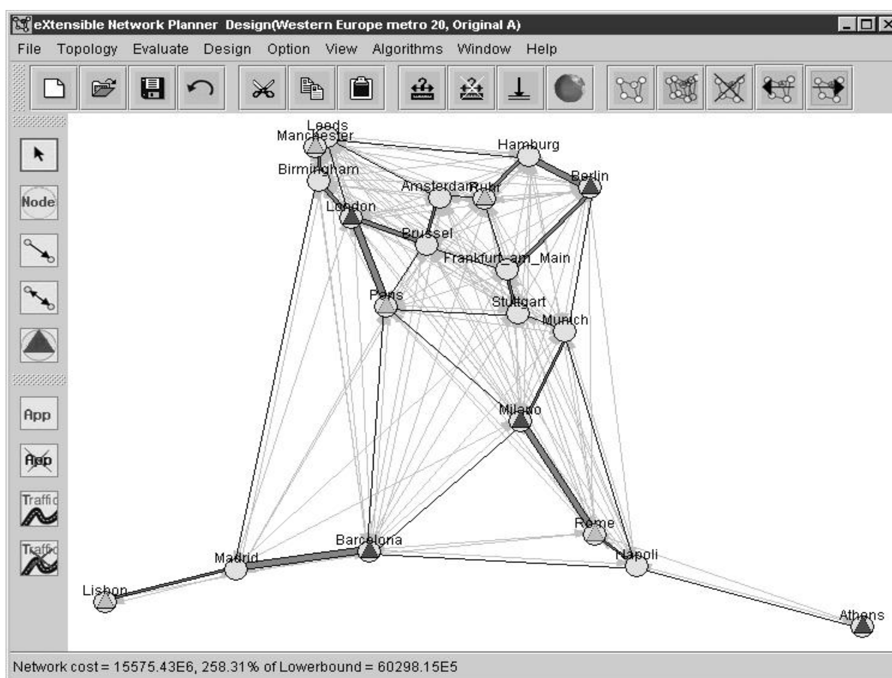
Source location	Receiver locations
London	Leeds, Manchester, Birmingham, Paris, Brussels, Amsterdam, London
Berlin	Hamburg, Ruhr, Frankfurt am Main, Stuttgart, Munich, Berlin
Madrid	Lisbon, Barcelona, Madrid
Rome	Milano, Napoli, Athens, Rome

We applied the five types of topologies described earlier and the following two processing placements.

- **Placement A**: London, Stuttgart, Milano, Barcelona, Athens
- **Placement B**: Paris, Lisbon, Ruhr, Rome, Manchester

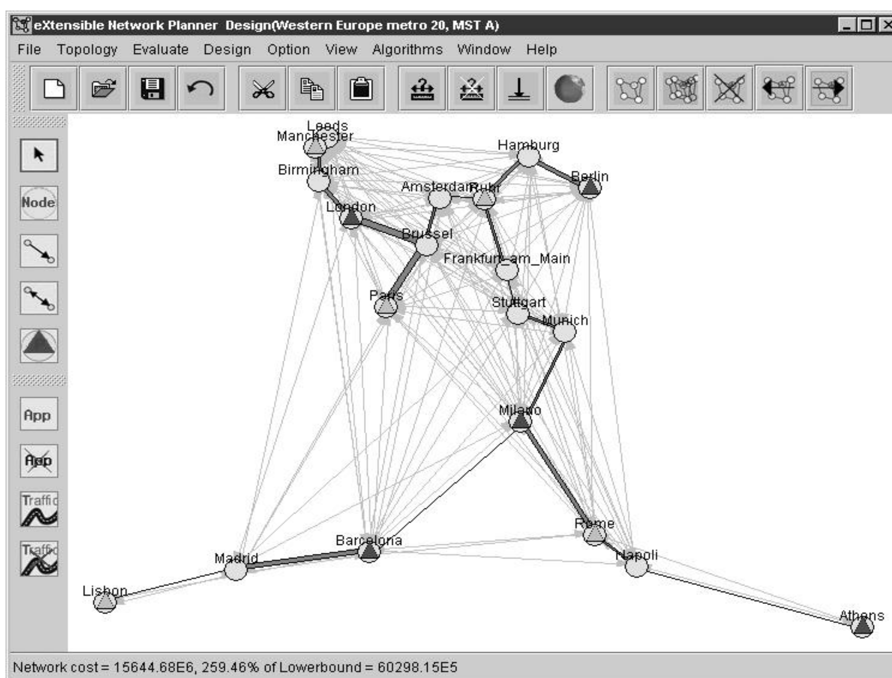


(a) Design space

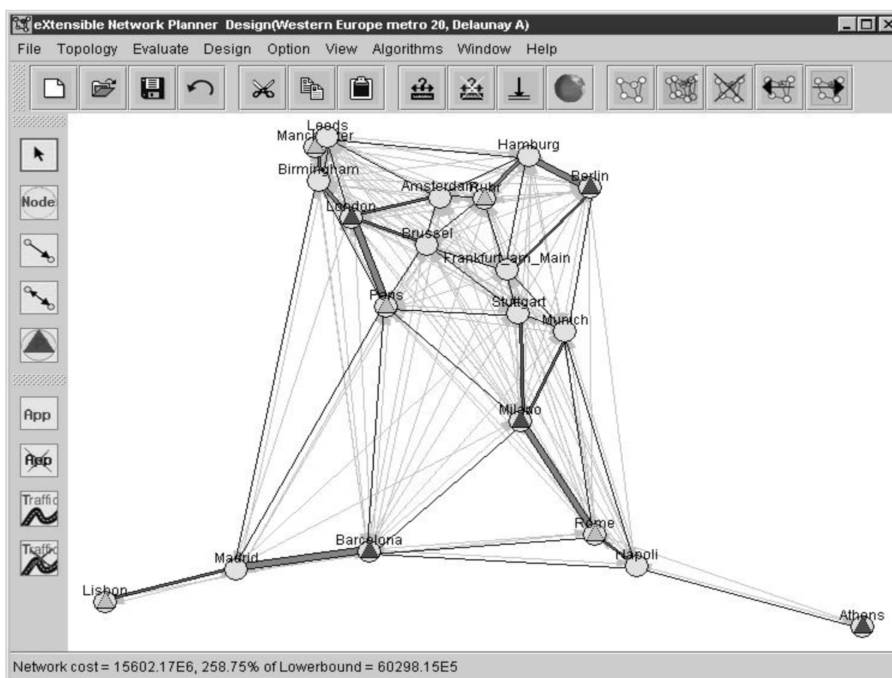


(b) Hand-crafted topology

Figure 6.7: Networks for the 20 largest metropolitan areas in Western Europe

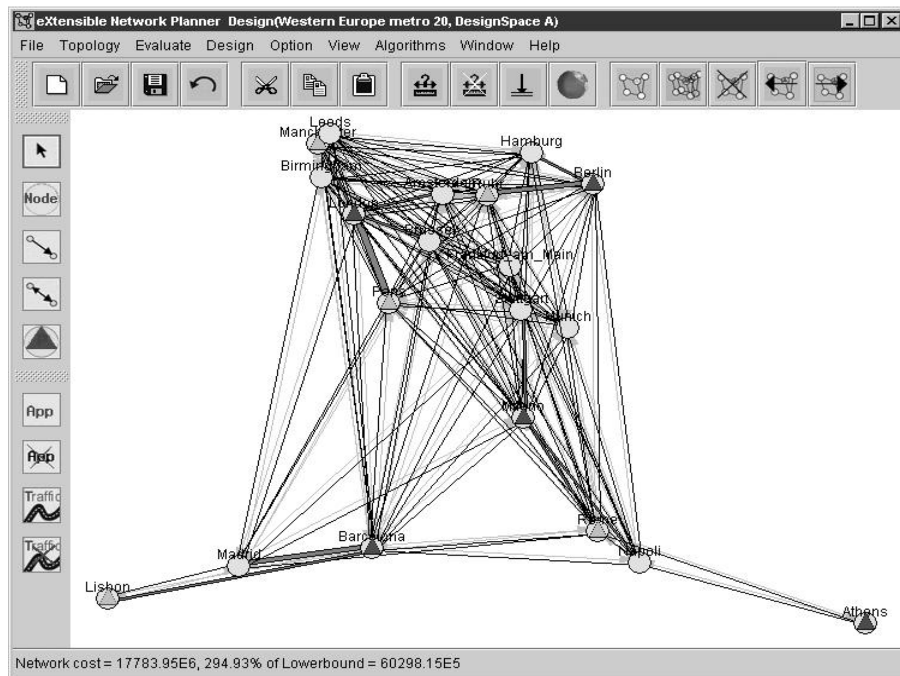


(a) MST topology

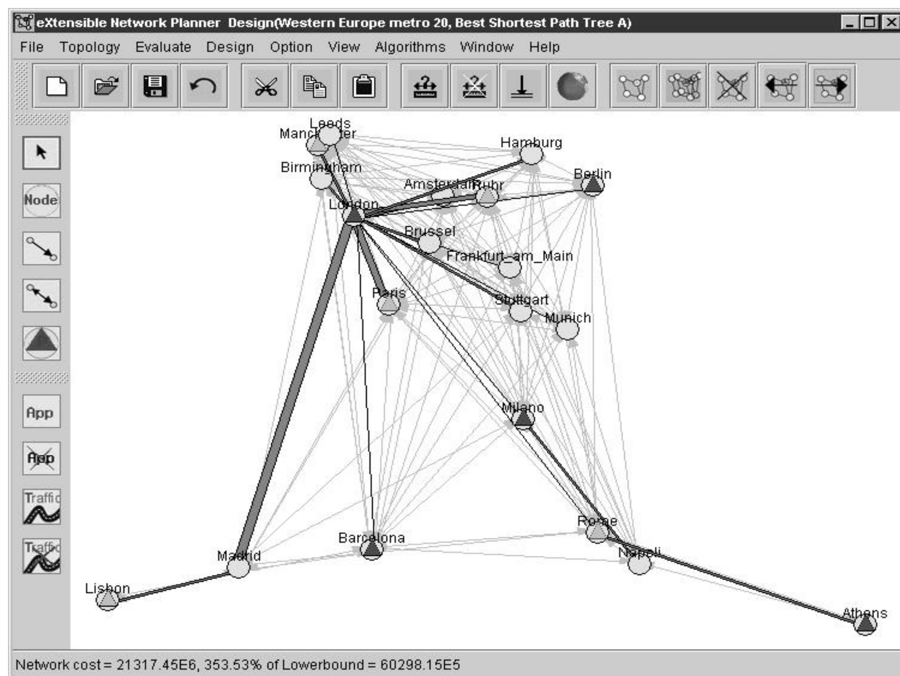


(b) Delaunay topology

Figure 6.8: Networks for the 20 largest metropolitan areas in Western Europe



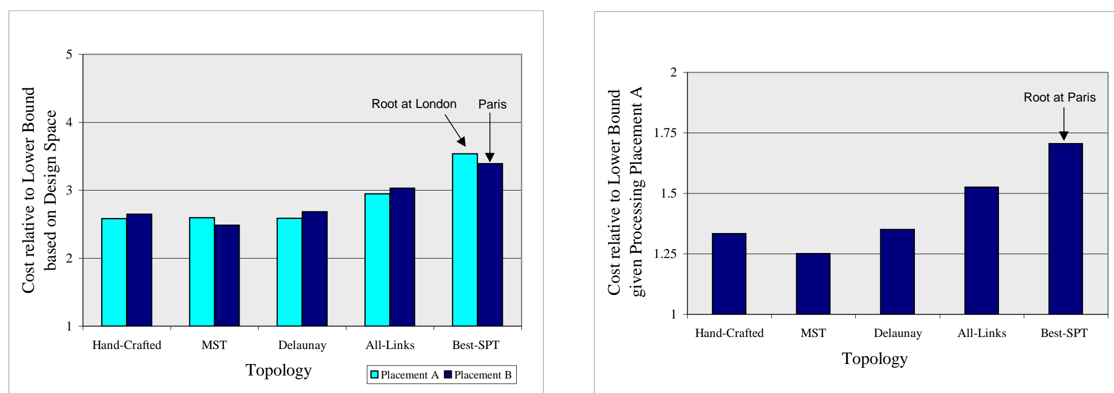
(a) All-links topology



(b) Best-SPT topology

Figure 6.9: Networks for the 20 largest metropolitan areas in Western Europe

Figures 6.7, 6.8, and 6.9 show the resulting screen shots for the five topologies with placement A.



(a) Comparison of the design costs relative to the lower bound

(b) Design costs relative to the lower bound with fixed processing placement B

Figure 6.10: Cost comparison for Western European topologies

Figure 6.10(a) shows cost comparisons that are different from those of the US topologies, reflecting the influence of geographic and demographic distributions. Here, the combination of the MST topology and placement B is preferred as the least-cost design.

Overall, the topologies created by the automated algorithms can be convenient points to start designing networks, and the search for a better design can be guided by comparing the resulting cost of the current design to the lower bound, or to the costs of the existing designs. In the next section, we extend our experiment to finding a better design for the US network and the Western European network by modifying the hand-crafted topology.

6.2 Enhancing the Trial Designs

We now turn to enhancing the designs introduced in the previous section. To make the network more representative of practical situations, we require that the resulting topology must be 2-connected. (A graph G is k-connected if there does not exist a set of $k-1$ vertices whose removal disconnects the graph). Our requirement ensures that no single node can disconnect the entire network when it fails.

Note that while the MST topology incurs a lesser or equivalent cost compared to the other trial topologies, as shown Figure 6.6(a) and 6.10(a), tree topologies in general do not satisfy our requirement for 2-connectedness. As the starting point in search of a better design, we instead select the hand-crafted topology.

In addition to the 2-connectedness requirement, we also restrict the number of processing locations to be five at most . While processing resources have no cost, as assumed in the previous section, their locations can affect the cost associated with link usage. To understand this effect better, let us take the hand-crafted design of the US network, place processing resources at all the potential locations specified by the design space, and compute the cost. Figure 6.11 shows the design and its cost with ten processing locations, which shows a significant improvement from the previous designs. While this particular design assigns more locations than are allowed, the result clearly shows that the design space provides a good set of candidate locations that can potentially lower the cost to 1.45 times the lower bound in the best case. While a better set of candidate locations may well exist, we will keep the design space as it is in this experiment and attempt to enhance the design within the space.

First, we select another two sets of processing locations, in addition to the previous placements A and B. The new placements C and D are given as follows.

- **Placement C**: New York, Los Angeles, Atlanta, Chicago, Dallas
- **Placement D**: New York, Los Angeles, Atlanta, Chicago, Seattle

Recall that the on-demand video application requires less bandwidth when the compression processing is placed closer to the receiver, because the processing expands

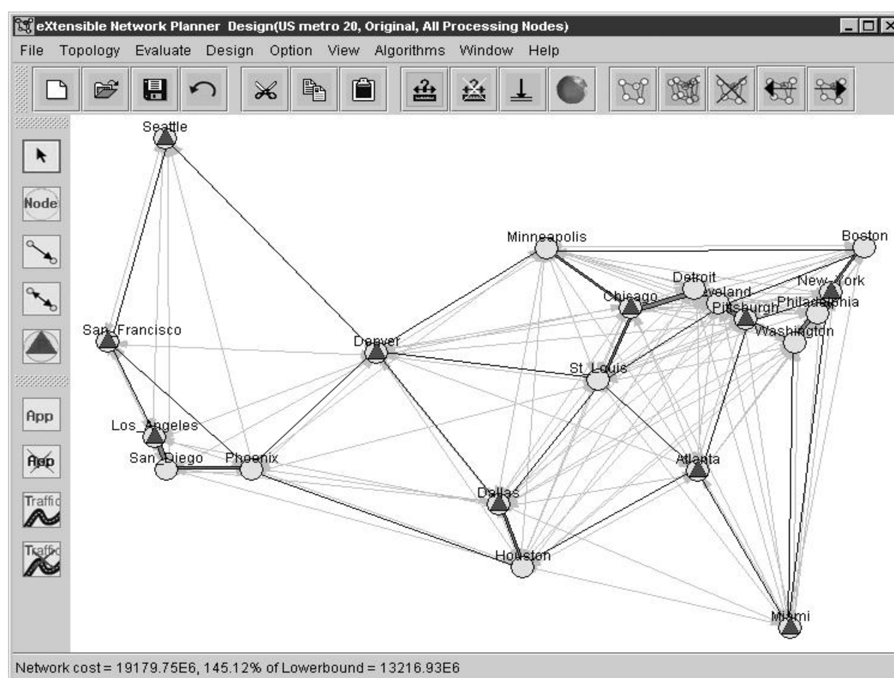
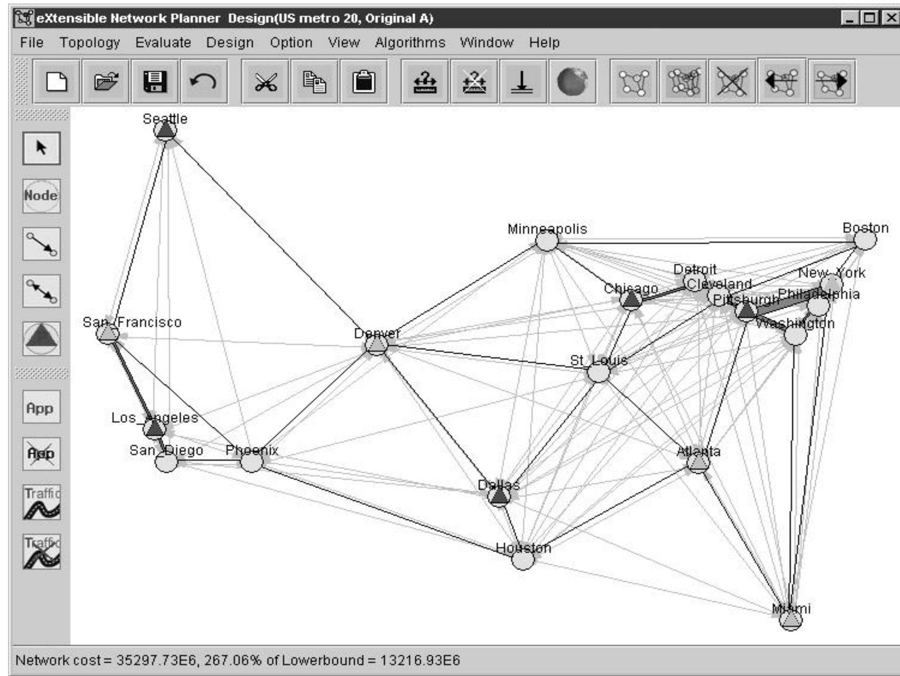


Figure 6.11: Networks for the 20 largest metropolitan areas in US, processing at all possible locations

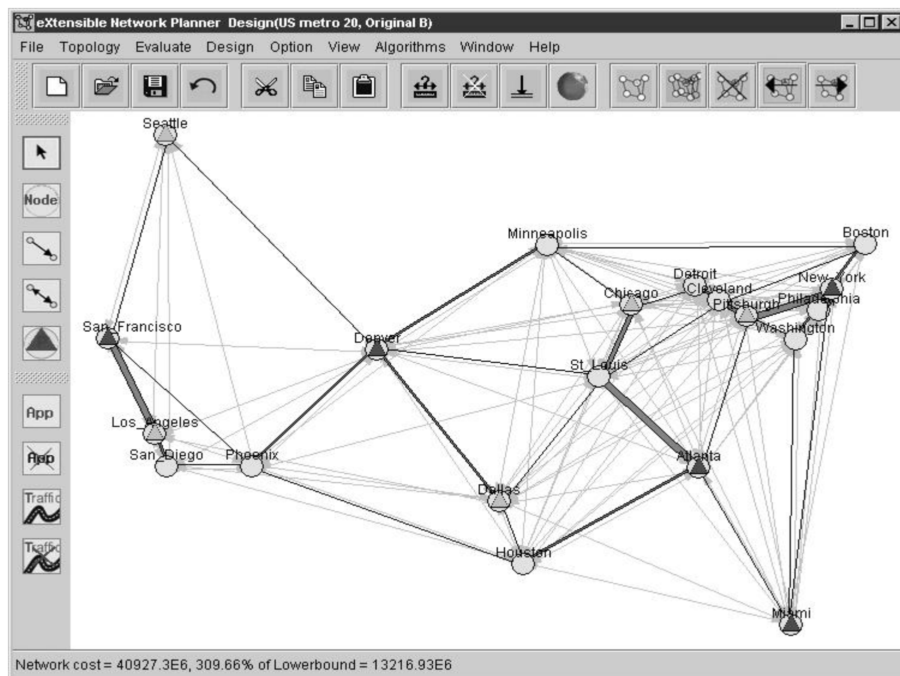
the bandwidth needed on the rest of the route. Therefore, by placing the processing closer to the locations that have more viewers, we can reduce the overall bandwidth requirement, and thus reduce the design cost. For this reason, we include in placements C and D the three most populous locations, New York, Los Angeles, and Chicago, which are also coincidentally video sources. Then, we select two more locations that are geographically closest to more receivers which have no nearby processing location. We choose Atlanta and Dallas for placement C, and Atlanta and Seattle for placement D.

Figures 6.12(a) and 6.12(b) show the designs and their costs when placements A and B are selected. Placement A gives a better cost, which is 2.67 times the lower bound, than that of placement B, which is 3.1 times the lower bound. The new placements C and D decrease the cost even more, down to 2.05 and 2.2 times the lower bound, as shown in Figures 6.13(a) and 6.13(b).

Now that we have identified better processing placements, we turn to modifying the topology. To provide better sharing of links among traffic flows while maintaining

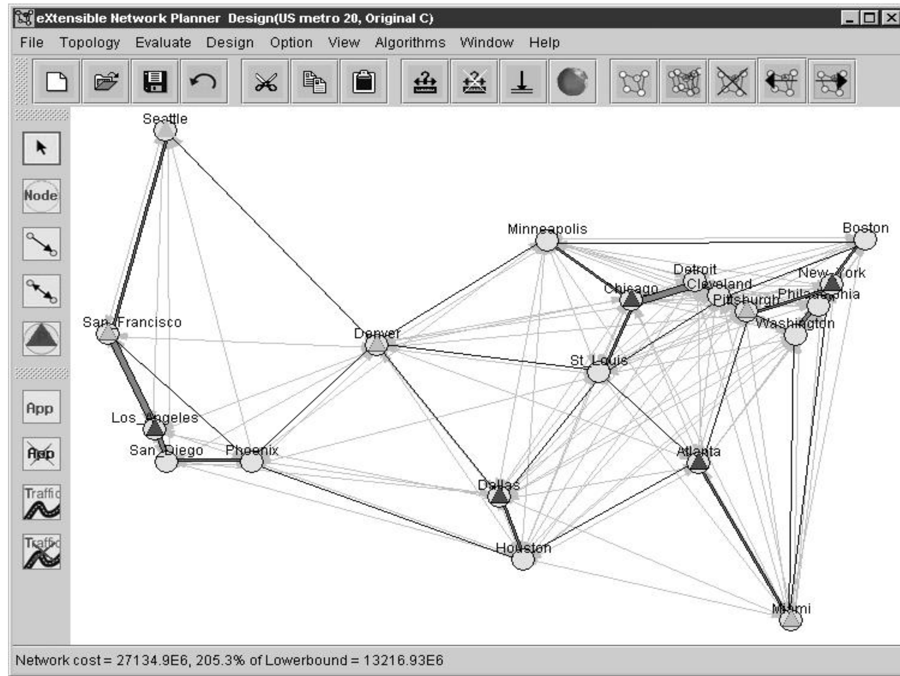


(a) Hand-crafted topology, Placement A

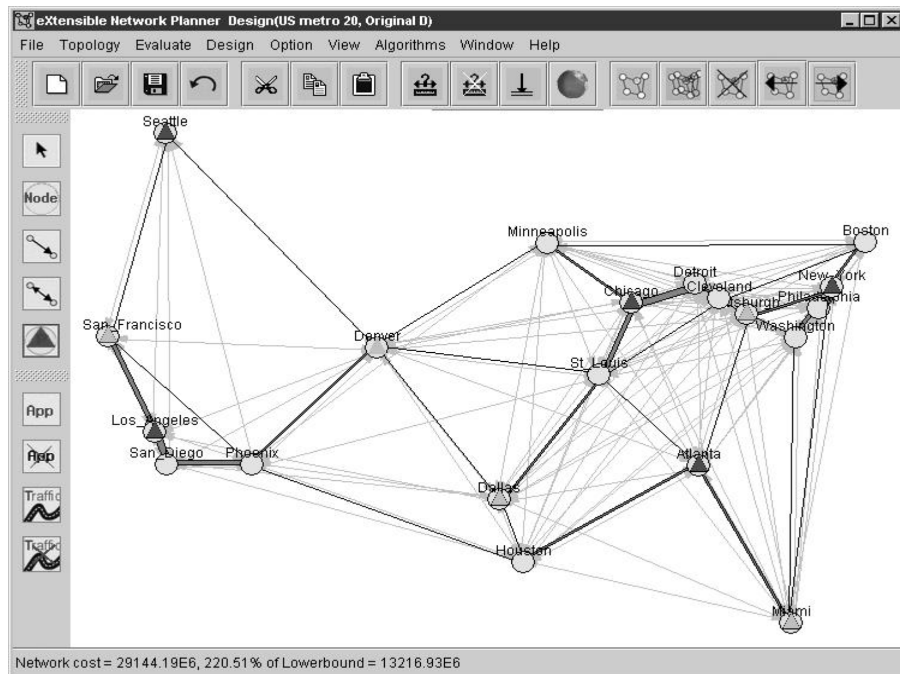


(b) Hand-crafted topology, Placement B

Figure 6.12: Networks for the 20 largest metropolitan areas



(a) Hand-crafted topology, Placement C



(b) Hand-crafted topology, Placement D

Figure 6.13: Networks for the 20 largest metropolitan areas

good connectivity, we simplify the hand-crafted topology by pruning some links and replacing others with alternative links. Table 6.3 lists the links that are removed from and added to the hand-crafted topology.

Table 6.3: Links added and removed for the US network

Removed Links	Added Links
(Phoenix, Denver)	(Los Angeles, Denver)
(Atlanta, Houston)	(Atlanta, Dallas)
(Denver, St. Louis)	(Denver, Chicago)
(Boston, Cleveland)	
(Minneapolis, Denver)	
(Miami, New York)	
(San Francisco, Phoenix)	

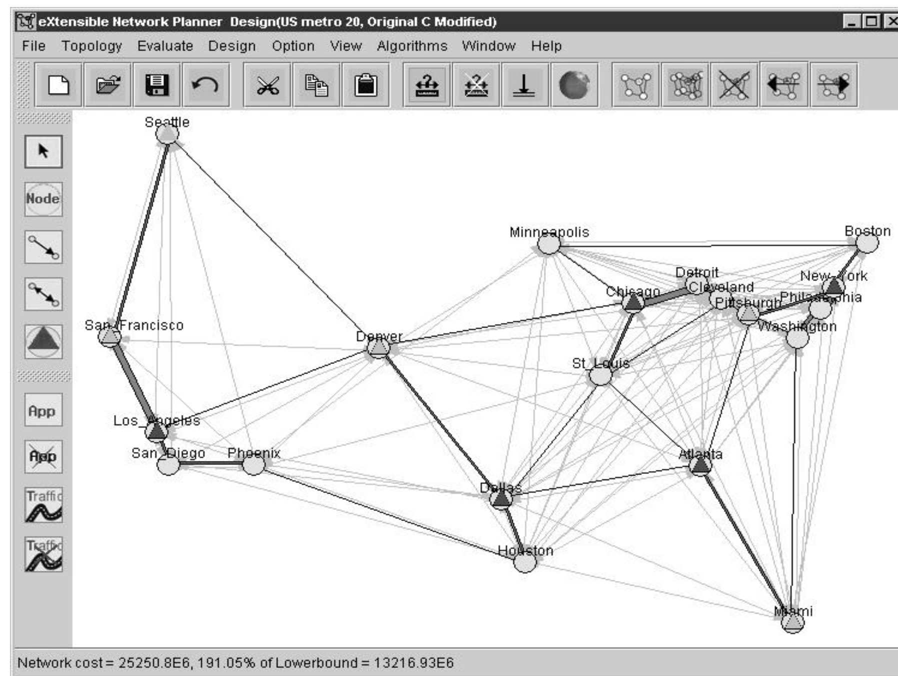


Figure 6.14: Modified hand-crafted topology, Placement C

Figure 6.14 shows the combination of the modified hand-crafted topology and placement C. Note that placement C, among other placements, provided the lowest cost when applied to the original hand-crafted topology. Now, the new combination reduces the cost to 1.91 times the lower bound. For a complete comparison of all

enhancements discussed in this section, Figure 6.15 compares the costs of the designs when each of placements A, B, C, and D was applied to the original and modified hand-crafted topologies. that we started from the hand-crafted topology and place-

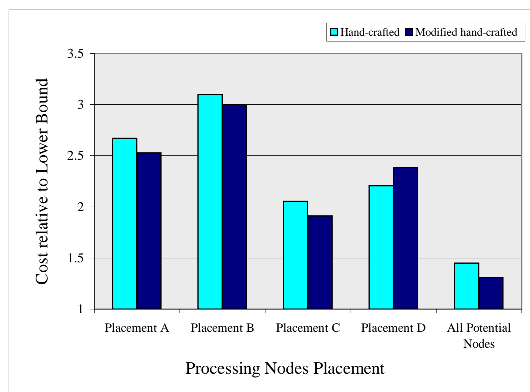


Figure 6.15: Comparison of different placements on hand-crafted and modified hand-crafted topologies for the US network

ment A, which incurred a cost that was 2.67 times the lower bound, we have effectively brought the cost of our design down by 28%.

We apply a similar process to the Western European network by first selecting the hand-crafted topology as the starting point. In this case, however, because the topology does not satisfy the 2-connectedness criterion, we add two more links, (Lisbon, Barcelona) and (Rome, Athens), prior to any enhancement attempt. Next, in order to measure the impact of the processing locations on the design cost, we activate every potential processing location given in the design space. Recall that Figure 6.10(a) showed that the hand-crafted topology approximately costs 2.61 times the lower bound. As shown in Figure 6.16, the cost decreases to 1.45 times the lower bound when all possible locations are selected for processing, indicating that the current processing placements A and B, can be modified to produce a lower cost.

As in the US network, we apply two additional placements:

- **Placement C**: London, Paris, Rome, Barcelona, Berlin

- **Placement D**: London, Ruhr, Rome, Barcelona, Berlin

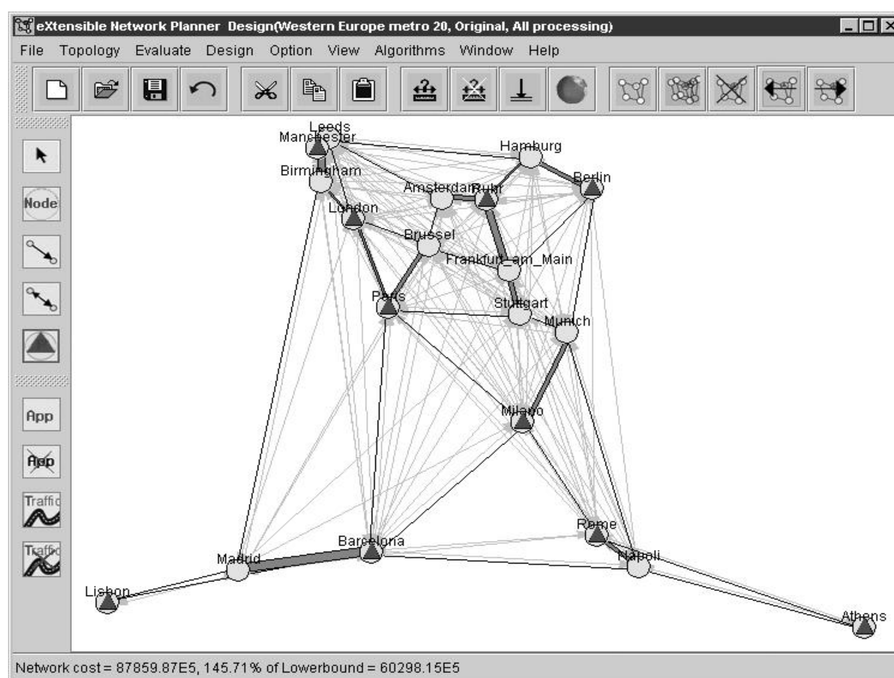
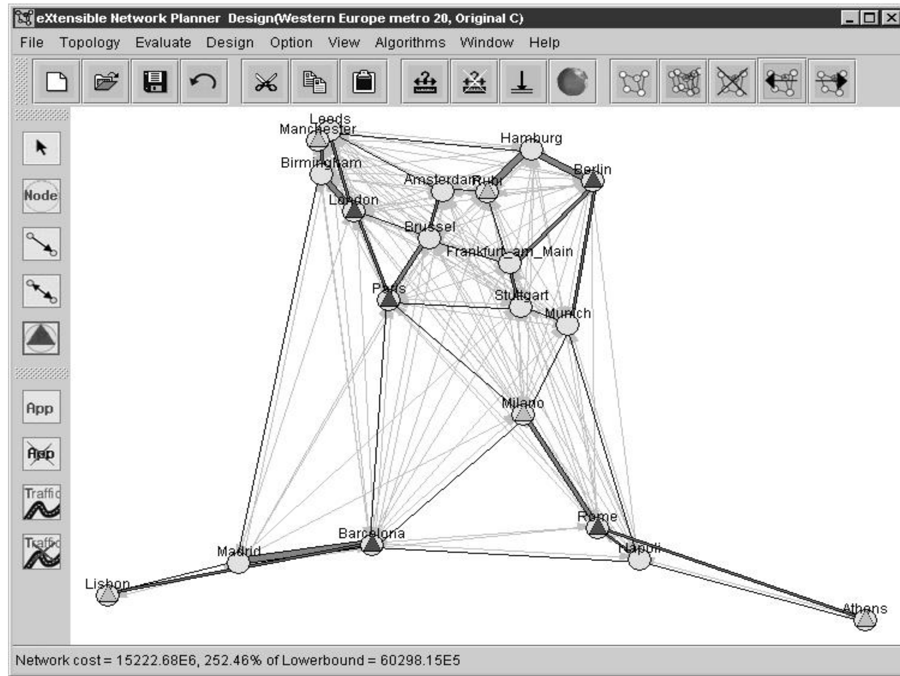


Figure 6.16: hand-crafted topology, processing at all possible locations

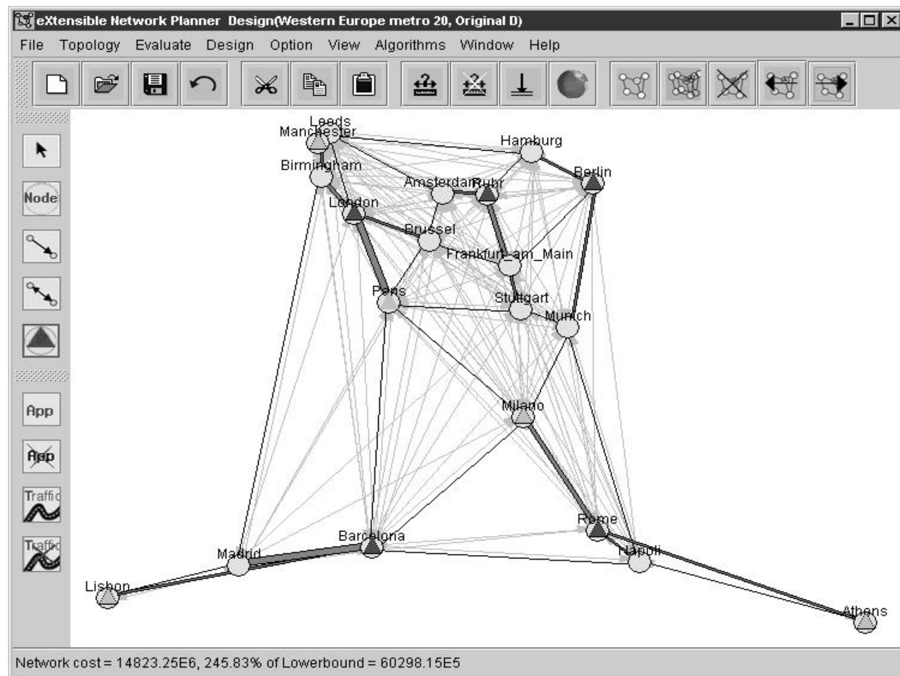
As shown in Figure 6.17(a) and 6.17(b), the new placements reduce the cost of the network only by 5.7% (placement D), much less than in the US network. A natural processing placement policy would be to select locations so that more video receivers can reduce their bandwidth usage. The problem, however, is more complicated in the Western Europe network, where demographic distribution is rather evenly spread out, in contrast to the US network, where the population is heavily concentrated on the coasts. All placements A, B, C, and D that attempt to distribute the processing locations throughout the network result in similar costs.

Given placement policies, we now modify the topology. This time, we only remove links, as listed in Table 6.4. Despite our effort to simplify the topology by removing seven links, the removal reduces the design cost by only a negligible amount of 2.4% when placement D is applied. Figure 6.18(a) shows the result.

Because the different processing placements and the topology simplification resulted in a rather small scale impact on the design cost, we relax our restriction

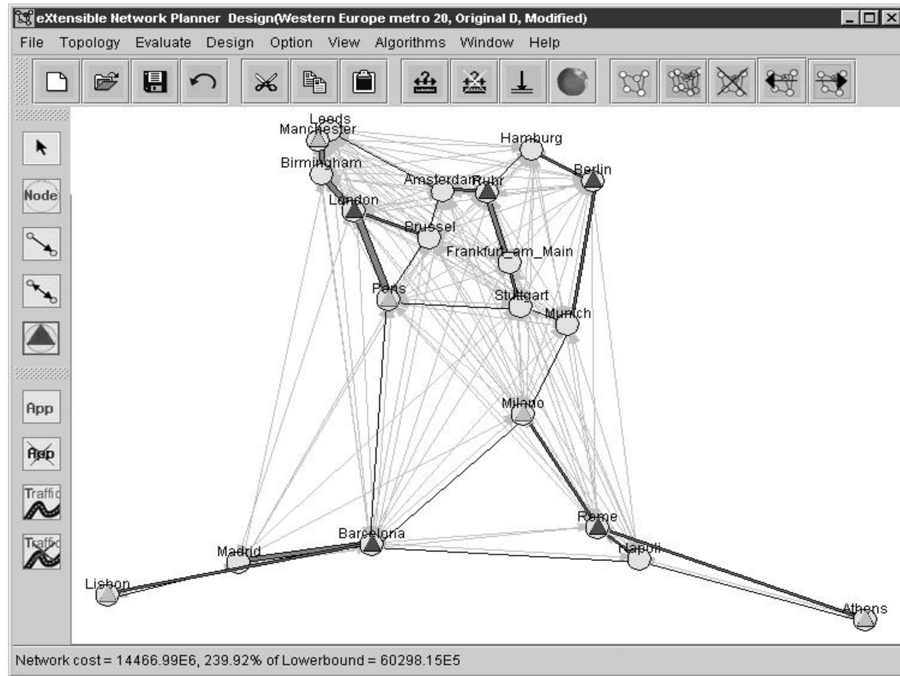


(a) Hand-crafted topology, Placement C

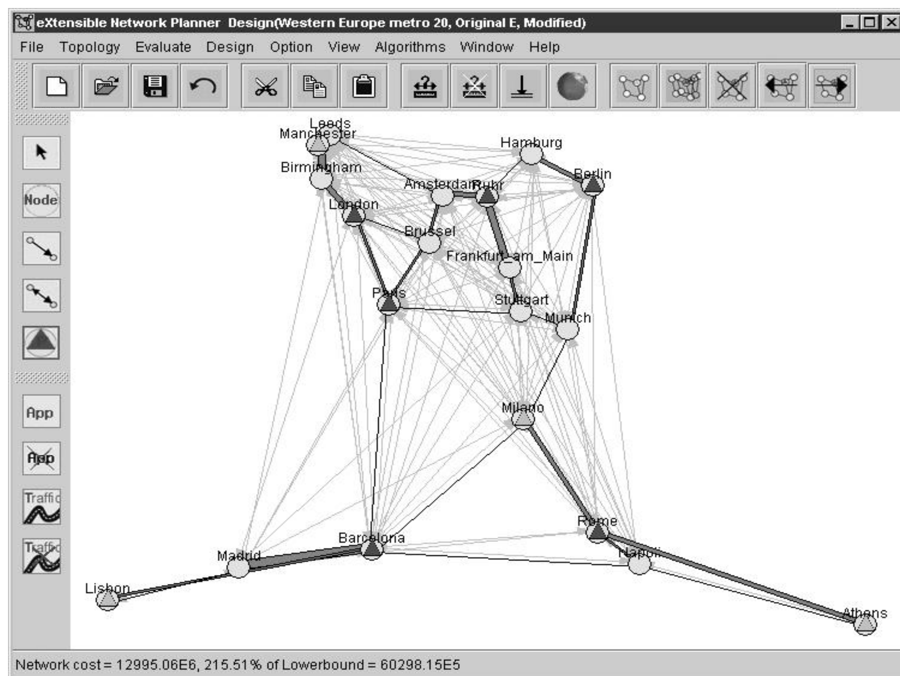


(b) Hand-crafted topology, Placement D

Figure 6.17: Networks for the 20 largest metropolitan areas in Western Europe



(a) Modified Hand-crafted topology, Placement D



(b) Modified Hand-crafted topology, Placement E

Figure 6.18: Networks for the 20 largest metropolitan areas in Western Europe

Table 6.4: Links added and removed for the Western Europe Network

Removed Links	Added Links
(Leeds, London) (Leeds, Hamburg) (Birmingham, Madrid) (Paris, Milano) (Brussels, Frankfurt am Main) (Frankfurt am Main, Berlin) (Munich, Napoli)	No link added

on the number of processing locations. Figure 6.18(b) shows a design with six processing locations, placement D with an additional location at Paris. We call this new placement E. Figure 6.19 compares the costs of all combinations of placements and the hand-crafted topologies. By relaxing the restriction, we have reduced the

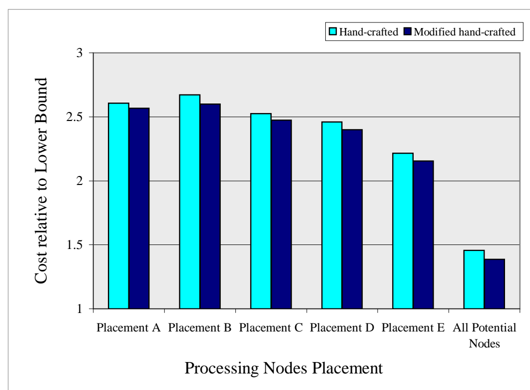


Figure 6.19: Comparison of different placements on hand-crafted and modified hand-crafted topologies for the Western Europe network

cost to 2.16 times the lower bound, which provides approximately 17% cost reduction compared to the cost of the original hand-crafted topology with placement A. This is a significant improvement from the hand-crafted design, which initially gave a cost that is 2.61 times the lower bound. However, the decision to accept the design as a valid solution must be made by evaluating the trade-offs between the actual benefit from the cost reduction and the penalty of using more processing locations.

In this chapter, we have considered two realistic situations to demonstrate how to use XNP to design, evaluate, and refine network configurations. We first generated trial topologies using the automated functions and the graphic interface of XNP, and placed processing resources according to application specifics and traffic distributions. We then evaluated and compared the designs to identify the least cost design. We also discussed how to refine the trial designs further, by adding practical restrictions and also by applying heuristic methods to select topologies and place processing resources.

Chapter 7

Summary and Future Work

The programmability of extensible networks opens up a broad range of ways to develop and operate applications by allowing customized processing or services at network routers. The provision of such services, either by routers or by network-attached processing sites, is potentially a significant benefit for network users, as it can relieve individuals from the need to acquire, install, and maintain software in end systems to perform required services. As such network services become more widely used, it will become increasingly important for service providers to have effective methods for configuring applications sessions so that they use resources efficiently. On the other hand, it is equally important to design such extensible networks properly in order to ensure desirable performance of applications. This dissertation has addressed these two key problems that arise in operating and provisioning extensible networks: configuring application sessions and designing extensible networks.

7.1 Session Configuration in Extensible Networks

First, we investigated the problem of configuring application sessions in extensible networks, which is complicated by the processing to be applied to the data flows of the sessions at intermediate nodes on the route. The processing not only requires additional (processing) resources to be configured on behalf of the sessions, but also

restricts the session routes to include such processing resources. Therefore, the configurations for those applications must be composed of a set of resources, including links and processing nodes, that can successfully carry data from the source end node(s) through the required processing components in the network, and deliver it to the destination(s).

As shortest path algorithms are not directly applicable to our session configuration problem, we have developed a generic method called the *layered graph*. This method introduces a new graph space derived from the original network graph and processing locations, and transforms the original configuration problem to a conventional shortest path problem in the new space. We proved the efficiency and scalability of the method by the simplicity of transforming the problem and the complexity of the shortest path algorithm in the new space. We have shown, through a series of examples, that the *layered graph* can be applied to a wide variety of different situations, including applications with multiple processing steps, applications whose processing can alter data bandwidth as a result, and single-source multicast applications. We have also considered applications that require reserved capacity, and studied several heuristic algorithms based on the *layered graph* method, including *link capacity tracking*, a novel extension of Dijkstra's shortest path algorithm. To account for capacity constraints, *link capacity tracking* attempts to resolve the resource contention that occurs when resources with limited capacity are used multiple times on a single route. Our simulation results demonstrate that the *link capacity tracking* algorithm matches the best performance that one can expect to achieve.

To make our ideas for configuring application sessions directly applicable, it will be necessary to automate the methodology, so that resource configuration software can automatically determine the best way to configure a session to satisfy its requirements. The requirements for reaching this objective include developing a general way of specifying application requirements for intermediate processing, one that is expressive enough to describe typical application scenarios, while being simple enough for application programmers to use effectively. This issue of application specification

was studied by Keller *et al.* in their active pipe model [53], and is worthy of further investigation. The other essential requirement for automating session configurations is a resource allocation system that provides the up-to-date status of the network topology and the resource availability of links and processing components of routers. This status information is necessary for the resource configuration software to identify available resources and to determine the best set of resources among them to configure application sessions. The resource allocation system must also control the resource allocation and deallocation for individual sessions. The general approaches taken by the PNNI protocol or the OSPF protocol can be extended to handle the demands of resource management and allocation.

7.2 Constraint-based Design of Extensible Network

The second problem examined by this dissertation is designing extensible networks, with the focus on supporting candidate applications that require processing at intermediate nodes. Such applications require an additional type of resources at network nodes to perform the customized processing. They must also use the existing link resources in more dynamic ways, because the processing can vary bandwidth in the midst of the session routes. To design extensible networks, network designers must first specify how resources are used by each of the potential applications, and then configure networks with resources that accommodate the anticipated traffic demands and the resource usage of the applications associated with the traffic.

We have introduced a methodology that generalizes the constraint-based network design methods originally developed for conventional networks. We have shown how to incorporate arbitrary application requirements in a flexible and general way, and also have shown how to extend the original framework to dimension both processing resources and link bandwidth. Particularly, the resource dimensioning problem has been formulated as a linear program computing the resource capacity that is necessary and sufficient, given traffic demands for the candidate applications. For a

specific subset of traffic demand patterns, we have introduced a method based on the maximum flow problem in graph theory. We have also shown how to compute lower bounds on the best possible network designs using the linear programming. These results have been incorporated into a software package, the Extensible Network Planner (XNP). XNP supports interactive experimentation with alternative designs by allowing designers to conveniently specify their traffic constraints and application formats, by automating the dimensioning of candidate network designs, and by providing lower bounds that help the designers evaluate the quality of their designs.

The most important direction for future research in this area is to automate the determination of the best topology and set of processing nodes. While simple topology generators can be used to quickly create candidate topologies for evaluation, they do not directly take into account the characteristics of the traffic constraints. In conventional network design, the traffic constraints largely determine the best topology, with star networks being best for some types of traffic constraints, and complete networks being best for other types of constraints. Extensible network design introduces the flexibility to support various application formats, which further complicates topology generation and additionally requires the set of processing nodes to be determined. The generality of traffic constraints and application formats makes it difficult to see how to automatically generate the best topology and to identify the best set of processing nodes for a given set of constraints and application formats. Nevertheless, it seems clear that the design of constraint-driven topology generators and processing node locators is the key research challenge for extensible network planning.

A useful direction for the topology generation and processing nodes placement would be iterative improvement methods that make local changes to a given topology in an effort to get a better one. Simulated annealing could be applied here to make more systematic and effective changes. In another method discussed in Chapter 6, the design process can start with a number of topologies generated by well-known algorithms to focus the design process on a smaller set of preferred topologies for

further improvement. One can enhance this method to see if differences between lower bounds and trial networks can be analyzed to lead more directly to better designs.

References

[1] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vicente, and Daniel Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, April 1999.

[2] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[3] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), 1996.

[4] Jacobus E. van der Merwe, Sean Rooney, Ian Leslie, and Simon Crosby. The Tempest—A practical framework for network programmability. *IEEE Network Magazine*, 12(3):20–28, 1998.

[5] Michael W. Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *International Conference on Functional Programming*, pages 86–93, 1998.

[6] Michael W. Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of IEEE Infocom*, 1999.

[7] B. Schwartz, W. Zhou, A. Jackson, W. Strayer, D. Rockwell, and C. Partridge. Smart packets for active networks. In *Proceedings of IEEE OPENARCH*, 1998.

[8] D. Scott Alexander, Marianne Shaw, Scott Nettles, and Jonathan M. Smith. Active bridging. In *Proceedings of ACM SIGCOMM*, pages 101–111, 1997.

[9] Elan Amir, Steven McCanne, and Randy H. Katz. An active service framework and its application to real-time multimedia transcoding. In *Proceedings of ACM SIGCOMM*, pages 178–189, 1998.

[10] Daniel Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernard Plattner. A scalable, high performance active network node. *IEEE Network*, January/February 1999.

[11] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach, 2nd Edition.* Morgan Kaufmann Publishers, 2000.

[12] Gary Malkin. *RIP Version 2*, November 1998. RFC 2453.

[13] John Moy. *OSPF Version 2.* IETF Network Working Group, April 1998. RFC 2328.

[14] ATM Forum Technical Committee. *Private Network-Network Interface Specification Version 1.0*, March 1996.

[15] Roche A. Gurin and Ariel Orda. Qos routing in networks with inaccurate information: theory and algorithms. *IEEE/ACM Transactions on Networking (TON)*, 7(3):350–364, 1999.

[16] Q. Ma and P. Steenkiste. Quality-of-service routing for traffic with performance guarantees, 1997.

[17] Hongzhou Ma, Inderjeet Singh, and Jonathan Turner. Constraint based design of atm networks, an experimental study. *Washington University Computer Science Department Technical Report WUCS-97-15*, 1997.

[18] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. A flexible model for resource management in virtual private networks. In *Proceedings of ACM SIGCOMM*, 1998.

[19] Andrew J. Fingerhut. Approximation algorithms for configuring nonblocking communication networks. In *Washington University Computer Science Department doctoral dissertation*, May 1994.

[20] J. Andrew Fingerhut, Subhash Suri, and Jonathan Turner. Designing least-cost nonblocking broadband networks. *Journal of Algorithms*, pages 287–309, 1997.

[21] Anupam Gupta, Jon M. Kleinberg, Amit Kumar, Rajeev Rastogi, and Bulent Yener. Provisioning a virtual private network: a network design problem for multicommodity flow. In *ACM Symposium on Theory of Computing*, pages 389–398, 2001.

[22] Inderjeet Singh. Cappuccino: An extensible planning tool for constraint-based atm network design. In *Washington University Computer Science Department masters dissertation*, May 1997.

[23] Russ Miller and Quentin F. Stout. Algorithmic techniques for networks of processors. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 46. CRC Press, 1999.

[24] Tom Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, San Mateo, CA, 1992.

[25] Ivan Hal Sudborough and Burkhard Monien. Embedding one interconnection network in another. *Computing Supplement*, 7:257–282, 1990.

[26] Ralph Keller, Sumi Choi, Dan Decasper, Marcel Dasen, George Fankhauser, and Bernhard Plattner. Active router architecture for multicast video distribution. *Proceedings of IEEE Infocom 2000*, March 2000.

[27] Gabriel Robins and Alexander Zelikovsky. Improved steiner tree approximation in graphs. In *Symposium on Discrete Algorithms*, pages 770–779, 2000.

[28] S. Ramanathan. Multicast tree generation in networks with asymmetric links. *IEEE/ACM Transactions on Networking*, 4(4):558–568, 1996.

[29] Moses Charikar, Chandra Chekuri, Toyat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, , and Ming Li. Approximation algorithms for directed steiner problems. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 192–200, January 1998.

[30] Prashant Chandra, Allan Fisher, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Resource customizable management for value-added customizable network service. *IEEE Network*, 15(1), January 2001.

[31] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NPCompleteness, [SP5].* W.H. Freeman and Company, 1979.

[32] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill Book Company, 1990.

[33] Aaron Kershenbaum. *Telecommunications network Design Algorithms.* McGraw-Hill Book Company, 1993.

[34] Andrews and Zhang. The access network design problem. In *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[35] Baruch Awerbuch and Yossi Azar. Buy-at-bulk network design. In *IEEE Symposium on Foundations of Computer Science*, pages 542–547, 1997.

[36] Sudipto Guha, Adam Meyerson, and Kamesh Munagala. A constant factor approximation for the single sink edge installation problems. In *ACM Symposium on Theory of Computing*, pages 383–388, 2001.

[37] Charlie Scott, Paul Wolfe, Mike Erwin, Andy Oram (Editor), and Scott Wolfe Erwin. *Virtual Private Networks.* O'Reilly & Associates, 1998.

[38] David B. Shmoys, va Tardos, and Karen Aardal. Approximation algorithms for facility location problems (extended abstract). In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 265–274. ACM Press, 1997.

[39] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice-Hall, 1993.

[40] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311, 1984.

[41] Raimund Seidel. Linear programming and convex hulls made easy. In *Proceedings of the sixth annual symposium on Computational geometry*, pages 211–215. ACM Press, 1990.

[42] Bernard Chazelle and Jiří Matoušek. On linear-time deterministic algorithms for optimization problems in fixed dimension. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1993.

[43] Dorit S. Hochbaum. *Approximation Algorithms for NP-hard Problems.* PWS Publishing, 1996.

[44] Sugih Jamin, Cheng Jin, Anthony R. Kurc, Danny Raz, and Yuval Shavitt. Constrained mirror placement on the internet. In *INFOCOM*, pages 31–40, 2001.

[45] P. Krishnan, Dan Raz, and Yuval Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, October 2000.

[46] Bo Li, M. Golin, Giuseppe F. Italiano, Xin Deng, and Kazem Sohraby. On the optimal placement of web proxies in the internet. In *INFOCOM*, 1999.

[47] Sherlia Shi and Jonathan Turner. Placing servers in overlay networks. In *International Symposium on Perform ance Evaluation of Computer and Telecommunication Systems (SPETS)*, 2002.

[48] Moses Charikar, Sudipto Guha, Eva Tardos, and David B. Shmoys. A constant-factor approximation algorithm for the k -median problem (extended abstract). In *ACM Symposium on Theory of Computing*, pages 1–10, 1999.

[49] Fabián A. Chudak. Improved approximation algorithms for uncapacitated facility location. *Lecture Notes in Computer Science*, 1412, 1998.

[50] Guha and Khuller. Greedy strikes back: Improved facility location algorithms. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1998.

[51] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York, NY, 1985.

[52] Arthur H. Robinson, Joel L. Morrison, Phillip C. Muehrcke, A. Jon Kimerling, and Stephen C. Guptill. *Elements of Cartography.* John Wiley & Sons, 1995.

[53] Ralph Keller, Jeyashankher Ramamirtham, Tilman Wolf, and Bernhard Plattner. Active pipes: Service composition for programmable networks. In *Proceedings of IEEE Milcom 2001*, October 2001.

Vita

Sumi Y. Choi

Date of Birth	December 30, 1971
Place of Birth	Chapel Hill, North Carolina
Degrees	**B.S.** Mathematics, February 1994 **M.S.** Computer Science, May 1999 **D.Sc.** Computer Science, December 2003
Professional Societies	Institute of Electrical and Electronics Engineers

Publications

1. Choi S. and Turner J. Configuring Sessions in Programmable Networks with Capacity Constraints, *Proceedings of IEEE ICC*, May 2003

2. Choi S. and Shavitt Y. Proxy Location Problems and their Generalizations, *International Workshop on New Advances of Web Server and Proxy Technologies (in conjunction with IEEE ICDCS)*, May 2003

3. Choi S., Turner J. and Wolf T. Configuring Sessions in Programmable Networks, *Computer Networks*, **41**(2): 269–284, February 2003

4. Wolf T. and Choi S. Aggregated Hierarchical Multicast for Active Networks, *Proceedings of IEEE MILCOM*, October 2001

5. Choi S., Turner J. and Wolf T. Configuring Sessions in Programmable Networks, *Proceedings of IEEE INFOCOM*, April 2001

6. Keller R., Choi S., Decasper D., Dasen M., Fankhauser G. and Plattner B. An Active Router Architecture for Multicast Video Distribution, *Proceedings of IEEE INFOCOM*, March 2000

7. Choi S., Decasper D, DeHart J., Keller R., Lockwood J., Turner J. and Wolf T. Design of a Flexible Open Platform for High Performance Active Networks. *Proceedings of the Allerton Conference*, October 1999

8. Decasper D., Parulkar G., Choi S., DeHart J., Wolf T. and Plattner B. A Scalable High-Performance Active Network Node, *IEEE Network*, **13**(1), January/February 1999

December 2003