Link failure recovery among dynamic routes in telecommunication networks

by

Dieter Stapelberg

Thesis presented in partial fulfilment of the requirements for the degree of Master of Commerce in Computer Science at Stellenbosch University

Department of Mathematical Sciences, Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisor: Prof. A.E. Krzesinski

December 2009

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: ....

Abstract

Link failure recovery among dynamic routes in telecommunication networks

D. Stapelberg
Department of Mathematical Sciences, Computer Science Division,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MComm (Computer Science)

December 2009

Since 2002 data traffic has overtaken voice traffic in volume [1]. Telecom / Network operators still generate most of their income carrying voice traffic. There is however a huge revenue potential in delivering reliable guaranteed data services. Network survivability and recovery from network failures are integral to network reliability. Due to the nature of the Internet, recovery from link failures needs to be distributed and dynamic in order to be scalable.

Link failure recovery schemes are evaluated in terms of the survivability of the network, the optimal use of network resources, scalability, and the recovery time of such schemes. The need for recovery time to be improved is highlighted by real-time data traffic such as VoIP and video services carried over the Internet.

The goal of this thesis is to examine existing link failure recovery schemes and evaluate the need for their extension, and to evaluate the performance of the proposed link failure recovery schemes.

Uittreksel

Netwerk skakel herstelling tussen dinamiese roetes in telekommunikasie netwerke.

("Link failure recovery among dynamic routes in telecommunication networks")

D. Stapelberg

*Departement Wiskundige Wetenskappe, Rekenaarwetenskap Afdeling,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.*

Tesis: MComm (Computer Science)

Desember 2009

Sedert 2002 het data verkeer die stem verkeer in volume verbygesteek [1]. Telekommunikasie / netwerk operateurs genereer egter steeds die meeste van hul inkomste met stem verkeer. Netwerk oorlewing en die herstel van netwerk mislukkings is integraal tot netwerk stabiliteit. Die samestelling van die Internet noodsaak dat die herstel van skakel mislukkings verspreid en dinamies van natuur moet wees.

Die herstel-skema van skakel mislukkings word evalueer in terme van die oorleefbaarheid van die netwerk, die mees effektiewe benutting van network bronne, aanpasbaarheid, en die herstel tydperk van die skema. Die vinnig moontlikste herstel tydperk word genoodsaak deur oombliklike data verkeer soos "VoIP" en beeld dienste wat oor die Internet gedra word.

The doel van hierdie tesis is om bestaande skakel mislukking herstel skemas te evalueer, en dan verder ondersoek in te stel na hul uitbreiding. Daarna word die voorgestelde skakel mislukking skema se effektiwiteit gemeet.

Acknowledgements

I am thankful to Prof. A.E. Krzesinski whose pursuit of answers was an inspiration to me. His knowledge and guidance proved invaluable. My gratitude also to Johannes Göbel for the discussions and work done on Successive Survivable Routing. Lastly to my parents, whose support brought me to this point.

Dedications

Hierdie tesis word opgedra aan my ouers. Baie dankie vir die kompas en die kaart.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of the most basic human desires is the need to communicate. With the advent of the Internet, a new and powerful medium of communication has been realised. The growth of the Internet has been exponential [2], as shown in Figure 1.1.

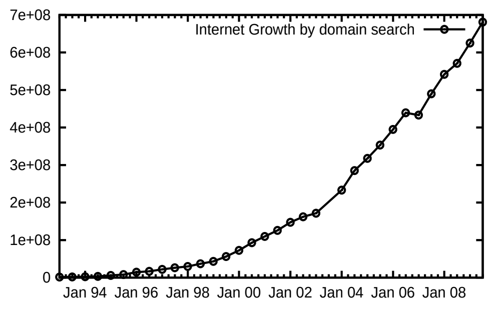

Figure 1.1: Internet growth by domain search.

The large-scale deployment of Voice over IP (VoIP) has resulted in the even more rapid growth of data traffic over the Internet. This creates an opportunity to increase the revenues earned from data traffic by optimizing the network carrying the data traffic. With more of this carried traffic being critical services, in addition to the increase in the overall volume of traffic carried over the Internet, recovery from network failures has become an important aspect of network engineering.

In both our personal and business life, we have become more and more dependent on the array of communication services at our disposal. Communication networks are subject to a variety of errors. Human error, overload and natural disasters are examples of unintentional failures. Intentional disruptions include maintenance actions and sabotage. Communication networks play an integral part of our daily lives. Commercial uses of communication networks are critical to business functions. This has led to availabilty (uptime) guarantees forming an integral part of service level agreements (SLAs) between service providers and their customers. Telecommunication and Internet companies require sound strategies to create a successfull business model using the Internet as a platform. Network reliability is arguably the most critical factor for Internet-enabled business.

No network is free from failures. Network *integrity* is the ability of the network to provide the desired quality of service (QoS), not only in normal network conditions, but also when network congestion or network failure occurs.

Network survivabilty is a subset of network integrity. Survivability refers to the ability of a network to recover in the event of failure, with little or no consequence to the user. Link failure recovery can take a long time (recovery times are measured in milliseconds) in traditional computer networks, in some cases even a few minutes.

This thesis first provides a background to IP architecture as it is relevant to network recovery. We then examine existing networking routing protocols, and focus on the edge-disjoint open shortest path first (ED-OSPF) routing algorithm. We briefly examine Multi-Protocol Label Switching (MPLS) traffic engineering (TE) which has enjoyed considerable success, which has led to the development of MPLS TE recovery techniques that require our attention. We next discuss existing network recovery mechanisms, and we investigate the Successive Survivable Routing (SSR) algorithm as part of the failure recovery mechanism.

We propose a recovery scheme which involves the development of link failure recovery algorithms that can recover from any link failure in the network model. Furthermore, the failure of multiple links (large-scale disaster) needs to be investigated. Failure recovery schemes should therefore deploy recovery routes that are link-disjoint to the failed route. We next study the effects of our proposed link failure recovery scheme against the criteria that we will set.

The outcome of this work is an effective scheme to recover from link failures, and the optimal provisioning of network spare capacity to recover from these failures. The goal is to minimize the effect of such failures on the network, and to return the network to the state prevailing before the network failure as soon as possible. We run simulations on sets of network topologies to measure the performance of the network recovery scheme. We also look at the network engineering required to improve the performance of the algorithm.

Simulation results are only useful if sensible input is provided to the simulation. The network model is a crucial "parameter" supplied to the simulator. We attempt to create more realistic network models using ISP provided network specifications, data obtained from the Rocketfuel project [3], and artificially generated network topologies [4].

Performing numerous simulation runs can be a tedious and time-consuming task. To accelerate this process we built a Beowulf cluster-computer which enables the parallel execution of simulation runs, and greatly reduces the elapsed time of simulations, as well as the coordination overhead required.

Chapter 2

Network theory

We begin with an introduction and discussion of the Internet Protocol (IP), which is a vital component in the structure of the Internet and is used by the Internet hosts and the routers to communicate. Routing in the network forms an integral part of network recovery and hence survivability. SSR, for example, uses IP (actually OSPF) to find the first set of recovery routes and then uses different cost metrics to discover more optimal recovery routes. We also present some relevant graph theory which lays the foundation for our discussion on link-state routing algorithms.

2.1 The Internet Protocol (IP)

The Internet Protocol (IP) is used for communicating data across a packet-switched internetwork. Together with the Transmission Control Protocol (TCP), it is the backbone of Internet data transfer. IP is concerned with two basic functions namely addressing and packetization.

Packetization is concerned with splitting the data to be transmitted into smaller packages, or datagrams. We are more interested in the addressing function of IP. From RFC 791 [5]:

> ...The Internet modules use the addresses carried in the Internet header to transmit Internet datagrams toward their destinations. The selection of a path for transmission is called routing ...
>
> ...In addition, these modules (especially in gateways) have procedures for making routing decisions and other functions.

These routing decisions are made by consulting routing tables that are maintained in each router. In order to make these routing decisions we need a method for finding paths in the network and distributing this path information among the routers. This is the function of the link-state OSPF algorithm discussed in Section 2.3 on Page 6.

Link-state Interior Gateway Protocols (IGPs) have a proven track record over many years. They are deployed in the majority of networks and form the foundation of the Internet. Link state routing has also had renewed interest for network survivability due to the fast recovery properties of link state routing protocols [6]. Refer to the IP routing recovery cycle in [6] as it is defined on the timeline.

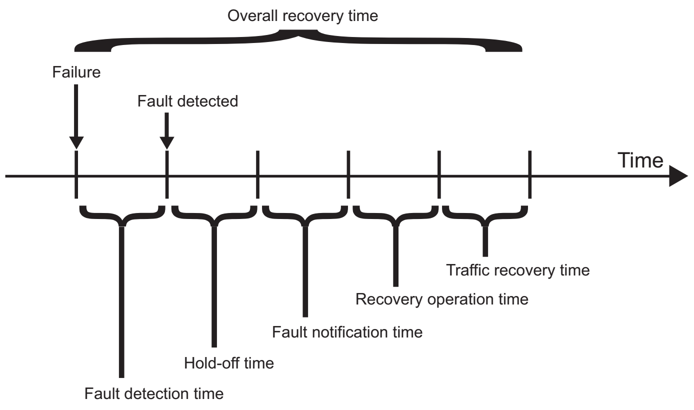

Figure 2.1: The IP recovery cycle.

OSPF operates as follows in reference to the recovery timeline in [6]:

1. **Fault detection and characterization**. Early detection of faults in the network has a considerable impact on the total recovery time.
2. **Hold-off timer**. In modern networks it is likely that there exist multilayer recovery mechanisms. This means that is appropriate for the IP layer to wait for the lower network layers to determine if their own recovery mechanisms can manage to recover from the fault. If the recovery time for the lower layer is defined as X, we need to define a hold-off timer Y for the IP layer, with $Y > X$.
3. **Fault notification time**. When a link or node failure occurs, the nodes directly connected to the point of failure will detect the error. They will issue a fault indication signal (FIS) to the nodes that they are connected to. Each node in turn that receives a FIS forwards it to its neighbours. OSPF uses a process called Link State Announcement (LSA) flooding to ensure that the FIS eventually reaches each connected node in the routing domain.
4. **Recomputation of the routing table**. Each node has to compute a new routing table using the updated network topology state information. In this process each node uses OSPF to create the Shortest Path Tree (SPT). The SPT contains the next node on the shortest path from this node to every other node in the routing domain.

5. **Route using the new Routing Information Base (RIB).** Once the SPT computation is complete each node will update its RIB and traffic that was routed on a failed path will now be routed on a recovery path.

2.2 The network graph

A graph is used to represent a computer network. A graph $G = (V, E)$ is a set of vertices (nodes) V connected by a set of edges (links) E. The nodes in a computer network can be computers, routers, switches or any other networking hardware device. The edges that connect these nodes are the communication links used to establish these connections. These could be Ethernet network connections, dial-up network connections, fiber-optic network connections, satellite connections, and so on.

Consider the network graph in Figure 2.2 below. Due to its topological layout it is referred to as the fish network graph.

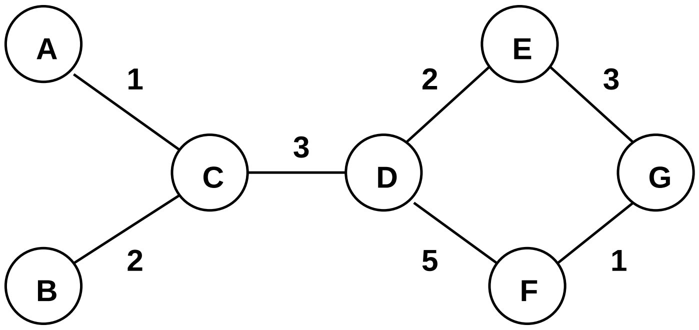

Figure 2.2: The fish network graph.

- The set of nodes in this network is $V = \{$A, B, C, D, E, F,G$\}$.
- The number of nodes in the network is denoted by $|V|$ (in this network $|V| = 7$).
- The set of links in this network is $E = \{$AC, BC, CD, DE, DF, EG, FG$\}$.
- The number of edges in the network is denoted by $|E|$ (in this network $|E| = 7$).
- The degree (v) of a node indicated by $d_G(v)$ defines the number of edges connecting to this node (in this network network $d_G(D) = 3$).

An edge is defined as a bi-directional link not necessarily of equal cost in both directions. We can now represent a real computer network as a network graph.

To route information between nodes in the network we need to know the paths which connect the nodes. We are concerned with a method of finding these paths in the network. We refer to the starting point (node) of a path as the origin node, and the end point (node) of the path as the destination node.

A path from a node i to a node j is a sequence of contiguous edges that connect these nodes. Consider the example in Figure 2.2 above. There exist two paths for the origin-destination (O-D) pair A and G. We can use a short-hand notation to indicate this O-D pair as (AG). We can denote the path for AG as a sequence of edges: (AG) = ACDEG or (AG) = ACDFG.

The next question is how to find these paths. OSPF is a standard link-state routing algorithm, which we discuss next in Section2.3.

2.3 Open Shortest Path First

Before the advent of the Internet, networks were smaller entities, which became known as Autonomous systems (AS). Routing within an AS was done by the Routing Information Protocol (RIP) [7]. A large contributing factor to RIP's success was its inclusion in the widely used Berkeley Standard Distribution (BSD) UNIX. However RIP, a distance vector protocol, suffered from several deficiencies [8]:

- slow convergence,
- routing loops,
- the "counting to infinity" problem,
- the "small infinity" problem,
- some of the problems above were solved by restricting paths to be no longer than 15 hops, which is a problem in itself.

As networks grew bigger, these flaws were exposed. The Internet Engineering Task Force (IETF) formed a working group in 1988 to develop a new link-state routing algorithm. This link-state algorithm was known as shortest path first (SPF). Their mandate was to design an Interior Gateway Protcol (IGP) based on the SPF algorithm. This algorithm is also known as Dijkstra's algorithm, named after its creator, Edsger W. Dijkstra. Dijkstra's own recollection of the creation of the SPF algorithm [9] is candid and surprising:

> I designed my first nontrivial algorithms. The algorithm for the Shortest Path was designed for the purpose of demonstrating the power of the ARMAC at its official inauguration in 1956, the one for the Shortest Spanning Tree was designed to minimize the amount of copper in the backpanel wiring of the X1. In retrospect, it is

> revealing that I did not rush to publish these two algorithms: at that time, discrete algorithms had not yet acquired mathematical respectability, and there were no suitable journals. Eventually they were offered in 1959 to"Numerische Mathematik" in an effort of helping that new journal to establish itself. For many years, and in wide circles, the Shortest Path has been the main pillar for my name and fame, and then it is a strange thought that it was designed without pencil and paper, while I had a cup of coffee with my wife on a sunny cafe terrace in Amsterdam, only designed for a demo...

The first version of OSPF was described in RFC 1131 [5] and published in 1989. It was replaced by OSPF v2 in RFC 1247 [5], published in 1991. OSPF v2, often referred to as OSPF, is in use today.

The SPF algorithm finds the shortest path in the network between each node pair. It does this by incremental search. We start with Dijkstra's algorithm as defined in Figure 1 [10]. A working example is provided to illustrate Dijkstra's algorithm.

2.3.1 Dijkstra's algorithm

Denote the source node by A and the destination node as Z. Dijkstra's algorithm is given in Algorithm 1.

```
Data: A connected network graph $V$
Result: The shortest path from A to Z.
  1. Start with $d(A) = 0$.

     $d(i) = \begin{cases} \ell(A) & \text{if } i \in \Gamma_A, \\ \infty & \text{otherwise } (\infty \text{ is a large number as defined below}). \end{cases}$

     $d(i) \equiv$ the distance of node $i$ ($i \in V$) from the source node $A$, which is
     the sum of arcs in a possible path from node $A$ to node $i$.
     $\Gamma_i \equiv$ the set of neighbouring nodes of node $i$.
     $\ell(ij) =$ length of the arc from node $i$ to node $j$.
     Assign $S = V - \{A\}$. Assign $P(i) = A \, \forall i \in S$.
  2. a) Find $j$ such that $d(j) = \min(d(i))$, $i \in S$.
     b) Set $S = S - \{j\}$
     c) If $j = Z$ (the destination node), END; otherwise go to step 3
  3. $\forall\, i \in \Gamma_j$ and $i \in S$, if $d(j) + \ell(ji) < d(i)$, set
     $d(i) = d(j) + \ell(ji), P(i) = j$.
     Go to step 2.
```

Algorithm 1: Dijkstra's algorithm

The symbol of ∞ (also refered to as INF) is a large
number used to initialise the distance of each of the nodes (excluding source node A) from node A. The value of INF must be larger than the length of the path to be determined. The shortest path length is not known *a priori*, as it is the output of the algorithm. We therefore set INF equal to or greater than the longest path between A and Z. We define INF to be greater than the sum of the lengths of the edges in the given graph, $G = (V, E)$.

Consider the following example of the OSPF algorithm applied to the network presented in 2.3:

In the above network we wish to find the shortest path from node A to node G. The steps performed by the algorithm are as follows:

Table 2.1: A working example of Dijkstra's algorithm

Step	**Details**
1.	$d(A) = 0$ $d(B) = 1, d(C) = 1, d(E) = 5, d(D) = d(F) = d(G) = \infty$ $S = \{B, C, D, E, F, G\}$ $P(B) = P(C) = P(D) = P(E) = P(F) = P(G) = A$

Continued on the next page

Table 2.1 – continued from the previous page

Step	Details
2.	$\min\{d(B), d(C), d(D), d(E), d(F), d(G)\} = d(B) = 1$ (a) $j = B$ (b) $S = \{C, D, E, F, G\}$ (c) $j = B \neq G$
3.	$\Gamma_B = \{A, C, D\}$ $S = \{C, D, E, F, G\}$ $\Gamma_B \cap S = \{C, D\}$ $d(C) = d(B) + \ell(BC) = 1 + 3 = 4$ $\Rightarrow 4 > d(C) = 1$ $d(D) = d(B) + \ell(BD) = 1 + 5 = 6$ $\Rightarrow 6 < d(D) = \infty$ $\Rightarrow d(D) = 6,\ P(D) = B$
2.	$\min\{d(C), d(D), d(E), d(F), d(G)\} = d(C) = 1$ (a) $j = C$ (b) $S = \{D, E, F, G\}$ (c) $j = C \neq G$
3.	$\Gamma_C = \{A, B, D, E, F\}$ $S = \{D, E, F, G\}$ $\Gamma \cap S = \{D, E, F\}$ $d(D) = d(C) + \ell(CD) = 1 + 2 = 3$ $\Rightarrow 3 < d(D) = 6$ $\Rightarrow d(D) = 3, P(D) = C$ $d(E) = d(C) + \ell(CE) = 1 + 3 = 4$ $\Rightarrow 4 < d(E) = 5$ $\Rightarrow d(E) = 4,\ P(E) = C$ $d(F) = d(C) + \ell(CF) = 1 + 1 = 2$ $\Rightarrow 2 < d(F) = \infty$ $\Rightarrow d(F) = 2,\ P(F) = C$
2.	$\min\{d(D), d(E), d(F), d(G)\} = d(F) = 2$ (a) $j = F$ (b) $S = \{D, E, G\}$ (c) $j = F \neq G$
3.	$\Gamma_F =$ {C,D,E,G} $S = \{D, E, G\}$

Continued on the next page

Table 2.1 – continued from the previous page

Step	Details
	$\Gamma_F \cap S = \{D, E, G\}$ $d(D) = d(F) + \ell(FD) = 2 + 4 = 6$ $\Rightarrow 6 > d(D) = 3$ $d(E) = d(F) + \ell(FE) = 2 + 1 = 3$ $\Rightarrow 3 < d(E) = 4$ $\Rightarrow d(E) = 3,\ P(E) = F$ $d(G) = d(F) + \ell(FG) = 2 + 2 = 4$ $\Rightarrow 4 < d(G) = \infty$ $\Rightarrow d(G) = 4,\ P(G) = F$
2.	$\min\{d(D), d(E), d(G)\} = d(D) = 3$ (a) $j = D$ (b) $S = \{E, G\}$ (c) $j = D \neq G$
3.	$\Gamma_D = \{B, C, F\}$ $S = \{E, G\}$ $\Gamma_D \cap S = \{\}$
2.	$\min\{d(E) = 3, d(G) = 4\} = d(E) = 3$ (a) $j = E$ (b) $S = \{G\}$ (c) $j = E \neq G$
3.	$\Gamma_E = \{$A,C,F,G$\}$ $S = \{G\}$ $\Gamma_E \cap S = \{$G$\}$ $d(G) = d(E) + \ell(EG) = 3 + 1 = 4$ $\Rightarrow 4 \equiv d(G) = 4$
2.	$\min\{d(G)\} = d(G) = 4$ (a) $j = G$ (b) $S = \{\}$ (c) $j = G \Rightarrow$ END

Using the predecessor list P that we built up in the above example we extract the path from node A to node G as ACFG.

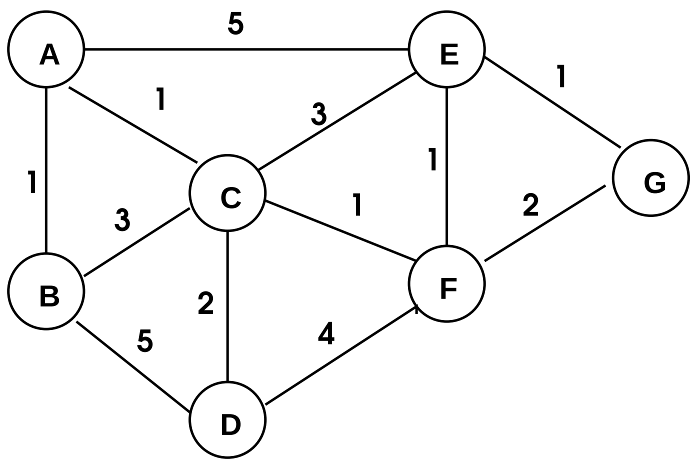

Figure 2.3: An undirected graph.

2.3.2 The modified Dijkstra's algorithm

In the following Section 2.4 on Page 12, we discuss the method of finding an optimal edge-disjoint pair of paths for any given O-D pair [10]. In the construction of these optimal edge-disjoint path pairs we will make use of graphs with negative arcs. An arc is a directional link between nodes. A bi-directional edge is composed of two arcs. AB refers to the arc from node A to node B and BA refers to the arc from node B to node A. These two arcs AB and BA together form the bi-directional edge between the nodes A and B. It is possible to assign different costs to the link between two nodes, dependent on its direction. For example, we can assign a positive cost of 5 to the arc AB and a negative cost of -3 to the arc BA. The usefulness of negative arcs will be illustrated in Section 2.4 on Page 12. Dijkstra's algorithm as discussed in Section 2.3.1 cannot work with negative arc costs. It therefore requires the modification as detailed in Algorithm 2.

1. Start with $d(A) = 0$,

$$d(i) = \begin{cases} \ell(A) & \text{if } i \in \Gamma_A, \\ \infty & \text{otherwise } (\infty \text{ is a large number as defined below}). \end{cases}$$

   $\Gamma_i \equiv$ set of neighbour nodes of node i, $\ell(ij) =$ length of arc from node i to node j.
   Assign $S = V - \{A\}$
   Assign $P(i) = A \, \forall i \in S$.

2. a) Find $j \in S$ such that $d(j) = \min(d(i))$, $i \in S$.
   b) Set $S = S - \{j\}$
   c) If $j = Z$ (the destination node), END; otherwise go to step 3

3. $\forall \, i \in \Gamma_j$, if $d(j) + \ell(ji) < d(i)$, set
   a) $d(i) = d(j) + \ell(ji), P(i) = j$.
   b) $S = S \bigcup \{i\}$.

   Go to step 2.

Algorithm 2: The modified Dijkstra algorithm

There are two changes in the modified Dijkstra algorithm. First, in Step 3, the modified Dijkstra algorithm scans all the neighbours of the node selected in Step 2(a). This is necessary as a node that was previously "permanently" labelled can now be relabelled since it may receive a lower label upon further scanning of the graph. Secondly, step 3(b) re-enters any node that was relabelled in Step 3(a) into the set S.

Note that for nonnegative arcs these steps become redundant and the modified Dijkstra algorithm reduces to the Dijkstra algorithm.

2.4 Optimal edge-disjoint path pair

In the Section 2.3 on Page 6 we discussed the Dijkstra and the modified Dijkstra algorithm. These algorithms are concerned with finding the shortest path between any O-D pair in the network. It is likely that more than one path exists between any O-D pair in the network. Not all of these paths will be independent of each other, meaning that they will share common edges. We define a pair of paths to be edge disjoint if they have no edges in common. The pair of edge-disjoint paths with the least total path length is defined to be the shortest pair of paths.

Before we examine the shortest edge disjoint pair of paths algorithm, we examine a simple algorithm to find a pair of paths for an O-D pair, the two-step-approach algorithm.

2.4.1 The two-step-approach (2SA) algorithm

The 2SA algorithm uses Dijsktra's algorithm to find the shortest path. The edges contained in this path are then removed from the graph (by setting their cost to infinity) and Dijkstra's algorithm is run again to find the new shortest path in the modified graph. We begin with the network graph as given in Figure 2.4 below:

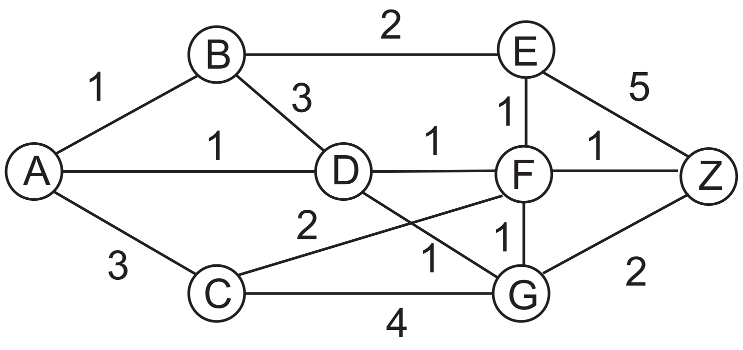

Figure 2.4: The two-step-algorithm network.

Running Dijkstra's algorithm we find the shortest path for AZ to be ADFZ. The next step is to remove the edges contained in the first path from the network graph. This produces the network graph given in Figure 2.5 below. Running Dijkstra's algorithm again, we find the second path as ABEFGZ. The cost for the first path ADFZ is 3. The cost for the second path ABEFGZ is 7. The combined cost for the 2SA algorithm shortest edge-disjoint pair of paths is 10.

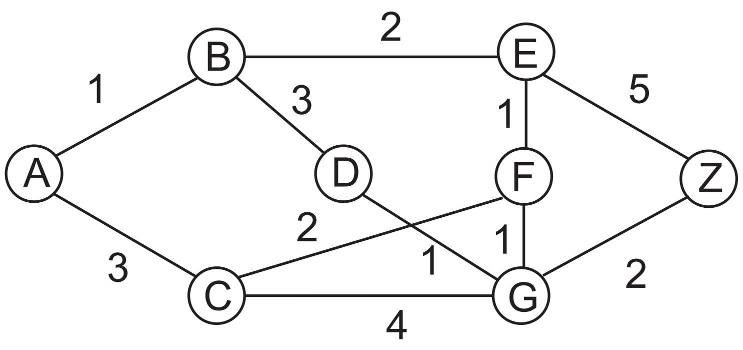

Figure 2.5: The two-step-algorithm fails to find the shortest pair of paths.

Note that using the edge-disjoint shortest pair path algorithm, which we discuss in Section 2.4.2 on Page 14, we find that the shortest pair of paths is ABEFZ and ADGZ, with a total path cost of $5 + 4 = 9$.

This example shows that the 2SA algorithm is suboptimal. Furthermore the 2SA algorithm can fail to find a pair of edge-disjoint paths, even when they exist. Figure 2.6 below provides such an example. For the remaining network graph examples we assume that the straight line from node A to node Z (which is the O-D pair for which we find the path(s)) is the shortest path between the O-D pair. This is represented by the dashed line in Figure 2.6. After the first run of Dijkstra's algorithm for the 2SA algorithm the links AB, BC, and CZ are removed. This leads to the second run of Dijkstra's algorithm for the 2SA algorithm which now is not able to find a path from A to Z. The network graph in Figure 2.6 is now disconnected. With the ED-OSPF algorithm, as indicated by the two dotted outer lines, the paths ADCZ and ABEZ with a total cost of 6 are found as the shortest pair of paths for O-D pair AZ.

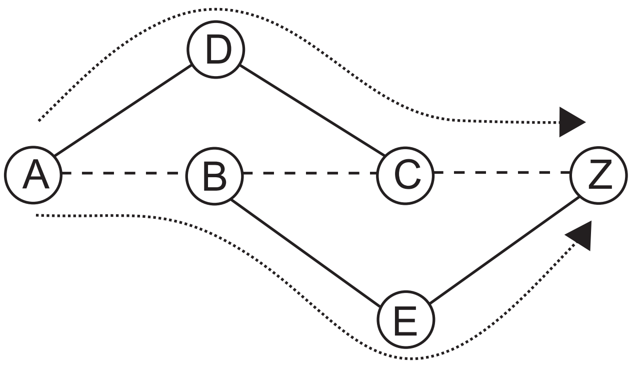

Figure 2.6: The two-step-algorithm fails to find a pair of paths.

2.4.2 Optimal edge-disjoint path pair development

We begin the development of the ED-OSPF algorithm by examining the set of possible paths that can deliver the shortest path pair.

Let path s denote the shortest path between the designated O-D pair. For now we will assume that s is unique. We define a segment of the path s as any contiguous subset of arcs on the path. For example, if the shortest path s is path ABCDEFGHZ, the segments BCDE and FGH are segments of the path s. Let

$$S_p = \text{ the set of all paths whose segments overlap with path } s, \tag{2.4.1}$$

$$S'_p = \text{ the set of all paths whose segments do not overlap with path } s \tag{2.4.2}$$

and let

- s_p denote a path $\in S_p$,
- s'_p denote a path $\in S'_p$,
- (γ_1, γ_2) denote a pair of paths γ_1 and γ_2.

From 2.4.1 and 2.4.2 we see that there are 6 possible combinations for (γ_1, γ_2):

$$(\gamma_1, \gamma_2) \in \{(s, s'_p), (s_p, s_p), (s'_p, s'_p), (s_p, s'_p), (s, s_p), (s, s)\} \tag{2.4.3}$$

The requirement for edge disjointness rules out (s, s_p) and (s, s).

We can also rule out (s'_p, s'_p), (s_p, s'_p), since these path choices will always be longer than (s, s'_p), due to the uniqueness assumption made for the shortest path s.

Next we look at the (s, s_p) combination as shown in Figure 2.7 below:

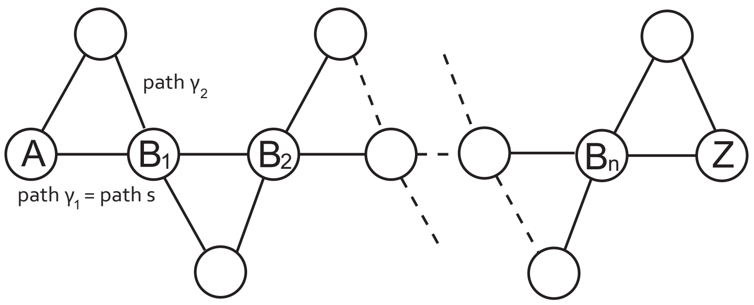

Figure 2.7: The (s, s_p) case; path γ_1 is path s, the straight line from A to Z. Path s'_p intersects path s at n vertices, but does not share any edges with path s.

Path γ_2 by definition cannot contain any segment of path s. Therefore the sum of the lengths of the two paths γ_1 and γ_2 is greater than twice the length of path s. This leaves:

$$(\gamma_1, \gamma_2) \in (s_p, s_p) \tag{2.4.4}$$

We now define each of the paths in (s_p, s_p) in 2.4.4 to contain a single (but exclusive) segment of path s. This leads to the the following 3 subcases:

1. One of the segments of the two paths terminates at one of the endpoint nodes.
2. Neither of the two segments terminates at an endpoint node.
3. Both of the segments terminate at an endpoint node.

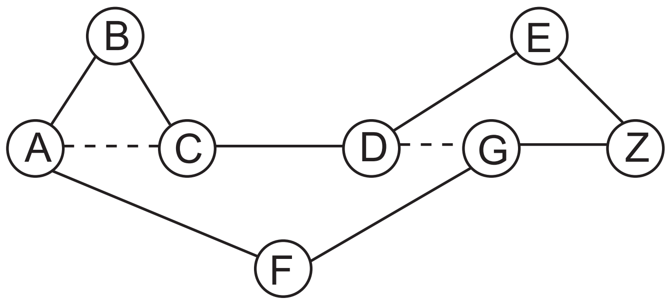

Figure 2.8: Case 1 of the (s_p, s_p) case.

We wish to show that the two paths constituting the shortest edge-disjoint pair intersect each other at a node on path s. We will prove this by contradiction. Figures 2.8 to 2.11 illustrate the cases where the pair of paths do not intersect each other. The first case is shown below:

Nodes A and C, which are on path s are connected via node B. The path ABC is longer than the direct path AC. This means that the path pair ACDEZ and AFGZ is shorter than the pair ABCDEZ and AFGZ.

The same argument can be applied for the following two cases in Figures 2.9 and 2.10 below:

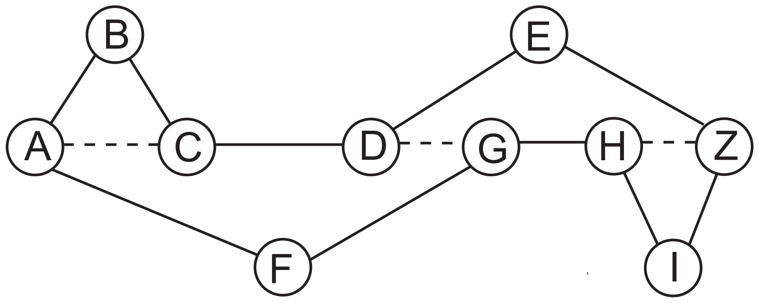

Figure 2.9: Case 2 of the (s_p, s_p) case.

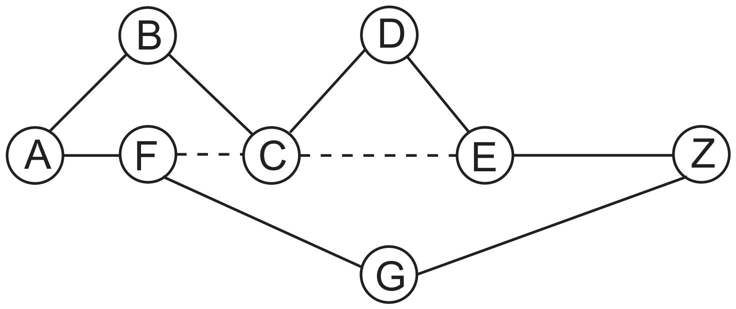

Figure 2.10: Case 3 of the (s_p, s_p) case.

Figure 2.11 needs further consideration. Each of the alternate paths is longer than the shortest path s. Thus the sum of these two paths is greater than twice the cost of path s. This is the same as the (s, s'_p) case which we have already ruled out. Also note that there is a single break DC in this graph. If we generalise this, there can be m breaks, with $m > 0$. We will come back to this in Figure 2.12 on Page 2.12. This figure illustrates this general case, with $m = 3$.

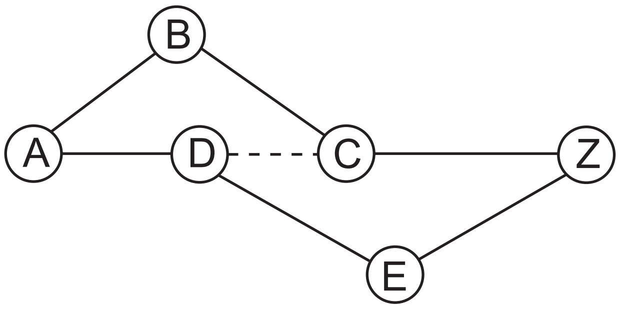

Figure 2.11: Case 4 of the $s_p \times s_p$ case.

Now we relax the constraint that the paths cannot intersect at a node on path s. This would mean that the node on path s where the paths intersect has a minimum degree of 4. For Figures 2.8 and 2.9 we combine nodes D and G into node D, with $d(D) = 4$. In Figure 2.10 we combine nodes C and F into node C, with d(C) = 4. The same arguments before combining the vertices still hold to make this configuration invalid.

Figure 2.11 can be modified into a valid configuration. If we combine vertices C and D into node C, with $d(C) = 4$, we have a valid configuration as it is equivalent to the $(s,\ s'_p)$ configuration.

This gives the shortest edge-disjoint pair representation as:

$$(\gamma_1, \gamma_2)_{\text{edge-disjoint shortest pair}} \in \{(s_p, s_p), (s, s'_p)\} \tag{2.4.5}$$

Furthermore, the two paths may intersect at any number of nodes on path s and also contain segments of path s so that there exist breaks in path s. Figure 2.7 shows the $(s,\ s'_p)$ case that occurs when all the segments of path s are part of the two edge-disjoint shortest pair paths.

Next we relax the constraint of the shortest path uniqueness. This allows for additional configurations, as in Figures 2.8, 2.9 and 2.10.

These configurations occur with the same total path cost for the pair of paths. Consider Figure 2.8; if path ABCDGZ has the same cost as as path s (ACDGZ), then the pair of paths (ABCDEZ, AFGZ) has the same path cost as (ACDEZ, AFGZ). This is the same as the configuration discussed for Figure 2.11.

The cases (s_p, s'_p) and (s'_p, s'_p) in equation 2.4.4 also have to be taken into consideration again. Once again, the configurations would occur with the same total path cost as the paths in the (s, s'_p) case.

In the case of more than one shortest path, it is clear that these additional configurations are of the same total length as the configurations based on the unique shortest path constraint. Therefore, in the construction of the shortest pair edge-disjoint paths, the configurations given by 2.4.5 are applicable. From the previous argument it follows that s can be either a unique path or one of several shortest paths selected by the path finding algorithm.

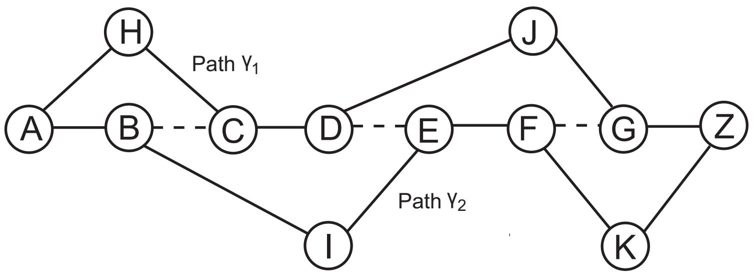

Figure 2.12: A (s_p, s_p) case with $m = 3$.

Consider Figure 2.12 above which corresponds to the $(s_p,\ s_p)$ configuration. When the breaks in Figure 2.12 – BC, DE and FG – are reduced to zero it is equivalent to the $(s_p,\ s'_p)$ configuration in Figure 2.7. We consider Figure 2.12 as the general configuration that includes the $(s,\ s'_p)$ case.

This means that any edge-disjoint shortest pair path algorithm that generates and optimises over all configurations as generalised in Figure 2.12 produces the shortest pair of edge-disjoint paths.

Referring back to the two-step algorithm, it corresponds only to the $(s,\ s'_p)$, and as already shown in the example, it is suboptimal and could possibly not find disjoint paths in a network graph even if they do exist.

2.4.3 The optimal edge-disjoint path pair algorithm

We can now state the algorithm for generating the edge-disjoint shortest pair path [10] as in Algorithm 3.

We illustrate this algorithm by using the same network example as in Figure 2.6 on Page 14. We include link costs to illustrate the graph modifications made by the ED-OSPF algorithm.

We now find the ED-OSPF paths as follows:

1. Let node A be the source node, and node Z the destination node. The OSPF shortest path is ABCZ.

2. Remove arcs BA, BC and ZC replace edges AB, BC and CZ respectively.

1. Select any of the two nodes as the source node and the other as the destination node. Use the modified Dijkstra algorithm (see algorithm 2) to find the shortest path for this O-D pair.

2. Replace each edge of the shortest path by a single arc directed to the source node. This means that the arc for each link on the shortest path from the source to the destination must be set to infinity.

3. Negate the length of the above arcs in the direction of the destination node to the source node.

4. Run the modified Dijkstra algorithm again on the graph with the modified arcs, and find the shortest path again for the same origin and destination nodes.

5. Revert the arcs modified in steps 2 and 3 back to obtain the original graph, and erase any interlacing edges of the two paths. The interlacing edges are defined as arcs that exist in both the first and second shortest path obtained steps 1 and 4, with the direction of the arc disregarded. Therefore if arc AB exists in the first path, and arc BA exists in the second both are removed from their respective shortest paths. Arrange and alternate the remaining segments based on the interlaced edges that were removed to obtain the shortest pair of edge-disjoint paths.

Algorithm 3: The edge-disjoint shortest path pair algorithm

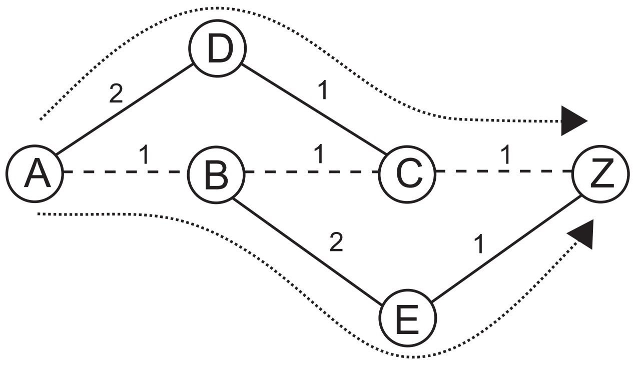

Figure 2.13: The 2SA failure example network with link cost.

3. Set BA = -1, BC = -1, ZC = -1.

4. The shortest path in the modified network graph as in Figure 2.14 below is ADCBEZ.

5. Reverting back to the original network, the two paths ABCZ and ADCBEZ have an interlacing edge BC in common. We remove this inter-

lacing edge as follows:

- The ABCZ path from node A to node B proceeds where we reach the interlacing edge BC. We proceed to node B on the ADCBEZ path. We then proceed from node B to node E, and from node E to node Z on the ADCBEZ path. This gives us the first ED-OSPF path ABEZ.
- The ADCBEZ path proceeds from node A to node D, then from node D to node C. Here we reach the interlacing edge BC. We proceed to node C on the ABCZ path. We then proceed from node C to node Z on the ABCZ path. This gives us the second ED-OSPF path ADCZ.

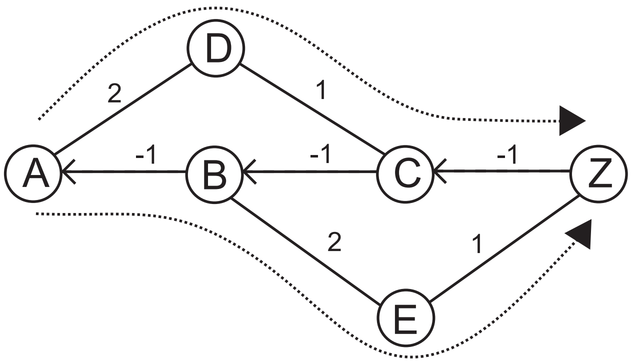

Figure 2.14: The modified network with negative costs.

2.5 Teletraffic Engineering

Traffic engineering is one of the most important elements of network design. In the field of network and telecommunications it is referred to as teletraffic engineering (TE). TE uses statistical techniques to predict and respond to the behaviour of telecommunication networks. The main goal is to route traffic so network resources are "efficiently" used.

The definition of "efficiently" depends on the perspective taken on teletraffic engineering. From the end-user perspective, we wish to minimise delays and avoid congestion in the network. In terms of the network operator, they could wish to route specific traffics along certains paths / segments which have the appropriate hardware and / or links to deal with this traffic.

From [11] the objective of teletraffic theory can be formulated as follows: "to make the traffic measurable in well defined units through mathematical models and to derive relationships between grade-of-service and system capacity in such a way that the theory becomes a tool by which investments can be planned."

For reasons of limited space, we refer the reader to references [11], [12] which provides more detailed information on teletraffic engineering.

2.6 Multipath Label Switching Protocol (MPLS)

In IP routing, discussed in Sections 2.1 and 2.3, teletraffic engineering is made possible by the manipulation of the link costs. Due to its widespread use many IP TE engineering techniques have been developed. Other networks use TE methods as well (public voice, ATM, and frame relay). We focus on Multi Path Label Swithching (MPLS) TE in this section and , [6], [13], [14], [15] will provide more background on MPLS and MPLS TE.

The need has arisen to have more control of the routing process, which in the case of IP TE is dynamic and distributed. MPLS was developed with this in mind. MPLS TE is a "tunneling" mechanism that establishes TE Label Switch Paths (TE LSPs) between origin-destination pairs. In MPLS TE we refer to the origin node as the head-end label switched router (LSR) and the destination node as tail-end LSR. Each TE LSP has its own set of constraints. These constraints, along with the network topology and resources, are used to compute the TE LSP that meets the given requirements. For the purposes of our link failure recovery scheme, the requirements would be rapid and equivalent recovery from network link failure. This requires the immediate availability of provisioned backup paths that can survive any link failure. This will be discussed in Chapter 4 on Page 34.

The path computation methods can be distributed or centralised. Once an LSP is established, the IP packets are routed according to the LSP. The intermediate nodes do not make any routing decisions. LSPs support the notion of link disjointness (also referred to as link diversity), where two LSPs do not have a link in common. Another concept to note is that of Shared risk link groups (SRLG). A SRLG provisions for the fact the failure of any single element (such as hardware failure) can result in the failure of multiple network elements. SRLG disjointness refers to an LSP being either link (L) disjoint or node (N) disjoint, i.e. if its path does not include any link $\ell \in L$, or node $n \in N$ in its path. Likewise two TE LSPs are SRLG disjoint if their respective sets of links do not have any SRLG in common.

2.6.1 MPLS TE components

MPLS TE consists of the following five main components:

1. Configuring the TE LSP on a head-end LSR.
2. Topology and resource information distribution. Note that is only required for distributed / on-line path computation.
3. TE LSP computation. The TE LSP path computation can be performed by an off-line or by an on-line tool. This stage of MPLS TE is where our SSR algorithm, which will be discussed in Chapter 4 on Page 34, can be employed.
4. TE LSP setup. Once a TE LSP is computed, the head-end LSR signals the TE LSP by using the Resource Reservation Protocol (RSVP) [14] signalling protocol as extended for LSP tunnels [15]. It is important to note that paths are not set up using RSVP. The paths are maintained and torn down using different RSVP messages.
5. Packet forwarding. Once the TE LSP is set up by the head-end LSR using RSVP, it updates its routing table and can start forwarding packets on the route.

2.6.2 MPLS advantages

MPLS TE should be considered for the following reasons.

1. Bandwidth optimization: MPLS TE can achieve the network resource utilization goals defined by the network administrator.
2. QoS: Various guarantees can be provided through engineering the network to meet the demands of network services.
3. Fast recovery: Several mechanisms are available that allow for fast recovery, and in particular we will examine MPLS TE Global Path Protection. The existing mechanisms offer fast convergence, but bandwidth optimization and QoS guarantees can also be met.

2.6.3 Recovery Operation Time

Referring to the IP recovery cycle illustrated in Figure 2.1 on Page 4 we now focus on the fourth segment of the cycle, the recovery operation time. We consider three MPLS recovery techniques:

- MPLS TE global default restoration which is the default MPLS recovery mode. This requires that when a failure occurs, the head-end LSR is notified by means of RSVP and the routing protocol. The head-end LSR will recompute a new path and re-signal the TE LSP along the new path.

- MPLS TE global protection. Two TE LSPs are set up by the head-end LSR, a primary LSP and a backup LSP. When a failure occurs and the head-end LSR is notified, it is not required to recompute a new path, the head-end LSR can immediatly start using the backup LSP.
- MPLS TE local protection. This is a local recovery scheme, where upon the occurance of network failure, the LSPs affected by the failure are locally rerouted by the node immediatly upstream from the failure.

2.6.3.1 MPLS Traffic Engineering Global Path Protection

We illustrate the use of MPLS through an example. Figure 2.15 illustrates a 15-node network where we show the paths for two O-D pairs, A-Z and J-Z.

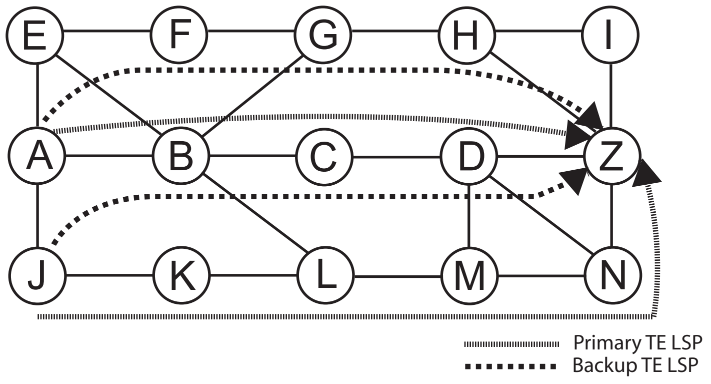

Figure 2.15: MPLS TE Global Path Protection.

The bold dashed line is the primary TE LSP used for routing traffic under normal network conditions. The lighter dashed lines are the backup (secondary) LSPs that will be deployed as backup routes if the primary TE LSP is inoperable due to network failure. Note that the backup LSP is signalled on the same path as the primary path. There is a requirement however that the backup route must be link diverse from from the primary TE LSP. This is necessary since, if any of the links on the primary path fail, then the working path has failed and the backup path must be used. If the backup path makes use of links on the primary path, we can not be sure that these links have not failed. With MPLS TE it is difficult to guarantee that the QoS requirements will be met. The QoS degredation depends on the use of the other LSPs sharing the same network resources as the backup LSP path. In MPLS TE it is also difficult to gaurantee the network bandwidth constraints, unless they have been reserved. This is where the SSR algorithm plays a vital part in our network recovery mechanism, as it provisions the backup capacity while minimising the overall backup capacity required.

From [6] we note the following advantages and drawbacks for MPLS TE Global Path Protection.

Advantages

- The method is easy to deploy in networks with many links and nodes and a limited number of LSPs to protect. Only the necessary diversly routed TE LSPs need to be configured. With local protection for example every network element would need to be protected with a backup path.
- The backup tunnel is computed and signalled before the failure. The advantage is that the path is deterministic and that the network administrator has control over the backup tunnel path. This is especially important where the individual backup tunnels are selected as part of a globally optimised backup path selection.

Drawbacks

- Global path protection requires doubling the number of TE LSPs, which can have a significant impact in full mesh networks. However, in Chapter 3 on Page 25 we show that full mesh networks are not common in "real-world" network topologies, where we see that preferential connectivity follows a power-law distribution when new nodes are connected to the network.
- Global path protection cannot in most cases (especially international networks) recover from network failure in tens of milliseconds, which is critical for time sensitive traffic like voice data. This is due to the fact that failure notification needs to be received by the head-end LSR before switching over to the pre-configured backup LSP.
- If a bandwidth guarantee is required, in order to provide bandwidth sharing, path protection requires the use of an external off-line tool for the computation of both the primary and the secondary TE LSPs. SSR in Chapter 4 on Page 34 can provide the tool for this requirement.
- The requirement for an end-to-end diversely routed path may in some cases imply the selection of a nonoptimal path for the primary TE LSPs. This problem has already been addressed by ED-OSPF, discussed in Section 2.4 on Page 12, where we select the optimal pair of network paths that have the lowest commbined path length. It has also been shown in Section 2.4.1 on Page 13 that the ED-OSPF algorithm succeeds where the 2SA fails.

Chapter 3

Network models and topology generation

Protocols and network behaviours must be investigated and optimised before implementation. Simulation is an important part of network research and is often used to investigate the performance of networks and protocols. The network model or topology with which we run the simulation is a crucial input parameter, and it has an important influence on the outcomes of the simulation.

In this chapter we examine several realistic network topologies obtained from the Rocketfuel project [3] and other sources of real network data. As in [16] we examine "POP-level" (point of presence level) topologies. We examine the BRITE topology generation tool [4] and we create several topologies which are investigated in Section 3.2.2 on Page 33.

3.1 Real world network models

Using the data from [3] we created representations of the AT&T topology of North America, the Ebone topology of Europe, the Tiscali topology of Europe and the Telstra topology of Australia. Each of these models was modified in two ways:

1. The POP-level representation was simplified in cases where multiple POPs are located in the same metropolitan area.

2. The network topologies are subject to a minimum node-degree of two. This is to facilitate network recovery algorithms such as ED-OSPF, discussed in Section 2.4, which require link disjointness.

Table 3.1 presents the notation for the network topology properties that we use to profile our networks.

Table 3.1: Network topology parameters

Number of nodes	N
Number of edges (arcs)	$L(A)$
Average node degree	$\overline{d}$
Average shortest path length	$\overline{p}$
Minimum shortest path length	$\min p$
Maximum shortest path length	$\max p$

3.1.1 North America - the AT&T model

The AT&T North American network model was created from [3] with nodes (POPs) that reside in the USA. The network parameters are given in Table 3.2.

Table 3.2: The AT&T USA model

N	50
$L(A)$	79 (158)
$\overline{d}$	3.16
$\overline{p}$	2.773
$\min p$	1
$\max p$	6

3.1.2 North America - the MCI model

The MCI network model was created by visual inspection of data provided on the ISP's site [17]. This network differs from the AT&T model in that it has international connections to locations such as London, Sao Paulo and Hong Kong. One modification was made (a link from Los Angeles to Seoul) to maintain the minimum node-degree. The network parameters are given in Table 3.3.

Table 3.3: The MCI USA model

N	43
$L(A)$	94 (188)
$\overline{d}$	4.372
$\overline{p}$	2.904
$\min p$	1
$\max p$	6

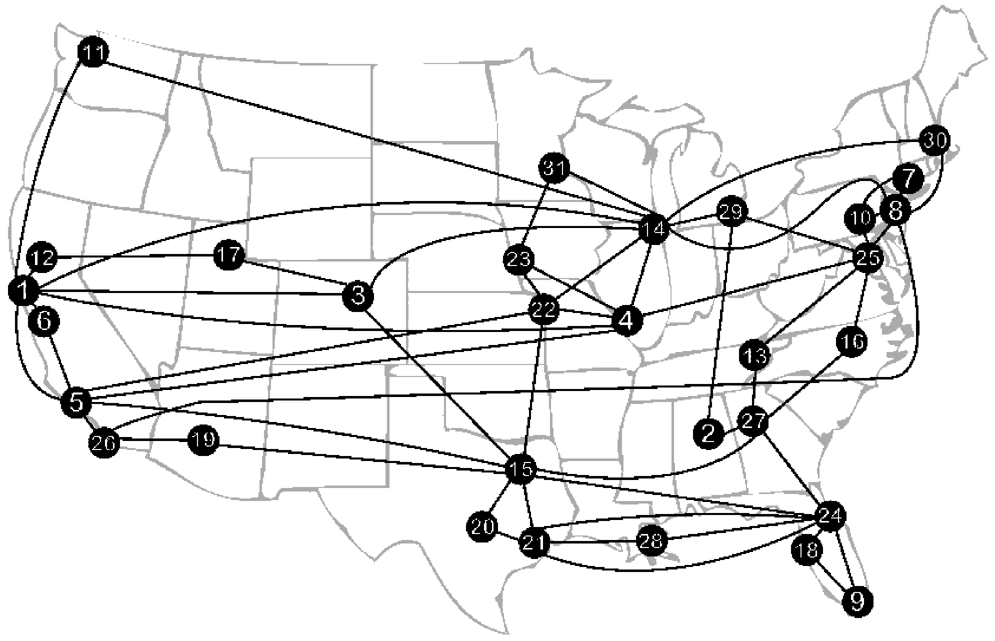

Figure 3.1: The AT&T North American network model.

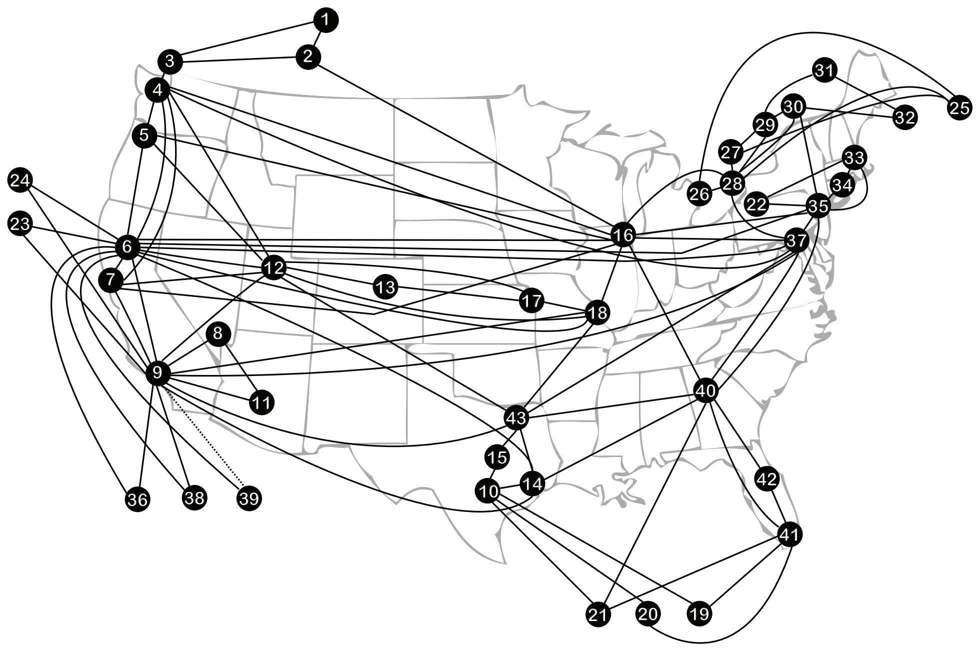

Figure 3.2: The MCI North American network model.

3.1.3 Europe - the Ebone model

The Ebone Europe model was created using the data in [3] with nodes (POPs) that reside in Europe. The network was modified to ensure the minimum node degree. The network parameters are given in Table 3.4.

Table 3.4: The EBone European model

N	26
$L(A)$	48 (96)
$\overline{d}$	3.692
$\overline{p}$	2.942
$\min p$	1
$\max p$	6

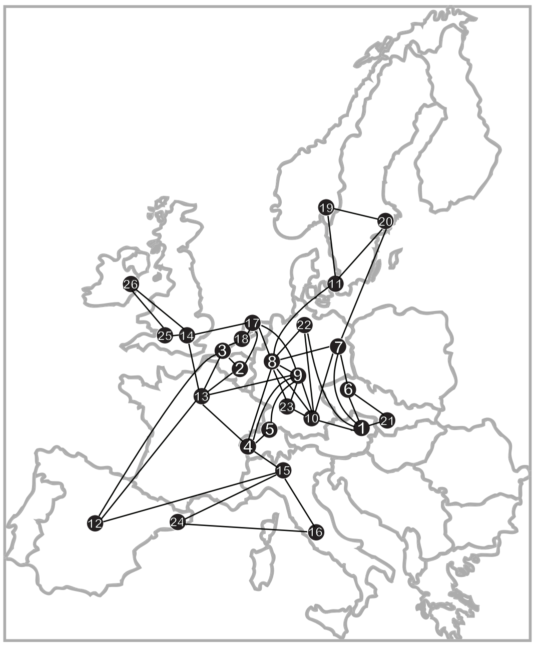

Figure 3.3: The European Ebone network model.

3.1.4 Europe - the Tiscali model

The Tiscali Europe model was created from [3] and, similar to the MCI network model in Subsection 3.1.2, it has connections to New York, Chicago and Washington. To ensure the minimum node-degree an additional connection from London to Chicago was established. The network parameters are given in Table 3.5.

Table 3.5: The Tiscali European model

N	26
$L(A)$	48 (96)
$\overline{d}$	3.692
$\overline{p}$	2.504
$\min p$	1
$\max p$	6

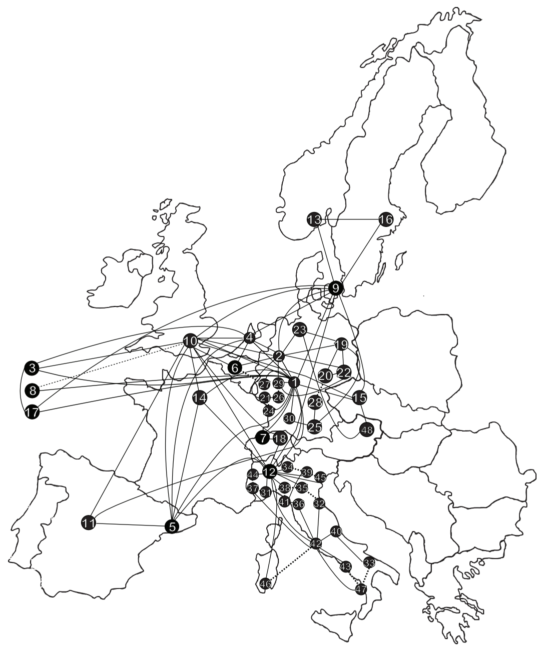

Figure 3.4: The European Tiscali network model.

3.1.5 Australia - the Telstra model

The Telstra Australia model was created in [3]. All of the nodes (POPs) reside in Australia, and it has no internationl links. Links were added to the network to ensure the minimum node degree. The network parameters are given in Table 3.6.

Table 3.6: The Telstra Australian model

N	26
$L(A)$	48 (96)
$\overline{d}$	3.16
$\overline{p}$	3.279
$\min p$	1
$\max p$	6

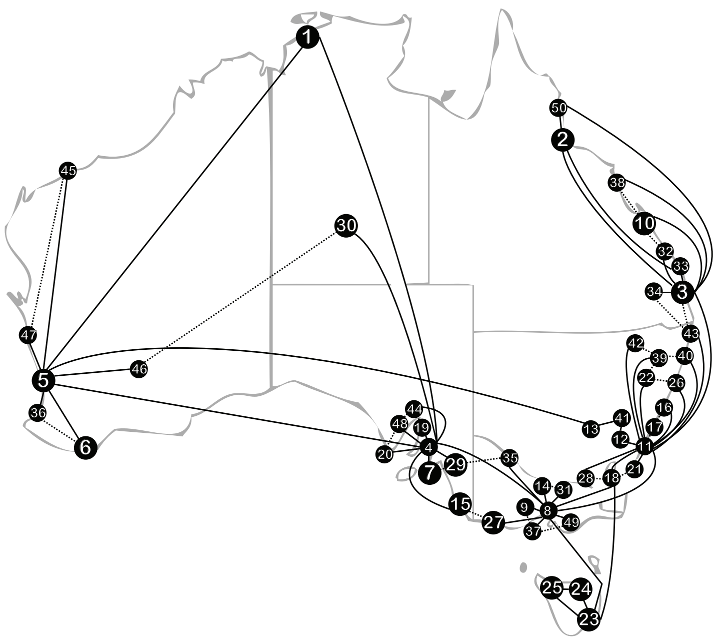

Figure 3.5: The Australian Telstra network model.

3.2 The BRITE network topolgoy generator

Real Internet topologies are not made fully and readily available, if at all. ISPs regard their router-level topologies as confidential information. The Rocketfuel project as discussed in Section 3.1 on Page 25 addresses the lack of realistic router-level network topologies and allowed the development of the "real-world" network topologies, with the exception of the MCI USA network topology, which was built with information published on the ISP's site.

The three main BRITE network topology generator principles [4] are

1. Representativeness - the topologies accurately reflect many aspects of the Internet topology.
2. Inclusiveness - to combine the strengths of as many topology generation models as possible in a single generation tool.

3. Interoperability - to provide interfaces to widely-used simulation and visualisation applications.

The BRITE network topologies will serve as synthetic network topologies for our simulations to compare to realistic and to other example network topologies. Although BRITE offers four main topology types we only focus on the Router Level topologies.

To generate a BRITE router level topology we need to supply values for the parameters as listed in Table 3.7.

Table 3.7: BRITE router level parameters

HS & LS	The size of one side of the plane, and of one side of a high-level square
N	The number of nodes in the network
Model	Waxman or Barabási-Albert
α	Waxman-specific parameter for node interconnecting probability
β	Waxman-specific parameter for node interconneting probability
Node placement	Random or heavy-tailed
m	Number of links per new node (node degree)
Growth type	All or incremental
BWDist	Constant, uniform, exponential or heavy-tailed
MaxBW, MinBW	The maximum and minimum bandwith values

The **HS** and **LS** parameters are important in topologies that use heavy-tailed node-placement as the grouping for these nodes into squares (size given by **LS**) on the plane (size given by **HS**) is employed. In contrast random node placement can be anywhere on the plane (size given by **HS**).

The model types (**Waxman** or **Barabási-Albert**) differ in the method of connecting nodes to the network. The **Waxman** node interconnecting probability for the nodes of the random topology is given by:

$$P(u, v) = \alpha e^{d/(\beta L)} \tag{3.2.1}$$

where $\alpha > 0, \beta \leq 1, d$ is the Euclidian distance from node u to node v, and L is the maximum distance between any two nodes.

The **Barabási-Albert** model suggests two possible causes for the emergence of a power law in the frequency of the outdegrees in network topologies:

- incremental growth - networks are formed by the continual addition of new nodes, and the gradual increase in the size of the network
- preferential connectivity - new nodes connect to existing nodes that are highly connected, which form hubs.

When node i joins the network, the probability that it connects to an existing node j is given by:

$$P(i,j) = \frac{d_j}{\sum_{k \in V} d_k} \tag{3.2.2}$$

where d_j is the outdegree of the target node; V is the set of nodes that have joined the network and $\sum_{k \in V} d_k$ is the sum of the outdegrees of all the nodes that have previously joined the network.

The growth type is important for our topology generation. Incremental growth states that a node has a higher probability of linking to nodes with more existing network connections. We oberve this in our "real-world" Australia model in Section 3.1.5 on Page 29 where the Adelaide, Brisbane, Perth, Melbourne and Sydney nodes have a high degree of connectivity as they act as hubs in the network. Random node placement means a node can connect to any existing node in the network at random. The number of links per new node determines the minimum amount of connections that a new node makes when it connects to the network.

The link bandwith and distribution can largely be ignored for the purposes of our simulations, as we use population-based traffic demands to determine the bandwidth requirements between nodes. We discuss this in Section 3.2.1. The number of nodes is given by **N** and this allows us to choose the size of our artificial network topology.

3.2.1 Population-based traffic and bandwidth requirements

The generated BRITE topologies do not determine the city populations that we use in our network topology models. The power law for the frequency of node degree in [18] has also been observed for city populations [19]. We use a Pareto distribution to create a set of city populations for our BRITE topologies.

We use random samples that are generated using inverse transform sampling. Given a random variate U drawn from the uniform distribution on the unit interval $(0, 1)$, the variate

$$T = \frac{x_m}{U^{1/k}} \tag{3.2.3}$$

with x_m being a positive scale parameter, and k the positive shape parameter, is Pareto-distributed.

3.2.2 Generated BRITE topologies

We begin by generating network topologies comparable to the SSR example networks as discussed in Section 4.4.1 on Page 45.

We created the following networks:

- 10 node, node degree of 4, Waxman connection model (**N10D4DWax**) network topology,
- 10 node, node degree of 4, BA2 connection model (**N10D4BA2**) network topology,
- 20 node, node degree of 2, Waxman connection model (**N20D2DWax**) network topology,
- 20 node, node degree of 2, BA2 connection model (**N20D2BA2**) network topology,
- 30 node, node degree of 2, Waxman connection model (**N30D2Wax**) network topology,
- 30 node, node degree of 2, BA2 connection model (**N30D2BA2**) network topology.

Table 3.8 presents the network profile for these topologies.

Table 3.8: BRITE topologies network profile

Property	**N10D4Wax**	**N10D4BA2**	**N20D2Wax**	**N20D2BA2**	**N30D2Wax**	**N30D2BA2**
N	10	10	20	20	30	30
L (A)	34 (68)	30 (60)	40 (80)	55 (110)	60 (120)	119 (238)
$\overline{d}$	6.8	6	4	5.5	4	7.933
$\overline{p}$	x	x	x	x	x	x
$\min p$	x	x	x	x	x	x
$\max p$	x	x	x	x	x	x

Chapter 4

Successive Survivable Routing

4.1 Review of SSR

The Successive Survivable Routing (SSR) algorithm [20] provides a computationally efficient approximate solution to the Spare Capacity Allocation (SCA) problem, discussed in Section 4.1.2. The purpose of the SSR algorithm is to minimise the total spare (backup) capacity neccessary to provide equivalent recovery routes in the case of link failure. We review the SCA problem, then review and verify the SSR algorithm. We then extend SSR to employ capacity giveback and we also develop a state-dependent version of SSR to further reduce the spare capacity requirement.

4.1.1 A new definition of a flow

Before we discuss the SCA problem, we need to redefine a flow. In [20] a flow is defined as a traffic demand between an origin-destination (O-D) pair. Thus for each additional traffic demand between an O-D pair, a new flow is added to the SSR data structures. With reference to Fig. 4.1, example 3 in [20] illustrates a new flow between the O-D pair AB. The addition of each new flow leads to the growth of the matrices **P**, **Q**, **M**, **D**, **U**, and **T** which are defined below. Furthermore the number of $\mathbf{G}^r$ matrices increases with the number of flows r. These matrices are summarised in Table 4.1 and explained in Sections 4.1.2 and 4.1.3.

We define a flow as the aggregated traffic demand between an O-D pair, thus on a per route basis. New traffic demand on any given O-D pair is therefore aggregated into the existing traffic demand for the given O-D pair. We provide an example that illustrates the working of the SSR algorithm on a per route basis.

4.1.2 The SCA problem

Using the network graph representation as discussed in Chapter 2 on Page 3 we represent an undirected network graph G, with N vertices (nodes), L edges (links), and R flows. A flow exists between each O-D pair.

We now examine the data structures employed by the SSR algorithm to represent the network configuration and calculate the backup paths and the backup capacities required. Let m_r denote the bandwidth requirement (flow) of flow r. The working and backup paths of flow r are given by the $1 \times L$ binary row vectors $\mathbf{p}_r = \{p_{r\ell}\}$ and $\mathbf{q}_r = \{q_{r\ell}\}$ respectively where L is the number of links in the network. For each link ℓ that is used in the path for flow r the ℓ-th element of of the vector is set to 1. The transposes of these r row vectors form the path link incidence matrix. This gives two $R \times L$ matrices $\mathbf{P} = \{p_{r\ell}\}$ and $\mathbf{Q} = \{q_{r\ell}\}$.

Let $\mathbf{M}$= Diag($\{m_r\}_{R\times 1}$) be the diagonal matrix that contains the bandwidth demands of the r flows where R is the number of flows. Scaling this matrix allows for partial/additional reservation of spare capacities.

The matrix $\mathbf{B} = (b_{n\ell})_{N\times L}$ represents the node incidence matrix where N is the number of nodes. If node n is either the origin or destination of the link ℓ, $b_{n\ell} = 1$. Similarly, $\mathbf{D} = (d_{rn})_{R\times N}$ represents the flow node matrix, where $d_{rn} = 1$ if $o(r) = n$ or if $d(r) = n$.

It is neccessary to record the spare capacity requirements for each of the ℓ links. Let $\mathbf{G} = \{g_{\ell k}\}_{L\times K}$ denote the spare provision matrix where K is the number of failure scenarios. The element $g_{\ell k}$ denotes the minimum spare capacity required on link ℓ when link k fails. In the case of single-link failures, $K = L$. We also record the maximum spare capacity for each link using column vector $\mathbf{s} = \{s\ell\}_{L\times 1}$.

Given the working paths $\mathbf{P}$, the initial set of backup paths $\mathbf{Q}$, and the demand bandwidth matrix $\mathbf{M}$ it is possible to state the SCA problem as follows:

$$\min_{\mathbf{Q},\mathbf{s}} \phi(s) \tag{4.1.1}$$

such that

$$\mathbf{s} = \max \mathbf{G} \tag{4.1.2}$$

$$\mathbf{G} = \mathbf{Q}^T \mathbf{M} \mathbf{U} \tag{4.1.3}$$

$$\mathbf{T} + \mathbf{Q} \leq 1 \tag{4.1.4}$$

$$\mathbf{Q}\mathbf{B}^T = \mathbf{D}. \tag{4.1.5}$$

In Eq (4.1.1) the objective is to minimize the total cost $\phi(s)$ of spare capacity, through the selection of the backup paths $\mathbf{Q}$ and the spare capacity allocations $\mathbf{s}$.

We use constraints (4.1.2) and (4.1.3) to calculate the spare capacity vector $\mathbf{s}$ from the spare provision matrix $\mathbf{G}$, where $\mathbf{M}$ is a diagonal matrix of bandwidth allocations and $\mathbf{U} = \mathbf{P} \odot \mathbf{F}^{\mathbf{T}}$ is the path failure incidence matrix, with **P** the working path link incidence matrix and **F** the link failure incidence matrix. The operators max and $\odot$ are defined in Table 4.1 on Page 38.

Link disjointness is enforced by constraint (4.1.4). $\mathbf{T} = \mathbf{U} \odot \mathbf{F}$ denotes the flow tabu matrix.

Constraint (4.1.5) guarantees that the backup paths contained in **Q** are feasible paths for the flows in these undirected networks, with **D** being the route node incidence matrix.

4.1.3 Successive Survivable Routing

SSR solves the SCA problem by solving it as a random sequence of single flow problems. The order of selection of the flows r could produce different solutions, and we can select the randomised sequence of flows that delivers the best resulting spare capacity requirement. We present the SSR algorithm from [20] in Algorithm 4. The notation for the SSR algorithm is summarised in Table 4.1.

4.1.4 A working example of SSR

We now present a working example of the SSR algorithm using the five-node network from [20] reproduced in Fig. 4.1.

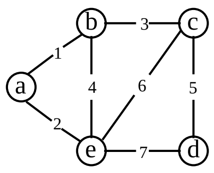

Figure 4.1: The example five-node network.

Each O-D pair has a flow r, with a unit demand to keep the matrices simple. A requirement of SSR is that the working paths and the backup paths are link disjoint. For each O-D pair we calculate a pair of edge-disjoint routes such that the sum of their lengths is minimal using [10] as discussed in Chapter 2. Recall that this is the optimal edge-disjoint pair. However, the calculation of the backup paths before running SSR is not required. It is nonetheless worth investigating if these optimal edge-disjoint pair paths lead to a faster convergence of the SSR algorithm. The SSR algorithm uses $\mathbf{q}_r^* = \mathbf{1}_{1\times L} - (\mathbf{T})_k$ as the backup path to compare to the new backup path. This is not the shortest path, but a list of all non-tabu links used to calculate the incremental cost of spare capacity on all the links.

The working paths are stored in the matrix **P**, the backup paths in the matrix **Q**, the spare capacity requirement in the spare provision matrix (SPM) **G**, and the maximum spare capacity requirement for each link in the column

```
Data: The number of links ($L$), the number of flows ($R$) and the
      number of failure scenarios ($K$). Matrices $\mathbf{P}$, $\mathbf{M}$ and $\mathbf{F}$. Cost
      function $\phi$.
Result: Matrix $\mathbf{Q}$ containing the backup paths.
begin
    Calculate $\mathbf{U} = \mathbf{P} \odot \mathbf{F}^T$
    Calculate $\mathbf{T} = \mathbf{U} \odot \mathbf{F}$.
    Use Algorithm 2 to intialise the backup path matrix $\mathbf{Q}$, only using
    non-tabu links ($t_{r\ell} = 0$).
    Calculate the spare provision matrix $\mathbf{G} = \mathbf{Q^T M U}$.
    Calculate $\mathbf{s} = \max_{rows} \mathbf{G}$
    repeat
        Calculate thisCost $= \sum \phi(\mathbf{s})$
        foreach path $r \in \{1, \ldots, R\}$ in random order do
            Calculate $\mathbf{q}_r = (\mathbf{Q})_r$ and $\mathbf{q}_r^* = \mathbf{1}_{1\times L} - (\mathbf{T})_k$
            Calculate $\mathbf{G}^r = m_r(\mathbf{q}_r^T \mathbf{u}_r)$
            Calculate $\mathbf{G}^{r*} = m_r(\mathbf{q}_r^{*T} \mathbf{u}_r)$
            Calculate $\mathbf{G}^{-r} = \mathbf{G} - \mathbf{G}^r$
            Calculate $\mathbf{s}^{-r} = \max_{rows} \mathbf{G}^{-r}$
            Calculate $\mathbf{s}^* = \max_{rows} \mathbf{G}^{-r} + \mathbf{G}^{r*}$
            Calculate $\mathbf{v} = \phi(\mathbf{s}^*) - \phi(\mathbf{s}^{-r})$
            Each non-tabu link $\ell$ ($t_{r\ell} = 0$) is enabled at cost $v_\ell$.
            All tabu links are disable by setting their cost to $\infty$.
            Use Algorithm 2 to determine the least cost backup path $q$
            for path $r$.
            Update vector $\mathbf{q}_r^{new} = \{q_r^{new}\}_{1\times L}$ set $(q_r^{new})_\ell = 1$ if $q$ makes
            use of link $\ell$ and 0 otherwise.
            if $\mathbf{q}_r^{new}\mathbf{v} < \mathbf{q}_r\mathbf{v}$ then
                Calculate $(\mathbf{Q})_r = \mathbf{q}_r^{new}$
                Calculate $\mathbf{G} = \mathbf{Q}^T\mathbf{MU}$
                Calculate $\mathbf{s} = \max_{rows} \mathbf{G}$
            end
            Calculate newCost $= \sum \phi(\mathbf{s})$
        end
    until thisCost $=$ newCost ;
end
```

Algorithm 4: The SSR algorithm.

Table 4.1: The SSR algorithm notation

N, L, R, K	Number of nodes, links, paths, failure scenarios.
n, ℓ, r, k	Indices of nodes, links, paths, failure scenarios.
$o(r), d(r)$	Origin and destination nodes of path r.
$\mathbf{P} = \{p_{r\ell}\}_{R\times L}$	Working path link incidence matrix: $p_{r\ell} = 1$ if the path r traverses link ℓ and 0 otherwise.
$\mathbf{Q} = \{(q_{r\ell}\}_{R\times L}$	Backup path link incidence matrix: $(q_k)_{r\ell} = 1$ if the backup path q_r for path r in scenario k traverses link ℓ and 0 otherwise.
$\mathbf{M} = \text{Diag}(\{m_r\})_{R\times R}$	Diagonal bandwidth demand matrix m_r of path r.
$\mathbf{F} = \{f_{k\ell}\}_{K\times L}$	Failure link incidence matrix: $f_{k\ell} = 1$ if link ℓ fails in scenario k and 0 otherwise.
$\mathbf{U} = \{u_{rk}\}_{R\times K}$	Flow failure incidence matrix: $u_{rk} = 1$ iff flow r's working path is affected in scenario k and 0 otherwise.
$\mathbf{T} = \{t_{rl}\}_{R\times L}$	Flow tabu-link matrix: $t_{r\ell} = 1$ iff link ℓ should not be used on flow r's backup path.
$\mathbf{D} = \{d_{rn}\}_{R\times N}$	Flow node incidence matrix.
$\phi = \{\phi_l\}_{L\times 1}$	Backup capacity cost function: ϕ_ℓ is the cost of a unit of backup capacity on link ℓ.
$\mathbf{G}_k = \{(g_k)_{\ell r}\}_{L\times R}$	Path backup requirement per scenario: $(g_k)_{\ell r}$ is the backup capacity required on link ℓ for path r in failure scenario k.
$\mathbf{G} = \{g_{\ell k}\}_{L\times K}$	Backup provision matrix: $g_{\ell k}$ is the backup capacity required on link ℓ for failure scenario k.
$\mathbf{B} = \{b_{\ell k}\}_{L\times K}$	Giveback capacity matrix: $b_{\ell k}$ is the giveback capacity released from link ℓ for failure scenario k.
$\mathbf{v}_r = \{(v_r)_\ell\}_{L\times 1}$	Link cost vector: v_ℓ is the cost of including link ℓ into the currently rerouted backup path q_r.
$\mathbf{s} = \{s_\ell\}_{L\times 1}$	Vector of link backup capacity: s_ℓ is the backup capacity required on link ℓ for any failure scenario.
$\mathbf{e} = \{e_\ell\}_{1\times L}$	Unit vector.
$\mathbf{U} \odot \mathbf{F}$	The $\odot$ operator denotes binary matrix multiplication. The matrix containing the multiplication results only stores the values 0 or 1. In the case where the matrix multiplication result for any entry is greater than 1, it is reset to 1.

vector **s**. This is illustrated in Fig. 4.2. Not shown is the failure matrix **F**, which is an identity matrix. This matrix represents the scenario of provisioning spare capacity to construct equivalent recovery paths for all single link failures.

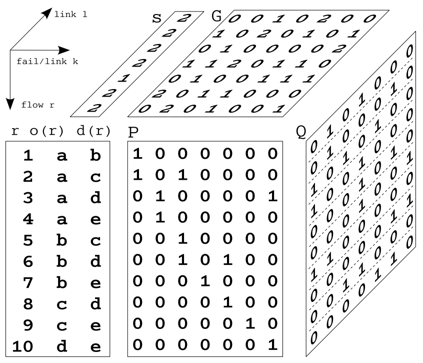

Figure 4.2: The SCA structure for the five-node network.

The axes at the top left provide the dimension of the matrices and the column vector. We focus on flows 2 and 6. The matrix **P** reveals that the working path for flow 2 uses links 1 and 3, while the backup path for flow 2 uses links 2 and 7. Likewise the working path for flow 6 uses links 3 and 5, and the backup path uses links 4 and 7. Note that the working paths the backup paths are link-disjoint.

These two sets of working and backup paths, combined with the flow failure incidence matrix **U**, give the SPM **G**, with the maximum spare capacity required stored in the column vector **s**. The current total of the vector **s** in Fig. 4.2 gives the required spare capacity to provision for the single link failure scenario provided by matrix **F**, without SSR optimisation. The current total spare capacity is 13.

We will follow the same steps to perform SSR as given in [20] on Page 204. The example is explained in Fig. 4.3.

Consider flow 2 between the O-D pair $a{-}c$. In the case of protecting against single link failures we have $\mathbf{t}_2 = \mathbf{u}_2 = \mathbf{p}_2$. Refer to Figure 4.3 above, this implies that for path $\mathbf{r} = 2$ - the flow tabu-link matrix, flow failure incidence matrix and working path link matrix are the same. We follow the 5 steps of the SSR algorithm:

1. Compute $\mathbf{u}_2$ and $\mathbf{t}_2$ as shown above.

2. Sum the $\mathbf{G}_r$ matrices into **G**.

3. a) Compute the SPM $\mathbf{G}^{-2} = \mathbf{G} - \mathbf{G}^2$ and the backup path matrix $\mathbf{Q}^{-2} = \mathbf{Q} - \mathbf{Q}^2$ after the current backup path $\mathbf{q}_2$ is removed. The

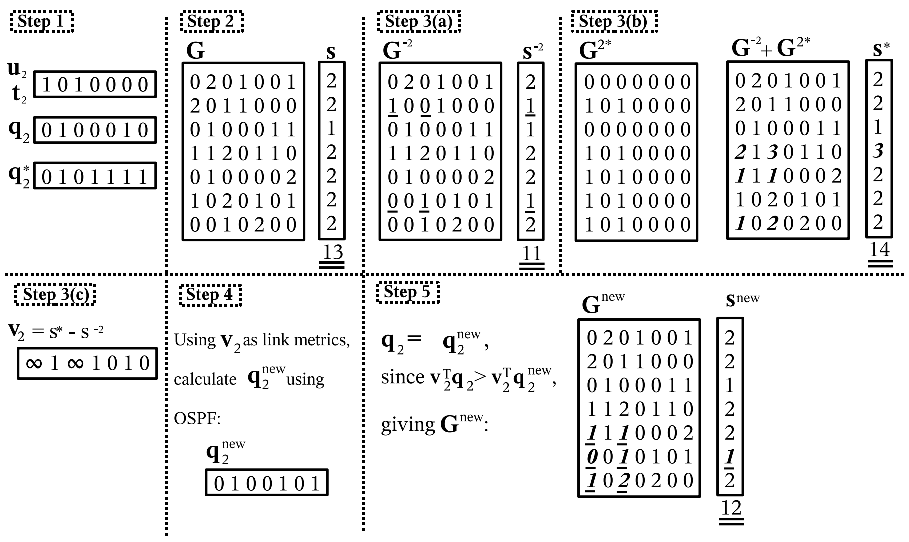

Figure 4.3: Find a new backup path for flow 2 using SSR.

differences between $\mathbf{G}$ and $\mathbf{G}^{-2}$, and between $\mathbf{s}$ and $\mathbf{s}^{-2}$ are indicated by the underlined entries in Step 3(a) of Fig. 4.3.

b) Use $\mathbf{q}_2^* = \mathbf{e} - \mathbf{t}_2$ as the new backup path using only non-tabu links. As we already have the information available in Step 1, this vector is shown in Step 1 above. We obtain $\mathbf{G}^{2*}(\mathbf{q}_2^*) = m_2 \mathbf{q}_2^{*T} \mathbf{u}_2$. This $\mathbf{G}^{2*}(\mathbf{q}_2^*)$ is the new spare provision matrix for flow r. Obtain the new capacity reservation vector from the new spare provision matrix as $\mathbf{s}^*(\mathbf{q}_2^*) = \max(\mathbf{G}^{-2} + \mathbf{G}^{2*}(\mathbf{q}_2^*))$. The capacity reservation vector now adds to 14. The changes between $\mathbf{G}$ and $\mathbf{G}^{2*}(\mathbf{q}_2^*)$, and between $\mathbf{s}$ and $\mathbf{s}^*(\mathbf{q}_2^*)$ are indicated by the bold and italic entries in Step 3(b) of Fig 4.3.

c) Calculate the vector of link cost metrics for flow r that gives the incremental cost of using a link on the backup path for the given flow. If $\mathbf{v}_{2\ell} = 0$, we can use link ℓ on the backup path of flow *2* without having to deploy any additional spare capacity. From Fig. 4.3 comparing $\mathbf{s}^{-2}$ and $\mathbf{s}^*$ we see that if the backup path uses any of links 2, 4, and 6 then it needs an additional unit of capacity on each of these links. The backup path will need no additional capacity on links 5 and 7 should it use these links.

4. Use the cost vector $\mathbf{v}$ as the link cost metric. Rerun the shortest path algorithm, obtaining the new backup path $\mathbf{q}_2^{new}$, which uses links 2, 5, and 7.

5. Deploy the new backup path $\mathbf{q}_2^{new}$ if it has a lower cost than the existing backup path, with the path cost based on the link metrics in $\mathbf{v}_2$:

   $\mathbf{q}_2 = \mathbf{q}_2^{new}$, when $\mathbf{v}_2^T \mathbf{q}_2 \; \mathbf{v}_2^T \mathbf{q}_2^{new}$.

As shown in Fig. 4.3 in Step 1, the current backup path uses links 2 and 6. The new backup path uses links 2, 5, and 7. In terms of hop count, the current backup path is the shortest. Comparing the incremental cost of the path using $\mathbf{v}_2$ we see that the current path requires 2 units of additional capacity (links 2 and 6), and the new path requires only 1 unit of additional capacity (link 2).

The new backup path is added to $\mathbf{Q}$ and $\mathbf{G}$ and $\mathbf{s}$ is recalculated. We end this iteration of the SSR algorithm with $\mathbf{G}^{new}$ and $\mathbf{s}^{new}$. The total spare capacity has been reduced from 13 to 12. The differences between $\mathbf{G}$ and $\mathbf{G}^{new}$, and between $\mathbf{s}$ and $\mathbf{s}^{new}$ are indicated by the bold and italic and underlined entries in Step 5 of Fig 4.3.

It is left to the reader to show that using the steps above for flow 6, we can further reduce the total spare capacity requirement from 12 to 11.

4.2 Capacity giveback

A route between a O-D pair consists of a set of links. If the flow on the route requires n units of capacity, each link that forms part of that route requires n units of capacity.

In any particular failure scenario we have a corresponding set of link failures. These failed links form part of the routes carrying the flow r between the route's O-D pair. As already mentioned, the working and backup paths are link disjoint. For every failure scenario, the non-failed links are still capacitated to carry the flow of the route affected by the failure scenario. The link disjointness requirement implies that a link in the working route can never be part of the backup route. This capacity is therefore unavailable to the backup path of the failed flow r that it supports. This capacity can however be used by the backup route of another flow that is allowed to use this link as part of its backup path.

In order to keep track of the capacity made available by the given failure scenarios, we calculate giveback capacity in (4.2.1)

$$\mathbf{B} = \mathbf{P}^{\mathbf{T}}\mathbf{M}\mathbf{U}. \tag{4.2.1}$$

We store the resulting giveback capacity in matrix $\mathbf{B}$, as listed in Table 4.1 on Page 38.

A link can be a surviving element of a route for more than one failure scenario. Continuing with the example network in Fig. 4.1 on Page 36, with the working and backup paths as given in Fig. 4.2 on Page 39, we examine link 3. As shown in the working path matrix $\mathbf{P}$, link 3 is used together with link 1 to form the route for flow 2, and together with link 5 to form the route for flow 6. We assume that flow 2 has a flow of 2 units of capacity, and flow 6 has a requirement of 6 units of capacity. In the event of link 1 failing, we have link 3 as the surviving link, with 2 units of capacity made available to it, and when link 5 fails, link 3 has 6 units of capacity made available to it. Different amounts of spare capacity are freed, depending on which failure scenario applies.

We modify Step 3 of the SSR heuristic in Algorithm 4 by calculating the spare provision matrix G as:

$$\mathbf{G} = [\mathbf{Q}^T\mathbf{MU} - \mathbf{B}]^+ \tag{4.2.2}$$

We now reduce the spare capacity requirement for $g_{\ell k}$, the backup path link ℓ used in failure scenario k, by the amount of capacity given back by $p_{\ell k}$, the surviving working path link ℓ when failure scenario k occurs. The reader will notice that we return the capacity of the failed link(s) ℓ in failure scenario k, in this case of single link failure, Diag($\mathbf{B}$). Consider however that $\mathbf{g}_{\ell k}$ will always be zero, since edge-disjointness requires that this link cannot be used in the backup path. Even if the giveback capacity from the failed link is deducted, the non-negative sum would reset the giveback capacities for the failed links to zero.

Also in Step 3, by implication, $\mathbf{G}^{-r}$ now represents the overall backup capacity requirement for each link and failure scenario with the current backup path for r removed, but with the surviving links working bandwidth being made available to backup routes, thereby reducing the cost of using these links in the backup paths. Note however that the requirement of edge-disjointness implies that the same backup path cannot use the surviving link giveback capacity. This can be utilised by other backup paths that are also affected by the same failure scenario.

4.3 State-dependent backup routing

One of the requirements of the SSR algorithm is that the backup path must be link disjoint from the working path. From [20] we recall:

$$\mathbf{U} = \mathbf{P} \odot \mathbf{F}^T \tag{4.3.1}$$

$$\mathbf{T} = \mathbf{U} \odot \mathbf{F} \tag{4.3.2}$$

$$\mathbf{T} + \mathbf{Q} \leq 1 \tag{4.3.3}$$

This restriction leads to the exclusion of the surviving links in the working path, which continue to function after the failure of the failed links. If we calculate the backup paths required in each failure scenario k it is possible to make use of all the surviving links in the network. In order to achieve this we need to modify SSR. We begin with the additional matrix operations required to achieve this, as given in Table 4.2.

First, the tabu link matrix $\mathbf{T}$ is no longer required, since all failed links are filtered using the failure matrix $\mathbf{F}$ and all surviving links are candidates for

Table 4.2: The SSR-SD definitions

$\mathbf{v} = \sum_{\text{rows}} \mathbf{A}$	Column vector containing the sum of each row of $\mathbf{A}$, i.e. (in case $\mathbf{A}$ is a $I \times J$ matrix) $v_i = \sum_j a_{ij}$

the backup path. We now require a set of backup matrices $\mathbf{Q}_k$ $(1 \leq k \leq K)$ that contains the computed backup paths for each failure scenario k. If a path r does not fail in scenario K, a backup path would not be required and $(\mathbf{Q_k})_\mathbf{r}$ will be zero.

We now require G_k, which is the SPM for each scenario's computed backup paths:

$$\mathbf{G}_k = \mathbf{Q}_k^T \mathbf{M}. \tag{4.3.4}$$

The SPM for each failure scenario has to be aggregated into the SPM $\mathbf{G}$. We are interested in the total backup capacity for all paths using capacity on link ℓ, therefore we use $\sum_{rows} \mathbf{G}_k$ to store the capacity requirement for scenario k in $\mathbf{C}_k$.

In order to present the spare capacity requirement in the SPM $\mathbf{G}$ we need to combine the results for each of the failure scenarios. We achieve this by concatenating each of the column vectors $\mathbf{C}$ with link capacity requirements for each failure scenario k:

$$\mathbf{G}_k = (\mathbf{C}_1 \,|\mathbf{C}_2\,| \ldots |\mathbf{C}_k) \ . \tag{4.3.5}$$

If capacity giveback is used we calculate $\mathbf{G}$ as:

$$\mathbf{G} = \left[(\mathbf{C}_1 \,|\mathbf{C}_2\,| \ldots |\mathbf{C}_K) \ - \mathbf{P}^T \mathbf{M} \mathbf{U} \right]^+ . \tag{4.3.6}$$

The end result is the SPM $\mathbf{G}$ and the remainder of the SSR-SD algorithm is the same as SSR.

Table 4.3: The additional SSR-SD notation

$\mathbf{C}_k = \{(c_k)_\ell\}_{L\times 1}$	Link capacity requirement per scenario: $(c_k)_\ell$ is the link capacity required for each failure scenario k
$\mathbf{Z} = \{z_{rk}\}_{R\times K}$	Failed backup path and failure scenario optimisation attempts matrix: z_{rk} counts the successive unsuccessful attempts of changing the backup path q_r for scenario k.

We are now ready to implement the SSR-SD algorithm which summarised in Algorithm 5.

```
Data: values $L$, $R$, $K$, matrices $\mathbf{P}$, $\mathbf{M}$, $\mathbf{F}$, $\mathbf{Q}_{1\leq k\leq K}$, cost function $\phi$
Result: Set of matrices $\mathbf{Q}_k$ containing backup paths. Column vector $\mathbf{s}$
        with backup capacity to employ
begin
    repeat
        Calculate curCost $= \sum \phi(\mathbf{s})$
        Calculate $\mathbf{Z} = (\mathbf{0})_{R\times K}$
        for each link $\ell \in [1...L]$ with $s_\ell > 0$ in random order do
            for each scenario $k \in [1...K]$ with $g_{\ell k} = s_\ell$ in random order
            do
                for each path $r \in [1...R]$ with $(g_k)_{\ell r} > 0$ and $z_{rk} < 2$ in
                random order do
                    Calculate $q_r = (\mathbf{q}_k)_r$.
                    Calculate $q_r^0 = (\mathbf{0})_{1\times L}$.
                    Calculate $q_r^* = (\mathbf{e})_{1\times L} - (\mathbf{F}^T)_k$.
                    Calculate $\mathbf{s}^0 = \mathbf{s}$ by setting $(\mathbf{q}_k)_r$ to $q_r^0$.
                    Recalculate $\mathbf{G}_k$, $\mathbf{G}$, and $\mathbf{s}$.
                    Calculate $\mathbf{s}^* = \mathbf{s}$ by setting $(\mathbf{q}_k)_r$ to $q_r^*$.
                    Recalculate $\mathbf{G}_k$, $\mathbf{G}$, and $\mathbf{s}$.
                    Calculate $\mathbf{v}_r = \phi(\mathbf{s}^*) - \phi(\mathbf{s}^0)$ add 1 to $(v_r)_\ell$.
                    Calculate the least cost backup path $q_r^{new}$ for path $r$ in
                    scenario $k$ using Algorithm 2 with any surviving link $\ell'$
                    (i.e. $f_{k\ell'} = 1$) enabled at cost $(v_r)_{\ell'}$
                    if $q_r^{new} \boldsymbol{v}_r < q_r \boldsymbol{v}_r$ then
                        Calculate $(\mathbf{q}_k)_r = q_r^{new}$;
                        Set $z_{rk} = 0$;
                    else
                        Calculate $(\mathbf{q}_k)_r = q_r$;
                        Increment $z_{rk}$ by 1;
                    end
                    Recalculate $\mathbf{G}_k$, $\mathbf{G}$, and $\mathbf{s}$;
                end
            end
        end
    until curCost $= \sum \phi(\mathbf{s})$ ;
end
```

Algorithm 5: The SSR-SD algorithm.

4.4 SSR results for link failure scenarios

In this section we present the results of the SSR algorithm. We first examine the results of the SSR example networks to verify our implementation. We next examine the results of the SSR algorithm on our "real-world" network topologies as discussed in Section 3.1 on Page 25. Lastly we examine the the results of the BRITE generated network topologies as discussed in Section 3.2 on Page 30.

4.4.1 SSR example network topologies results

The eight network topologies provided in [20] on Page 206 are used as input models to verify our implementation of the SSR algorithm. In addition the performance of capacity giveback discussed in Section 4.2 and state-dependent SSR discussed in Section 4.3 is evaluated. Table 4.4 gives the number of nodes (N), the number of undirected / bidirectional links (L), and the average node degree ($\overline{d}$).

Table 4.4: Example network information

Network	1	2	3	4	5	6	7	8
N	10	12	13	17	18	23	26	50
$L(A)$	22 (44)	25 (50)	23 (46)	31 (62)	27 (54)	33 (66)	30 (60)	82 (164)
$\overline{d}$	4.4	4.17	3.54	3.65	3	2.87	2.31	3.28

The simulations were run on an Intel Pentium 1.5GHz with 1.5Gb of memory. Following the network redundancy performance results in Figure 4.4 the computation time required to run 20 iterations of the SSR algorithm for 20 replications are given in Figure 4.5 below. We also provide the figures used for charting Figure 4.4 in Table 4.5 on Page 46.

The following five variations of the SSR algorithm are evaluated:

1. SSR,
2. SSR with capacity giveback,
3. SSR state-dependent,
4. SSR state-dependent and capacity giveback, and
5. SSR state-dependent, capacity giveback, and non-unit link capacity.

The first four cases are variations on topics already discussed. In the fith case we evaluate the case of a network capacitated with non-unit link capacities. This gives a more acurate presentation of a real-world network topology.

With the unit link capacities presented in [20] the deviation in spare capacities required in the spare provision matrix **G** is more limited than with non-unit link capacities. We show the effect this has on the backup capacity provisioning.

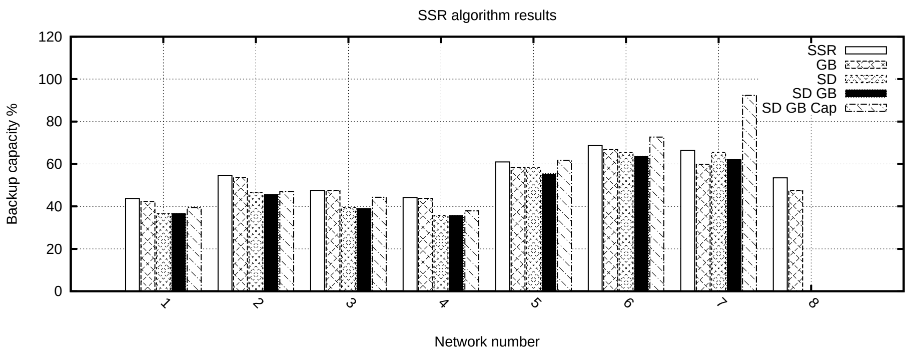

Figure 4.4: The SSR algorithm backup capacity requirements.

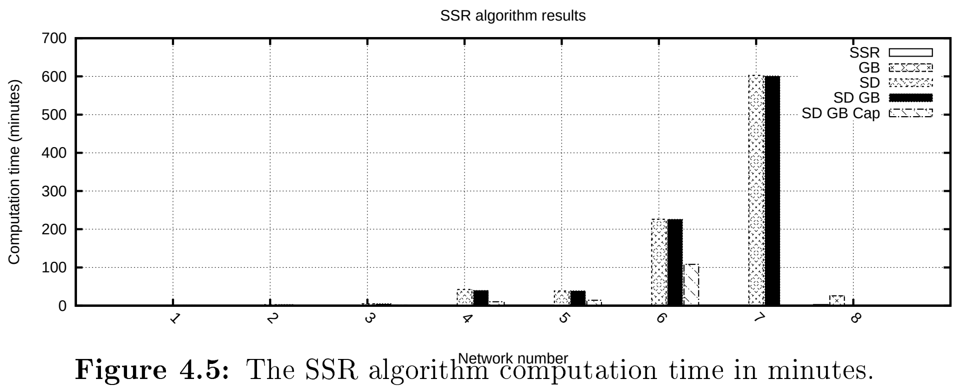

Figure 4.5: The SSR algorithm computation time in minutes.

Table 4.5: SSR backup capacity requirement

Network	1	2	3	4	5	6	7	8
% SSR	43.7	54.5	47.5	44.1	61	68.6	66.4	53.5
% GB	42.3	53.8	47.5	43.8	58.3	66.8	59.9	47.6
% SD	36.6	46.4	39.5	35.7	58.3	65.4	65.4	-
% SD & GB	36.6	45.5	38.9	35.7	55.4	63.6	62	-
% SD & GB & Cap	39.4	46.9	44.3	37.9	61.7	72.7	93.3	-

Note that there are no results for the three state-dependent simulations of network 8. The size of the network and number of states to be maintained in the simulation is too large to timely compute the results.

The decrease in backup capacity where SSR capacity giveback is used ranges from roughly 0.3 to 6.5 percent for the eight networks. Network 4 shows the least improvement with roughly 0.3 percent, and Network 7 shows the most improvement of 6.5 percent. This is noteworthy as Network 7 has the lowest average node degree of all the eight networks and gains the largest benefit from capacity giveback.

The limited node connectivity, as given by the node degree, means that path selection is more limited in these networks and that paths share more links than highly connected networks. This indicates that the SSR algorithm has fewer alternate paths to select in order to maximise backup capacity utilisation, as indicated by Network 7 having the second worst backup capacity requirement of the eight networks. This same characteristic leads to the increased benefit of capacity giveback. The working paths share more common links, and therefore in any given failure scenario the surviving links return bandwidth to the network that can then be better utilised by the unaffected working paths.

4.4.2 "Real-world" topologies SSR results

We next examine the results of the SSR algorithm when applied to the following "real-world" network topologies:

- AT&T USA network topology,
- MCI USA network topology,
- EBone European network topology,
- Tiscali European network topology, and
- Telstra Australian network topology.

Table 4.6 gives the number of nodes (N), the number of undirected / bidirectional links (L), and the average node degrees ($\bar{d}$).

Table 4.6: "Real-world" network profiles

Network	AT&T	MCI	EBone	Tiscali	Telstra
N	31	43	26	48	50
$L(A)$	65 (130)	94 (188)	48 (96)	101 (202)	79 (158)
$\bar{d}$	4.194	4.732	3.692	4.208	3.16

The simulations were run on an Intel Pentium 1.5GHz with 1.5Gb of memory. Figure 4.6 presents the network redundancy performance results and Figure 4.7 presents the computation time required to run 20 iterations of the

SSR algorithm for 20 replications. The same five variations of the SSR algorithm as for the SSR example networks are evaluated. Note that for all the results that are beyond the scale of the computation time axis, the simulation could not be completed due to the excessive run-time of the algorithm. There is a stopping rule at 5000 minutes. If one SSR replication cannot be completed by the 5000 minute limit, the simulation is aborted. In the cases where at least one SSR replication could be completed, we use the one result obtained. If there is no result for the given network and SSR algorithm combinations, the table will have a dashed entry to indicate no result.

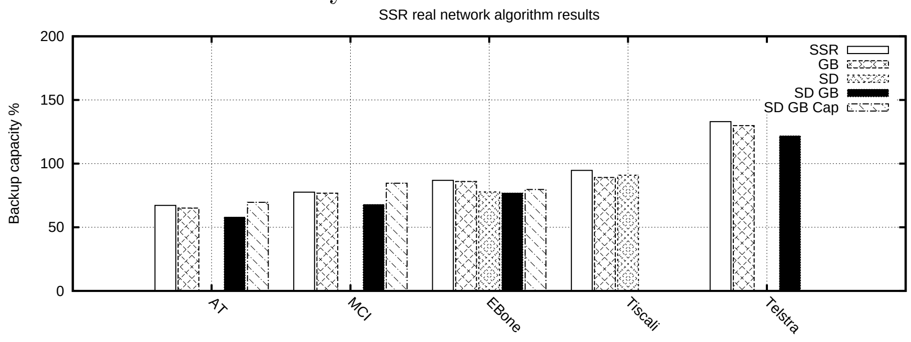

Figure 4.6: The SSR algorithm backup capacity requirements.

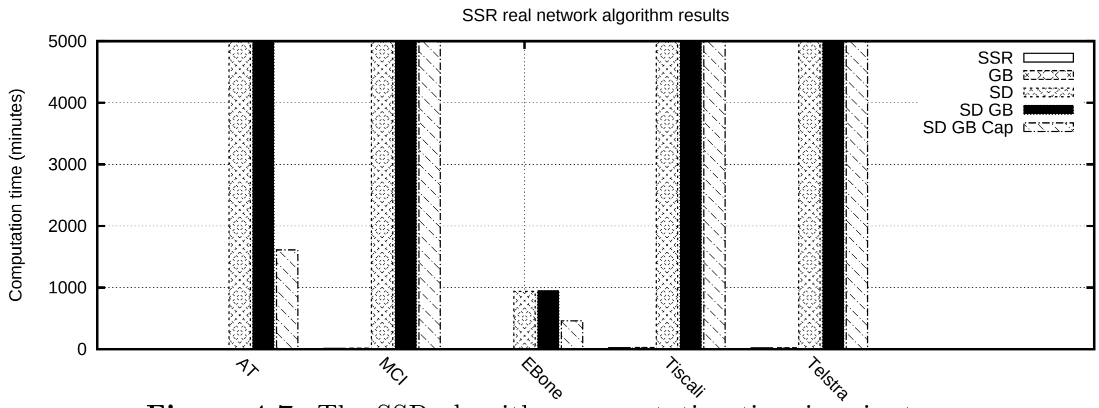

Figure 4.7: The SSR algorithm computation time in minutes.

4.4.3 BRITE generated topologies SSR results

We provide the SSR performance results for six BRITE generated network topologies. In addition to the performance of the SSR algorithm, the performance of the capacity giveback discussed in Section 4.2, and state-dependent SSR discussed in Section 4.3 are evaluated.

Table 4.8 gives the number of nodes (N), the number of undirected / bidirectional links (L), and the average node degrees ($\overline{d}$).

Table 4.7: SSR "real-world" network result data

Network	AT&T	MCI	EBone	Tiscali	Telstra
% SSR	67.2	77.6	86.8	94.7	133
% GB	65	76.8	86	89.1	129.8
% SD	-	-	77.7	90.9	-
% SD & GB	57.8	67.7	76.8	-	121.7
% SD & GB & Cap	69.6	84.6	79.7	-	-

Table 4.8: BRITE topology profiles

Network	N10D4Wax	N10D4BA2	N20B2Wax	N20B2BA2	N30D2Wax	N30D2BA2
N	10	10	20	20	30	30
$L(A)$	34 (68)	30 (60)	40 (80)	55 (110)	60 (120)	119 (238)
$\overline{d}$	6.8	6	4	5.5	4	7.933

We next present the results for the SSR algorithm on the BRITE generated network topologies in Figure 4.8, while the computation times for the algorithms are given in Figure 4.9. The network results that were used to plot the results are given in Table 4.9.

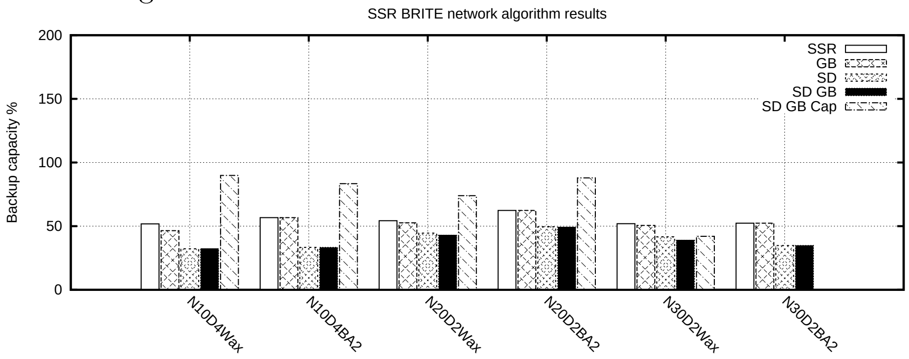

Figure 4.8: The BRITE topologies SSR algorithm backup capacity requirements.

Table 4.9: SSR BRITE network result data

Network	N10D4Wax	N10D4BA2	N20D2Wax	N20D2BA2	N30D2Wax	N30D2BA2
% SSR	51.8	56.7	54.2	62.4	52	52.4
% GB	46.4	56.7	52.5	62.4	50.6	52.4
% SD	32.1	33.3	44.6	49.4	41.7	34.7
% SD & GB	32.1	33.3	42.9	49.1	38.8	34.7
% SD & GB & Cap	89.8	83.4	74	88	-	-

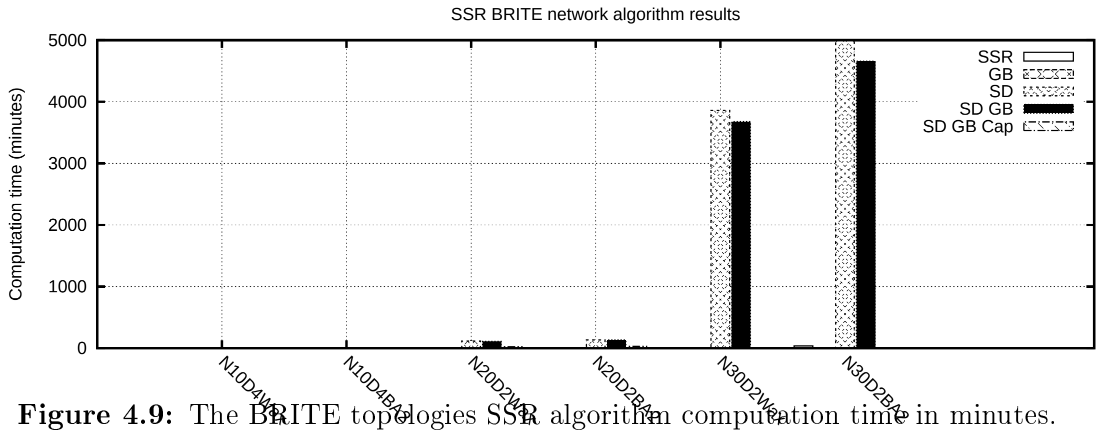

Figure 4.9: The BRITE topologies SSR algorithm computation time in minutes.

4.5 Network topology engineering for SSR

The SSR algorithm results for the SSR example networks, "real-world" networks and BRITE generated networks required further investigation regarding the network profiles and their resulting backup requirements. The SSR example networks on average required 54.9% backup capacity. The "real-world networks" on average required 92.1% backup capacity. The BRITE generated network topologies required 54.9% backup capacity. It is of concern that the inherent properties of our "real-world" network topologies have such a significant impact on the performance of the SSR algorithm.

We evaluated the SSR example 6 network, the EBone, AT&T and MCI "real-world" networks and the N10D4BA2 and N20D2BA2 networks. Both the SSR example network 6 and the N20D2BA2 BRITE generated networks had the largest required backup capacity (68.7% and 62.3%) for their respective categories. The EBone network had the third highest percentage required backup capacity of the "real-world" networks, but the number of network nodes (26) and links (48) made engineering the topology easier. We included the AT&T and MCI networks for the same reason.

We first examine the network profile of the SSR example network 6. The average node degree of 2.87 is the second lowest of all the SSR example networks. SSR example network 7, which has the second highest backup capacity requirement for the SSR example networks, has the lowest average node degree of 2.31.

The EBone, AT&T and MCI networks show that the average node degree does not primarily determine the spare capacity requirement. The EBone network has a 86.8% backup capacity requirement, and an average node degree of 3.7. The AT&T network has a 67.2% backup capacity requirement, and an average node degree of 4.2. The MCI network has a 77.6% backup capacity requirement, and an average node degree of 4.7. The EBone network has the lowest average node degree and the highest backup capacity requirement. The MCI network however, with a higher node degree than the AT&T network, has a higher backup capacity requirement. If we look at the BRITE generated network topologies we see that the N20D2BA2 and N10D4BA2 networks, which have the highest (62.4%) and second-highest (56.7%) backup capacity requirements, have high average node degrees (5.5 and 6). Furthermore the average node degree for the SSR example networks is 3.4%, for the "real-world" network examples it is 3.9% and for the BRITE generated networks it is 5.7%. The low average node degree for the SSR example networks show that it may not be possible to reduce the spare capacity requirement simply by adding more links (thus increasing the average node degree).

We next evaluate the deviation of the node degrees. The backup capacity requirements as well as the standard (SD) and maximum deviation (MD) for the six networks we focus on are given in Table 4.10.

Table 4.10: SSR standard and maximum deviation investigation

Network	SSR6	EBone	AT&T	MCI	N10D4BA2	N20D2BA2
% SSR	68.65	86.83	67.16	77.61	56.67	62.36
SD	0.76	1.72	2.28	3.23	1.63	3.49
MD	2.13	4.31	5	10.63	2	7.5

The maximum deviation for the EBone and MCI networks, the two networks with the worst backup capacity requirements are noticably high. The N20D2BA2 network however has the second highest maximum deviation and the highest standard deviation, yet its backup capacity requirement is the second lowest. On closer examination of the N20D2BA2 network we see the following node degrees (d) and difference from the average node degree (Δd) for the network as given in Table 4.11.

Table 4.11: The N20D2BA2 network node degrees

Node	d	Δd	Node	d	Δd
1	13	7.5	11	7	1.5
2	3	2.5	12	3	2.5
3	9	3.5	13	2	3.5
4	8	2.5	14	2	3.5
5	6	0.5	15	4	1.5
6	8	2.5	16	3	2.5
7	3	2.5	17	3	2.5
8	7	1.5	18	3	2.5
9	13	7.5	19	3	2.5
10	8	2.5	20	2	3.5

Note that the main contributors to the high standard deviation are nodes 1 and 9, which both have the maximum difference of 7.5 from the average node degree. We now compare these properties to that of the EBone network model, which has the highest backup capacity requirement of all the 6 networks we investigated. The node degrees (d) and difference from the average node degree (Δd) for the EBone network are given in Table 4.12

Table 4.12: The EBone network node degrees

Node	d	Δd	Node	d	Δd
1	5	1.3	14	4	0.3
2	3	0.7	15	4	0.3
3	4	0.3	16	2	1.7
4	5	1.3	17	5	1.3
5	2	1.7	18	2	1.7
6	3	0.7	19	2	1.7
7	5	1.3	20	3	0.7
8	8	4.3	21	2	1.7
9	7	3.3	22	3	0.7
10	6	2.3	23	3	0.7
11	3	0.7	24	2	1.7
12	3	0.7	25	2	1.7
13	6	2.3	26	2	1.7

The EBone network topology has node 8 with a maximum node degree difference of 4.308, and node 9 is second highest with a node degree difference of 3.308. Also of interest is that 8 nodes in total have the minimum required node degree of 2. In the N20D2BA2 network, while having 6 nodes less than the EBone network, there are only a total of 3 nodes that have the minimum required node degree of 2.

We therefore re-engineer the EBone network topology to a more preferentially connected (hubbed) network as in [21]. We keep the same number of nodes and links in the network, moving network links as required for the hubbed topology. We choose nodes 22 and 24 as the hub nodes to which we relocate links. After these changes all nodes have a node degree of 3 or more. The links are moved as follows:

- Remove node 1 to 7 link, and add node 5 to 14 link.
- Remove node 4 to 9 link, and add node 8 to 18 link.
- Remove node 9 to 10 link, and add node 16 to 21 link.

- Remove node 9 to 13 link, and add node 14 to 19 link.
- Remove node 12 to 13 link, and add node 12 to 14 link.
- Remove node 7 to 10 link, and add node 14 to 24 link.
- Remove node 8 to 17 link, and add node 2 to 25 link.
- Remove node 2 to 17 link, and add node 17 to 26 link.

We run the SSR algorithm on the modified, hubbed topology. Table 4.13 compares the results and network topology of the original EBone network and the hubbed EBone network.

Table 4.13: The EBone network properties and results

EBone network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	86.83	% SSR	45.77	Δ % SSR	-41.07
$\overline{d}$	3.692	$\overline{d}$	3.692	$\Delta\overline{d}$	0
SD	1.715	SD	1.35	Δ SD	-0.365
MD	4.308	MD	4.308	Δ MD	0

By moving 8 links in the network we gain an improvement of 41.1% in the required backup capacity. The standard deviation is reduced by 0.365 degrees. The maximum node degree deviation is unchanged.

We now use the same approach with the SSR example network 6. We keep the same number of nodes and links in the network, and move the links as required for the hubbed topology. We choose nodes 6, 11 and 16 as hubs. The links are moved as follows:

- Remove node 1 to 2 link, and add node 6 to 7 link.
- Remove node 3 to 18 link, and add node 8 to 11 link.
- Remove node 2 to 5 link, and add node 6 to 11 link.
- Remove node 9 to 20 link, and add node 6 to 16 link.
- Remove node 12 to 14 link, and add node 11 to 16 link.
- Remove node 15 to 18 link, and add node 5 to 6 link.

Table 4.14: The SSR6 network properties and results

SSR6 network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	68.6	% SSR	58.6	Δ % SSR	-9.9
$\overline{d}$	2.87	$\overline{d}$	2.87	$\Delta\overline{d}$	0
SD	0.75	SD	1.14	Δ SD	0.39
MD	2.13	MD	3.13	Δ MD	1

We run the SSR algorithm on the modified, hubbed network topology. Table 4.14 compares the results and network topology of the original SSR example 6 network and the hubbed SSR example 6 network.

By moving 6 links in the network we gain an improvement of 9.9% required backup capacity. We also increased the standard deviation by 0.39. The maximum node degree deviation has increased by 1. Note that the orignial SSR example 6 network contains 7 nodes with the minimum required node degree of 2. Due to the restriction of maintaining the same number of links in the modified, hubbed SSR example 6 network, the modified network has 12 nodes with the minimum required node degree of 2.

We now compare the hubbed network approach with a more connected network, where we increase the minimum node degree for all nodes to 3 and we bring down the standard deviation of the node degree. Due to the average node degree of 2.87 for the SSR example network 6 we need to add two more links to ensure the minimum node degree of 3 for all nodes. We move and add links as follows:

- Remove node 1 to 2 link, and add node 6 to 7 link.
- Remove node 2 to 5 link, and add node 5 to 10 link.
- Remove node 18 to 19 link, and add node 19 to 21 link.
- Add node 8 to 11 link.
- Add node 2 to 13 link.

We run the SSR algorithm on the modified, more connected network topology. Table 4.15 compares the results and network topology of the original SSR example 6 network and the more connected SSR example 6 network.

A more connected network topology with a minimum node degree of 3, we gain a 31.9% improvement in required backup capacity as opposed to the 9.9% improvement in our first attempt at re-engineering the SSR example 6 network. We had moved 3 links and added 2 more to the network.

Table 4.15: The SSR6 network properties and results

SSR6 network					
Network	Original	Network	Connected	Network	Δ
% SSR	68.56	% SSR	36.59	Δ % SSR	-31.97
$\bar{d}$	2.87	$\bar{d}$	3.04	$\Delta\bar{d}$	0.17
SD	0.75	SD	0.209	Δ SD	-0.541
MD	2.13	MD	0.957	Δ MD	-1.173

We now confirm the minimum node degree recommendation of 3 by applying this topology change for the “real-world” MCI network. We leave network nodes 6, 9 and 16 as network hubs, as they have 15, 14 and 10 network links respectively. We move the network links as follows:

- Remove node 12 to 18 link, and add node 1 to 8 link.
- Remove node 12 to 43 link, and add node 11 to 13 link.
- Remove node 4 to 7 link, and add node 15 to 19 link.
- Remove node 4 to 12 link, and add node 20 to 22 link.
- Remove node 7 to 12 link, and add node 23 to 26 link.
- Remove node 28 to 29 link, and add node 31 to 34 link.
- Remove node 28 to 30 link, and add node 32 to 38 link.
- Remove node 40 to 41 link, and add node 36 to 39 link.
- Remove node 35 to 40 link, and add node 9 to 42 link.

We run the SSR algorithm on the modified hubbed network topology with the minimum network degree increased from 2 to 3. Table 4.16 compares the results and network topology of the original MCI network and the modified MCI network.

This delivers a 27.2% improvement in the backup capacity requirement, the original network hubs were not significantly modified, with one link being added to hub node 9. The node degrees of hub nodes 6 and 10 were not changed. The biggest improvement in the performance of the SSR algorithm is made by increasing the minimum node degree from 2 to 3 for all networks being provisioned with backup capacity and routes.

We apply the same minimum node degree increase for the AT&T “real-world” network model. We select nodes 14 and 15 as the hub nodes. We move the following links:

Table 4.16: The MCI network properties and results

MCI network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	77.61	% SSR	50.43	Δ % SSR	-27.18
$\overline{d}$	4.732	$\overline{d}$	4.732	$\Delta\overline{d}$	0
SD	3.23	SD	2.854	Δ SD	-0.376
MD	9.628	MD	10.628	Δ MD	1

- Remove node 1 to 3 link, and add node 6 to 14 link.
- Remove node 1 to 4 link, and add node 7 to 15 link.
- Remove node 1 to 11 link, and add node 11 to 15 link.
- Remove node 4 to 5 link, and add node 9 to 14 link.
- Remove node 4 to 22 link, and add node 12 to 14 link.
- Remove node 4 to 25 link, and add node 14 to 19 link.
- Remove node 5 to 22 link, and add node 2 to 22 link.
- Remove node 24 to 27 link, and add node 14 to 20 link.
- Remove node 9 to 24 link, and add node 9 to 15 link.
- Remove node 21 to 28 link, and add node 15 to 28 link.
- Remove node 8 to 14 link, and add node 14 to 31 link.
- Remove node 16 to 25 link, and add node 16 to 18 link.

We run the SSR algorithm on the modified hubbed network topology with the minimum network degree increased from 2 to 3. Table 4.17 compares the results and network topology of the original AT&T network and the modified AT&T network.

The improvements can be explained in terms of how SSR operates. The backup path is selected using the OSPF Algorithm 2 (explained on Page 12), with the incremental backup capacity being the link cost. If each node has a minimum network degree of 3, we have more options available for the routing in the network to miminize this backup capacity cost. With the SSR algorithm and our minimum node-degree of 2 the routing options are limited even further. Remember that the set of links in the path that fails are set as tabu-links and this results in the cost for those links being set to infinity. This means that our selection for certain links on 2-connected nodes are restricted to the remaining

Table 4.17: The AT&T network properties and results

AT&T network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	67.16	% SSR	50.8	Δ % SSR	-16.36
$\bar{d}$	4.194	$\bar{d}$	4.194	$\Delta\bar{d}$	0
SD	2.28	SD	2.75	Δ SD	0.47
MD	3	MD	9.806	Δ MD	6.806

link on the node, regardless of the cost of this link. With a 3-connected node, even after the cost of one of the links are set to infinity, we still have a selection between the two remaining links.

We perform similar modifications to the BRITE generated topologies, N10D4BA2 and N20D2BA2. Tables 4.18 and 4.19 present the results for the modified N10D4BA2 network and the modified N20D2BA2 network respectively.

Table 4.18: The N10D4BA2 network properties and results

N10D4BA2 network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	56.67	% SSR	43.33	Δ % SSR	-13.33
$\bar{d}$	6	$\bar{d}$	6	$\Delta\bar{d}$	0
SD	1.632	SD	0	Δ SD	0
MD	2	MD	0	Δ MD	0

Table 4.19: The N20D2BA2 network properties and results

N20D2BA2 network					
Network	Original	Network	Hubbed	Network	Δ
% SSR	62.36	% SSR	65.03	Δ % SSR	2.68
$\bar{d}$	5.5	$\bar{d}$	5.5	$\Delta\bar{d}$	0
SD	3.487	SD	3.749	Δ SD	0.262
MD	7.5	MD	10.5	Δ MD	3

We now provide the improvement in SSR backup capacity requirements for the nodes under investigation, as well as the change in their network topology.

The improvement obtained by network re-engineering in some cases exceed and in some cases fall short of the SSR-SD results for improvement in required backup capacity. Note that there is no additional computation time required,

Table 4.20: The hubbed networks' SSR standard and maximum deviation investigation

Network	SSR6	EBone	AT&T	MCI	N10D4BA2	N20D2BA2
Δ %	-31.971	-41.07	-16.356	-27.181	-13.334	2.675
Δ SD	-0.541	-0.365	0.47	-0.376	-1.633	0.262
Δ MD	-1.173	0	6.806	1	-2	3

as the original SSR algorithm is used with the re-engineered network. This means that we can reduce the backup capacity requirements for the AT&T and MCI networks by 16.4% and 27.2% respectively, where the SSR-SD simulation was not computationally feasible. Also the re-engineered EBone network delivers a 41.1% backup capacity requirement improvement over the 9.1% of SSR-SD.

Chapter 5

Beowulf cluster-computing

5.1 An introduction to Beowulf cluster computing

Beowulf cluster computing yields a supercomputer built by using commodity off the shelf (COTS) computer systems. The required components are the personal computer, Ethernet networking, and the Linux operating system.

The personal computer has evolved over the past two decades to become a viable component for a supercomputing solution. The first IBM PC, in 1981, had an Intel 8088 CPU running at 4.77MHz, no hard disk and 64KB of memory. Moving 20 years forward, we compare it to a Pentium 4 Processor running at 1.7GHz, with a 80Gb hard disk, and system memory ranging of 1GB and higher. The Pentium 4 clock speed is more than 300 times faster than the original 1981 IBM PC. As the computing power kept growing the prices kept falling. Jeffrey Rayport [22] made a striking but inconclusive comparison between microprocessor development and the automotive industry. If the automotive industry kept pace with semiconductor development, a Rolls-Royce would cost $2.75 and get 3 million miles per gallon. Comparing it to the aircraft industry - a Boeing 767 would cost $500 and circle the globe in 20 minutes, using 5 gallons of fuel.

The Ethernet standard was developed at the Xerox Palo Alto Research Centre [23] by Robert Metcalfe who designed a networking system to enable all the PCs on site to connect to the first laser printer being built there. In 1976 Metcalfe and his assistant published the first Ethernet paper. Ethernet quickly became an industry standard. It was adopted by the LAN standards committee of the Institute of Electrical and Electronics Engineers (IEEE 802) as "IEEE 802.3 Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications". It was adopted by the International Standards Organisation (ISO) as a networking standard. The IEEE committee thereafter worked on increasing the transmission rate while using the same access method. They succeeded in increasing the 10Mbps transfer rate to 100Mbps. This 100Mbps Ethernet is often referred to as Fast Ethernet. Fast Ethernet was an important factor in Beowulf computing, in allowing fast communication between Beowulf cluster nodes.

Linux, and the open source movement, also plays a vital role in the development of Beowulf clusters. Linux originated by Linus Torvalds [24], at the time a 21 year old second-year computer science student. He wanted to use the MINIX [25] operating system as a basis for a new Unix-like operating system. Torvalds was in part inspired by Richard Stallman's GNU project [26] which aspired to providing free quality software. Stallman created the GNU C Compiler (GCC) as a basic tool striving to build a free operating system. The development of Linux started in 1991 as a hobby, and has grown into a force rivalling the Microsoft Windows operating system. The two most important factors in the success of Linux are that it is free and it is stable. Supercomputers have always been extremely expensive closed architecture machines. No-one could conceive of rivalling these machines using PCs. However, the desktop PCs of today rival the supercomputers of several years ago. The bottleneck created by these processing nodes communicating cheaply has been alleviated by fast Ethernet. For each node to operate, we need an operating system. If the operating system had to be bought, the costs would grow linearly with each node added. Enter Linux, with no operating system (or even application software) cost for each node. Scientists at the NASA Goddard Space Flight Centre realised these advantages and combined them to form the first Beowulf cluster, called Wiglaf. This Beowulf consisted of a 16-processor system with Intel 80486 66MHz processors. These soon became 100MHz DX4 processors. They achieved a sustained performance of 4.6Mflops per node (76 Mflops total). From there the Beowulfs have multiplied and are in use in research centres, universities and schools.

5.2 Setting up a Beowulf cluster

The setup of the Beowulf cluster can be divided into 3 sections:

- node requirements and setup
- network topology and configuration
- Linux configuration.

5.2.1 Node requirements and setup

A Beowulf node differs from a typical desktop PC. A Beowulf node's main (and usually only) responsibility is to perform the activities and capabilities associated with application execution. Therefore we mostly require the following from our Beowulf nodes:

- instruction execution
- high speed temporary information storage
- high capacity persistent information storage
- communication with other nodes, and possibly external nodes.

From these requirements it follows that the node's processor, memory, storage and networking specifications are of particular importance. The processor can range from a Pentium to a Pentium 4, or their binary compatible AMD/Cyrix processors. The memory and storage requirements will vary according to the application of the Beowulf cluster. The best price/performance ratio for the networking layout would be using Fast Ethernet. Therefore we require a standard 10/100Mbps network interface card (NIC) for each node.

Due to the expandability of COTS PCs we can easily add or upgrade the components required for each node. Special mention should be made of the "world" node, see Figure 5.1 below. This node acts as a router between the external network and the internal Beowulf network. It will also be the storage node for the applications and data that will be shared among the Beowulf nodes using NFS. We will discuss this in the Linux configuration section on Page 63. The world node will therefore require another additional NIC, as it interfaces to two networks: the external network and the internal Beowulf network. The world node will also require more storage space than the other nodes, due to the NFS mount points from this node. Additional storage can be added as required.

5.2.2 Network topology and configuration

In the layout of the Beowulf network, we have the following three options:

- an isolated Beowulf network
- an open Beowulf network
- a connected Beowulf with a single entry point.

We will implement the connected Beowulf with a single entry point. An isolated Beowulf network can be defined as having all nodes connected to one another but no node is connected to the external network. An open network is defined as having each of the nodes connected to the external network that it forms a part of. The reason for chosing the single entry point option is that an isolated Beowulf network would mean needing physical access to the cluster each time it is used. On the other hand, an open Beowulf network is unnecessary, and a security risk. The world node, being the single entry point to the Beowulf, is the only "interactive" node. We can submit all our jobs, and view the output here. The other nodes are processing points requiring no direct access. The network topology for our Beowulf will thus be as in Figure 5.1

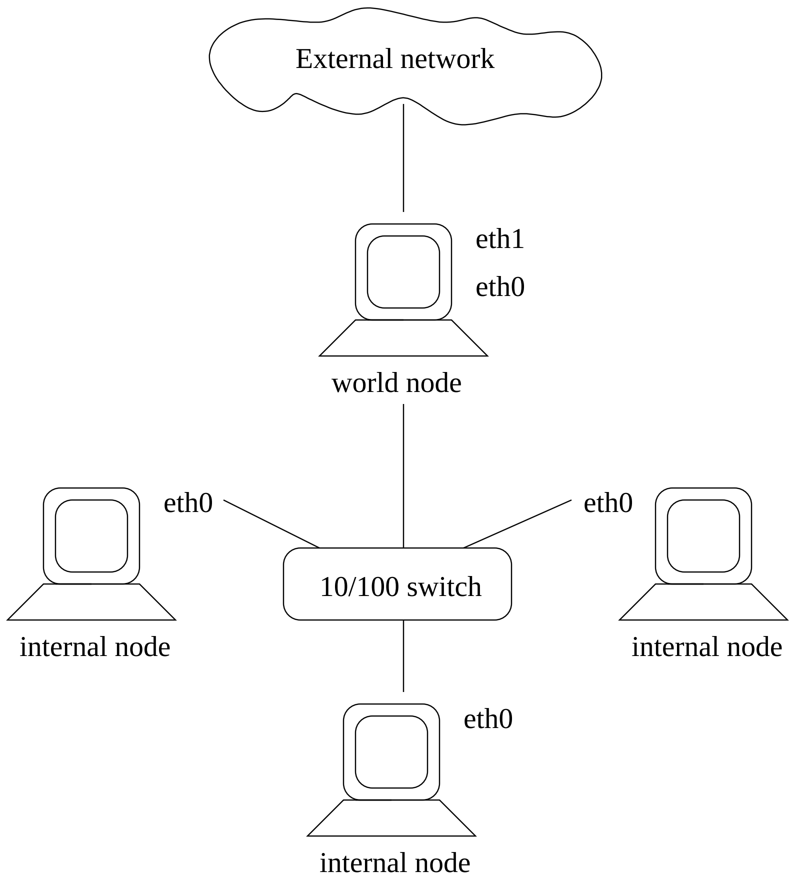

Figure 5.1: A typical Beowulf network topology.

With reference to Figure 5.1 we will use the first interface (eth0) on all the Beowulf nodes, to connect to the internal network, and the second interface (eth1) for the world node only, to connect it to the external network. For our initial setup we have only one node, the world node, with a second network interface. We can however, if required, attach any node directly to the external network by adding a second interface.

The IP addressing scheme is dictated by the network topology chosen. In our topology we have control over our internal network. The following three IP address ranges have been reserved for use by private networks:

- 10.0.0.0 - 10.255.255.255
- 172.16.0.0 - 172.31.255.255
- 192.168.0.0 - 192.168.255.255

These address ranges are never assigned for Internet use, and packets carrying these addresses will not be forwarded by Internet routers. The IP address range from 192.168.1.0 - 192.168.1.255 will be used by our network. Note that the addresses 192.168.1.0 and 192.168.1.255 are reserved.

There are considerations in naming our nodes. The scalability of our Beowulf is an issue. Managing a 5-node Beowulf cluster is possible. A 10-node cluster already presents a challenge for the replication of node configurations. A Beowulf cluster growing beyond this size can impede on the efficient management of the cluster configuration. Therefore we use a standard naming scheme. We prefix all our node host names with beo and concatenate it with the node number, starting at 1. Why not use 0? Going back to the previous paragraph, we have explained that the host address 0 and 255 are reserved. We therefore start our IP address assignment from 1. The host name for the external node will most probably be determined by the network administrator. Note that you can however still determine the second interface host name, in other words the one connecting to the Beowulf network.

We would like our host name number to correspond to the IP host address for that number. For example, using our IP address range beo1 would have IP address 192.168.1.1, and this is the Beowulf network interface host name for the world node. The cluster nodes will be next starting with host name beo2 and IP address 192.168.1.2. Furthermore the management process will be alleviated by using scripts to manage the nodes. If we have a standard naming scheme we can easily address and configure these nodes using scripts.

5.2.3 Linux configuration

The nodes cannot function without an operating system. As explained in the introduction, Linux was a vital component in the advent of Beowulf cluster computing. Due to the phenomenal growth of Linux many different distributions are available. We need to configure our Beowulf to be as simple as possible and conform to a standard. We therefore keep the different Linux distributions used to a near minimum. Red Hat Linux 9 and Mandrake Linux 9.2 were the two distributions chosen. The reason for choosing both lies in hardware support and the applications available. Since our Beowulf cluster is composed of COTS systems, we have many hardware components from different vendors. The probability of getting a node up and running is higher if we use two main Linux distributions.

Not only do these Linux distributions provide the necessary platform for application execution, they also have the necessary tools needed to run a Beowulf cluster. We will discuss these tools:

- the user administration, used to create global user(s) for different levels of Beowulf cluster access
- the Network File System (NFS), used to create and mount share points

- Perl, the scripting language used to dispatch given jobs to given processors
- Secure Shell (`ssh`), used to log into nodes, and execute applications on their processors.

5.2.3.1 User administration

We need a user account to gain access to any Linux machine, whether locally, or across the network. The user account specifies what resources we have access to. A normal user account and an administration account are required. For administration it is straightforward – we need root access to setup hardware, install applications, modify system files and set permissions. It is recommended to keep the same root password for all Beowulf nodes.

We also need a general Beowulf cluster user account for the users executing applications on the Beowulf cluster. As we will explain in the ssh section 5.2.5 on Page 67, this user account should be configured with the same username and password across all the Beowulf nodes. This enables easily automated ssh access to all Beowulf nodes.

5.2.3.2 NFS

NFS is used to mount a file system on a remote computer as if it were local to your own machine. NFS operates over a TCP/IP network. Setting up NFS involves the following:

- specify the shares in the /etc/exports file
- set the security permissions in the /etc/hosts.deny and `hosts`.allow files
- run the three daemons `rpc`.mountd, `rpc`.nfsd, and `rpc`.portmapper
- test the share mount points using `mountd`
- set up automatic mounting in the /etc/fstab file.

Setting up the NFS shares involves the client and server configurations. We will start with the server. The /etc/exports file contain entries in the following format:
`dir`ectory machine1(option11,option12) machine2(option21,option22). The directory above would be the local directory that is to be shared. The machine option will be a list of machines allowed to access the share. There are various options, see [27] for details. It is important to decide whether to allow read-only (`ro`) or read-write (`rw`) access.

For example suppose that the server 192.168.1.1, wishes to share two local directories /home/BeoShare and /home/BeoResult. We store the executable program files and their input files in BeoShare. We want the nodes to pipe their processing results to BeoResult.

For these purposes we will make BeoShare read-only to prevent any malicious or accidental damage to our program files or their input. BeoResult will be read-write, to ensure that the processing nodes can pipe their output to one central storage location on the server. We therefore have the following entries in /etc/exports:

```
/home/BeoShare 192.168.1.0/255.255.255.0 \
  (ro,sync,no\_root\_squash)
/home/BeoResult 192.168.1.0/255.255.255.0 \
  (rw,sync,no\_root\_squash)
```

Next we set the security permissions for the shares. There are two files involved namely /etc/hosts.deny and /etc/hosts.allow. Note that the server allows a request as follows:

- Check if the machine is in hosts.allow. If it is, allow access.
- If it is not in hosts.allow, check hosts.deny. If the machine is listed here, deny access.
- If the machine is listed in neither file, allow access.

From this we can see that it is better to lock the share access as strictly as possible and then explicitly allow access. We begin by modifying hosts.deny as follows:

```
portmap:ALL
lockd:ALL
mountd:ALL
rquotad:ALL
statd:ALL
```

This blocks all machines from using the NFS services. Next we explicitly allow access to all services to our 192.168.1.0 class C subnet in /etc/host.allow:

```
ALL: 192.168.1.0/255.255.255.0
```

Having set up the shares and allowed access to our machines, we set the shares to be mounted automatically at boot time on each node. For this we need to modify the /etc/fstab file on each node by adding:

```
192.168.1.1:/home/BeoResult /mnt/BeoResult nfs
  user,exec,suid,rsize=8192,wsize=8192,bg 0 0
192.168.1.1:/home/BeoShare /mnt/BeoShare nfs
  user,exec,suid,rsize=8192,wsize=8192,bg 0 0
```

These shares will now be mounted at boot time to /mnt/BeoShare and /mnt/BeoResult. Any changes or additions to the shares may be configured by editing the NFS server's /etc/exports file and editing each client node's /etc/fstab file, and then rebooting, or restarting NFS.

Now that we have centralised the storage of our programs, input files, and output files, management is made much easier. We need only compile our program in one location, and we have all the output from each processing node stored in one location.

5.2.4 Perl

Now that we the nodes are ready, and the shares are in place to centrally access applications and store output, we need a dispatcher to send the jobs to the nodes as they become available. Doing this manually is not only a cumbersome task, but will also lose valuable processing time in detecting free processors and dispatching new jobs to them manually. We will use a Perl script to act as our dispatcher. We use dispatcher given in [28] as a basis. Perl is both a scripting and a programming language. From Perl in a Nutshell [29]:

> Perl is especially popular with systems programmers and web developers, but it also appeals to a much broader audience. Originally designed for text processing, it has grown into a sophisticated, general-purpose programming language with a rich software development environment complete with debuggers, profilers, cross-referencers, compilers, interpreters, libraries, syntax-directed editors, and all the rest of the trappings of a "real" programming language.

The Perl script is called prun and must be available in the application share. We also need to give the Perl script a list of commands to execute (cmd_list), and a list of processors to execute them on (proc_list). The contents of cmd_list are typically:

```
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModelBeo1 \
  > /mnt/BeoResult/OSPFModel1Res
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModelBeo2 \
  > /mnt/BeoResult/OSPFModel2Res
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModelBeo3 \
  > /mnt/BeoResult/OSPFModel3Res
```

From above we see that we execute the ospfsim application 3 times. Each time we give it a different input file (OSPFModel1 to OSPFModel3), and redirect the results to the corresponding output files.

The proc_list file contains a list of the host names of the nodes in the Beowulf cluster, with each host name on a separate line:

```
beo1
beo2
beo3
```

Note that we can provide different users with different processor lists, thereby allowing different levels of resource access. We now execute the Perl script with these two files as input. The syntax is:

```
./prun proc_list cmd_list
```

There is one more hurdle to overcome before we can execute applications across the Beowulf cluster. The Perl script automates the dispatching of the processes to processors. The way in which these applications are executed on these processors is by using the secure shell (ssh). We discuss this next.

5.2.5 ssh

ssh is a program used to securely log in to a remote machine and execute commands on it. Seeing as this program allows access to any machine using ssh on the network, strict security measures are necessary. If we use our Perl script as is to run processes on the processors, we need to authenticate ourselves each time we access a processor using ssh. Provision has been made for this, and there are several authentication methods available to us. The RSA based authentication method was chosen as it can be used transparently. Each machine needs to maintain a list of public keys of other hosts permitted to log in. We use the key generator ssh-keygen to create the RSA key-pair. In our example below we show how to set up automatic ssh access to a new Beowulf node for the server. Suppose that beo1 is our server (world node), and that we are adding automatic ssh access to beo2.

On beo2 we do the following:

1. execute $ ssh user@beo2 to access a command shell on beo2. Note that the user would match the username set up for Beowulf execution use as in Section 5.2.3.1 on Page 64

2. execute $ ssh-keygen -t rsa to generate the key-pair. Accept the defaults at the prompts

3. execute $ cp .ssh/id_rsa.pub ./ssh/authorized_keys to place the public key in the right location

On beo1 we do the following:

1. execute $ ssh-keygen -t rsa to generate the key-pair. Accept the defaults for the prompts. One should only perform this step if you have not

generated a key-pair on this node before. If you have, abort the ssh-keygen process and use the already existing id_rsa.pub file in the next step.

2. execute $ cat .ssh/id_rsa.pub | ssh beo2 "/bin/cat \
>> .ssh/authorized_keys" to add the server's public key to the list of authorised public keys for node beo1

Whenever you ssh to node beo1 you will be automatically authenticated and allowed access.

5.3 Usage of the Beowulf cluster

The usage of the Beowulf cluster is simple. If the reader has any questions regarding concepts in this section, consult the Beowulf cluster setup section 5.2 on Page 60. We first look at the information that the user will be provided with:

- The external host name and/or IP address of the Beowulf cluster world node.
- The `username` and `password` to access the Beowulf cluster.
- The locations of the program and the output shares.
- The location of the dispatcher Perl script and processor list.

The IP address and username will together allow ssh access to the Beowulf cluster from any location on the external network. The dispatcher Perl script will be used to automate the application execution process. The processor list contains the names of all the processors available to the user to execute applications on. The applications themselves will be stored in one central location, and their output in another.

For example, suppose that we were provided the following information:

1. The world node external network host name is heathcliff.
2. The world node internal Beowulf host name is beo1.
3. The username and password is beouser / hr0thgar.
4. The application share is located in /mnt/BeoShare, the output share is located in /mnt/BeoResult.
5. The location of the dispatcher Perl script is /mnt/BeoShare/prun, and the processor list is in /mnt/BeoShare/proc_list.

First we need access to the Beowulf cluster through the world node. Using our given username beouser, we ssh to the world node heathcliff as follows: $ ssh beouser@heathcliff. Now in keeping with our Beowulf naming scheme, we ssh to the world node's internal network host name:
$ ssh beouser@beo1. Next we change directory to our application share:
$ cd /mnt/BeoShare.

Assume that we want to run a simulator application called ospfsim, and we want to do 3 different runs using the following input files: OSPFModel1, OSPFModel2, OSPFModel3. In the same directory location as the Perl Script and processor list, we create the command list file cmd_list whose contents are:

```
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModel1 \
  > /mnt/BeoResult/OSPFModel1Res
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModel2 \
  > /mnt/BeoResult/OSPFModel2Res
/mnt/BeoShare/ospfsim /mnt/BeoShare/OSPFModel3 \
  > /mnt/BeoResult/OSPFModel3Res
```

We next execute the command list on the world node using:
$ ./prun proc_list cmd_list.

Basic information output is given as the processes are dispatched. The current command being executed is shown on the world node. Once the application run is finished, the user will receive a message confirming completion. All the output results are stored in one location. For example, we change directory to $ cd /mnt/BeoResult. Here we have the application output files OSPFModel1Res, OSPFModel2Res, OSPFModel3Res in one location, ready to be processed.

5.4 The basic Beowoulf cluster

The realisation of the Beowulf cluster is so effective and easy it seems to be an obvious development. The computing power placed in the hands of research and educational facilities offers easy access to viable high end computing. A Beowulf cluster is open to user control. Considering the rise of the Linux operating system and the growing appeal of its open source and standards-based nature, and combining this with COTS hardware, we see encouraging price / performance ratios for Beowulf cluster super computers. Adding more computing power to increase the Beowulf cluster's performance is made affordable and easy by COTS hardware.

Beowulf cluster computing is still a relatively new phenomenon. At present we can dispatch various application runs over an array of Beowulf nodes. As the need for more powerful parallel programming grows, the standardised message passing interface (MPI) [30] may be used to build highly parallelised applications. The most immediate improvements to be made the Beowulf cluster are to automate the node maintenance, and to ensure maximum uptime of the nodes. The physical placement and cooling of the Beowulf cluster needs to be addressed as the cluster grows.

We are now able to execute more simulations in the same given time, increasing the application output sample space, thereby greatly improving our confidence intervals for these simulations.

5.5 Beyond the basic Beowulf cluster

We have introduced Linux Beowulf cluster computing and explained how to build and use such a Beowulf cluster, which will be referred to as the cluster. After the initial setup of the cluster, its attributes and requirements have changed. The main attribute change is the network topology. The machines made available to the cluster are powerful Xeon dual-processor machines. It was decided that these computing resources should be made widely available. The cluster evolved from a connected Beowulf network with a single entry point, to an open Beowulf network, where each node in the cluster, which will be referred to as a node, is also available on the external network. The idea is to maximize the usage of the processors, while prioritizing simulation processes on the cluster. We therefore implemented priority scheduling to ensure that simulation runs receive the highest priority, and thus the most processor allocation. This was implemented using the following mechanisms:

- controlling the number of secure shell (`ssh`) sessions per user
- ensuring that the simulation run receives the highest priority on each processor
- lowering the priority of external users logged on to the machine, but only if simulation runs are executed on it.

These three mechanisms work together to ensure optimum processor allocation to the simulation runs executed on the cluster. We implemented these mechanisms at the user and / or group level. It is important to standardize the user accounts and system configuration across our Beouwlf cluster. Before implementing these mechanisms, we focus first on automating the user administration and system configuration in the next section. We then focus on the three priority scheduling mechanisms in detail in Section 5.7 on Page 72

5.6 Linux administration

The reader is referred Section 5.2 on Page 60 on the principles involved in building the cluster. We will extend the section on user administration, as this is crucial to the proper working configuration of NFS. Before we can effectively automate the user administration process, we need to make certain system configuration changes. We discuss this system administration, and then the user administration.

5.6.1 System administration

Before we can fully automate the cluster configuration and administration, we need to address the use of ssh, which is central in allowing access to the nodes. The steps involved in trusted ssh access have already been discussed in Section 5.2. Non-interactive ssh access is implemented through the use of RSA-keypairs, with the public keys shared between machines. In this report we will only present beorsakey, the automated version of this setup procedure, in Figure A.1 on Page 83.

Note that automated ssh access is only allowed for the user specified in the script parameters. Furthermore, the user account executing this script must correspond to the user account specfied in the script parameters. The automated access is one-way from the master node to all other cluster nodes. Root access to the master node can still be privileged, even if root access is granted to other cluster nodes.

5.6.2 User administration

We create Linux user accounts to allow uniform access to all of our nodes. The user account enables accounting of the resources used and more importantly, allows secure access to the cluster. We will explain why we need to replicate the same user and group setup across all nodes. Replication of the setup for the cluster user, referred to as beouser, is handled by beouseradd. See Figure A.2 on Page 84.

The parameter checking has been abbreviated in this example, but can be extended if necessary. Note that the last parameter node_list supplies the name of a text file containing the host names for all the nodes, with each host name in node_list contained in a separate line. This file, combined with ssh access to each of these nodes, serves to replicate the configuration across all nodes. We use the same concept to replicate the configuration files to all nodes with beofilerep (see Figure A.4 on Page 86). This will be discussed later.

The groupadd and useradd commands, together with the options as specified to beouseradd are used to create the necessary user and group. After being created, we need to set the new user account password. This can be done using the passwd command. It was necessary to modify the behaviour of passwd so as to input the new password token in a non-interactive way, avoiding inputting the same password twice for each host contained in node_list. The --stdin option provides this functionality, allowing us to pipe the password token with the echo command into passwd. As mentioned, the user setup is crucial to the proper working of NFS [27]. NFS allows file access through standard UNIX user and group file permissions. These user and group file permissions are, however, communicated by using the userid (uid) and groupid (gid) supplied by the client. For this reason we need to ensure that our uid for the user beouser, and the gid for the group beousers match across all nodes to the master node. Our nodes use uid 600 and gid 701.

5.7 Priority scheduling

Any multitasking operating system includes process switching. This means that we switch execution from one process to another in a very short time frame, making it appear as if these processes are executing simultaneously [31]. The Linux operating system is no exception. We will look at Linux process scheduling, which is concerned with when to switch which processes. Priority scheduling refers to prioritizing the scheduling process to suite specific processing needs.

5.7.1 Controlling ssh sessions

Users of the cluster, or individual nodes, will gain access via ssh. Whatever their processing needs, we cannot assume that users will never try to abuse the system. We will allow ssh access to any machine, but limit the users to a maximum of 2 logons per machine. If we take into account that any cluster by definition should contain any number of machines, then this restriction is diminished as the cluster grows bigger. In effect this helps us to load balance power users over the cluster. To enforce this ssh logon limit, we use pluggable authentication modules for Linux (Linux-PAM or PAM). PAM enables us to customize a security scheme for our machine at a per application and per user level, as described [32]:

> It is the purpose of the Linux-PAM project to separate the development of privilege granting software from the development of secure and appropriate authentication schemes. This is accomplished by providing a library of functions that an application may use to request that a user be authenticated. This PAM library is configured locally with a system file, /etc/pam.conf (or a series of configuration files located in /etc/pam.d/) to authenticate a user request via the locally available authentication modules. The modules themselves will usually be located in the directory /lib/security and take the form of dynamically loadable object files.

On our nodes the PAM library is configured as system files within the /etc/pam.d directory. Here we focus on the configuration file /etc/pam.d/sshd. It configures the privileges for sshd, the ssh daemon allowing ssh sessions.

sshd listens for new connections from ssh clients. Each time a client tries to connect, sshd forks a new daemon for each connection. It also loads the PAM libraries and uses the application's PAM configuration file to authenticate and validate the ssh client.

We do not need to modify this configuration file beyond the default settings. Note however that we need the 4 modules (auth, account, password, and session) to be required, and using the system-auth service. We use pam_stack.so to be able to call from one service, the stack of another service. This enables the use of one system-wide setup shared by multiple services. The contents of /etc/pam.d/sshd are as follows:

```
#%PAM-1.0
auth     required   pam_stack.so service=system-auth
auth     required   pam_nologin.so
account  required   pam_stack.so service=system-auth
password required   pam_stack.so service=system-auth
session  required   pam_stack.so service=system-auth
```

We execute the system-wide setup of the user limits. This is specified in the PAM limits configuration file /etc/security/limits.conf. This is the default file location on a Mandrake Linux 9.2 distribution, but can differ on other Linux distribution. In limits.conf we define limits for users using the following syntax:

```
<domain>    <type>    <item>   <value>
```

The <domain> specification sets the domain of the restrictions for:

- username: a user, based on a username,
- @groupname: a group, based on a groupname,
- *: a wild-card, for the default entry,
- %: a wild-card, used for the maxlogins item only, can also be used with the % group syntax.

The <type> specification sets the limits to be enforced as:

- hard: on hard limit set here is enforced by the Linux kernel. The user cannot increase her requirements above this hard level,
- soft: a soft limit enables users to raise or lower their requirements, within a range set by hard limits,

- -: used to enforce hard and soft limits together.

There are various <item> options to set limits on their corresponding resources. We focus on:

- maxlogins: the maximum number of remote logon sessions that can be made to the machine,
- nproc: the maximum number of processes that may we created by the user on the machine.

In <value> we specify the value of the limit on each item. It is important to note that these restrictions apply per login session, and that user level restrictions have priority over group level restrictions. Our basic configuration is as follows:

```
@beousers       hard    maxlogins       20
@beousers       hard    nproc           100
*               hard    maxlogins       2
*               hard    nproc           10
```

We allow the beousers group members 20 simultaneous ssh sessions, and each session can create 100 processes, while all other users are allowed 2 simultaneous logon sessions, with 10 processes allowed to each. The 20 allowed simultaneous logon sessions for beousers should be sufficient, seen in the context of how the simulation runs are distributed and executed. Once an ssh session is initiated with a node, we execute the first waiting command in the command list. The dispatcher script only executes another command on this node once it has completed the currently executing command. Therefore we only have one running ssh session at any given time. It is necessary to enforce priority scheduling now that our commands (processes) are executing across the cluster on all the nodes.

5.7.2 Simulation run priority

Every command that is run on any of our nodes has a priority of 0 by default. This priority value can range from -20 (the highest) to 19 (the lowest). This priority value is governed by the use of the nice command, which runs a value supplied as a parameter to nice, with an adjusted scheduling priority. The default adjustment value of 10 lowers the priority to 10. This default behaviour is where the command name nice is derived from. The idea is that users can run non-critical applications with a lower priority using nice in order to give more processing cycles to the other users of the system. We are doing the opposite. We want to ensure that our simulation receives the highest priority. We do this by setting the adjustment value to -20. There is one problem however. The beouser account does not have the privilege to increase the scheduling priority, we can only lower the priority at present. We need root access to increase scheduling priorities.

All users cannot be entrusted with root access to our cluster. Not only is this a security risk, it is also ineffectual. It is only required to execute specific commands with root privileges. For this purpose we use the sudo command. It enables authorized users to execute permitted commands as the superuser (or any other user) as specified in the sudoers file. We edit the sudoers file, so as to allow us privileged use of nice, to execute all commands on our cluster with higher priority. We next look at configuring the sudoers files and executing our commands with higher priority.

5.7.2.1 Configuring the sudoers file

In the Mandrake Linux 9.2 distribution sudoers is located in etc/sudoers. We do not edit this file directly, however. We use the visudo command, acting as an interface to the sudoers file. It locks the file and prevents simultaneous edits, provides sanity checks, and checks for parse errors. By default visudo uses the editor vi. Another editor may be specified as parameter to visudo.

We now discuss the syntax of the sudoers file (Figure A.3 on Page 85). We focus on the user privilege specification as starting point. It contains these two specifications:

```
root ALL=(ALL) ALL
BEOUSERS BEOHOSTS = NOPASSWD: NICE, RENICE, SUDO
```

The first specification is standard and allows the root user on any host permission to execute all commands. We focus on the second privilege. The NOPASSWD: modifier prevents the following sudo command default behaviour: sudo prompts the user running a permitted privileged command for the user's account password. This presents the same problem as encountered with ssh access to our cluster nodes – the automation is rendered useless if we are interactively prompted for a password every time a command is executed. NOPASSWD: prevents these password prompts and automatically allows sudoers to perform command execution. The other parameters in the user privilege specification are all aliases, defined in their own sections. An alias has the following syntax:

```
Alias_Type      Alias_Label     Alias_Details
```

We now show the three types of aliases and give an example of each:

- User_Alias which specifies an user list, e.g. User_Alias BEOUSERS = beouser, beoadmin
- Host_Alias which specifies a host list, e.g. Host_Alias BEOHOSTS = beo1, beo2, beo3

- Cmnd_Alias which specifies a command and it's file location, e.g. Cmnd_Alias NICE = /bin/nice

We can specify different permutations of these aliases in the user privilege section to effect different sudo execution permissions. Looking at our sudoers privilege specification again, we allow the user accounts BEOUSERS, running on BEOHOSTS, a list of commands to execute (in this case NICE, RENICE, and SUDO), without a password prompt. We end this section by discussing the utility beofilerep to replicate the sudoers file across the cluster nodes (see Figure A.4 on Page 86).

Usage of the script is simple if seen as a wrapper for the scp command namely to:

- Supply beofilerep with the file(s) to be copied in the local_files parameter.
- Specify where it is to be copied to with the remote_dir parameter
- Lastly node_list points to the text file containing a list of hosts to which the file(s) will be copied, with each host in a seperate line.

Note that the script detects the user executing the beofilerep command and uses this user account to copy the file(s) to the cluster nodes. This user account determines the local read permissions for local_files, and the remote write permissions for remote_dir on each node. Seeing as this administration utility needs privileged access, it has to be executed as root.

5.7.2.2 Using the sudo command

We can now execute nice using root privileges, as follows:
sudo nice –adjustment=-20 "command". We need only prefix this to each command in the command list (cmd_list) supplied to our job dispatcher, and it will be executed with the highest priority on each cluster node.

5.7.3 Bumping external user priorities

The remaining task is to lower the process priority for all other users of the nodes. We manage this by using the renice command. It is in effect the same as the nice command, except it governs the priority of processes already running. Refer to Subsection 5.7.2.1 on Page 75, where we configured sudoers to grant the renice command the required priveleges to adjust process priorities both ways. We now discuss the beoprioritize script (see Figure A.5 on Page 87).

The renice command can alter the scheduling priority of one or more running processes. This task is further aided by renice allowing us to specify a user, for whom all his running processes are altered. Therefore we need to know which users are currently logged on to our system. The users command provides this information. The list of logged on users supplied by users is processed by awk to convert this list with users to a format for use with the cut command. Using cut with a bash for-loop, we lower the process priority for each of the logged-on users, except root and beouser, using renice with the -u option.

This is not a complete solution yet. renice only affects the current running processes. If an user logs on after the simulation run has been started, their process priorities will be assigned the default value of 0, and will not be lowered. It is necessary to periodically lower other user's process priorities, and once the simulation run has been completed, to restore them to normal. If we need to periodically schedule a command, we use the cron command. cron [33] uses the crontab command to schedule commands to be executed by cron. The crontab command manages the file located in /var/spool/cron/$username, where $username is the name of the user account from which the schedule was created. We do not have permissions to edit this file directly, so we can only use crontab to schedule commands.

Fortunately there is a non-interactive method of supplying crontab with entries to be scheduled: execute crontab filename, with filename being any text file with schedule entries each in a separate line. The syntax for these entries is:

```
mins hrs day-of-month month weekday cmd
```

To schedule our prioritize script to run every 5 minutes, and lower external user process priorities to 19, we edit crontabfile to contain

```
5 * * * * /mnt/BeoStore/prioritize -u 19
```

and then execute crontab crontabfile. This configures the cluster so that while the simulation is running, we also have the scheduled priority control script beoprioritize running. Once the simulation run is finished, we need to stop adjusting external user process priorities, as they are now allowed full use of the processing resources again. For this purpose execute crontab with the -r option. There is one more consideration, the currently running user processes which priorities have been lowered. We need to restore these back to the original default.

We discuss the beocronsched script (see Figure A.6 on Page 88), which takes care of both tasks for us. With the -a option given, we schedule our beoprioritize script, and the -d option removes the beoprioritize script from the schedule. The current version of beocronsched does no more than the crontab command itself. The idea is to extend beocronsched for the following scenario. If at any stage we decide we need to schedule tasks for beouser other than beoprioritze, we cannot simply add and remove the scheduled tasks with crontab. We need to carefully add and remove the crontabfile entries, leaving any other scheduled tasks untouched. This can be performed by beocronsched

All that remains for managing the priority scheduling of external users is to combine the above scripts for use in the beorun script (see Figure A.7 on Page 89). We begin by lowering the process priorities for the external users, and then scheduling this action to take place every five minutes. Then we execute the simulation run with the Perl dispatcher script. Once it has completed, we remove the process priority script from the scheduler, and resume executing all current running processes at the default priority. The date command logs the start and finish of the simulation run into a simulation result file, to keep track of processing time for each simulation run.

5.8 Summary

We extended the basic cluster to provide priority scheduling to ensure optimal processor allocation for beouser simulation runs. It was clear that we need a uniform system configuration across the cluster. The development of a set of scripts, referred to as beoscripts, has automated this process. The UNIX philosophy of keeping programs and systems simple, focused, and reusable has been effective and helpful in achieving our priority scheduling goal. Linux commands and files interact to allow the easy implementation of our administrative requirements. The beoscripts utilities can be combined and adapted to suit the needs of different cluster requirements. These scripts, together with the Beowulf reports, are available online [34].

Further adminstrative work for the Beowulf cluster includes the following main targets, which may be adjusted as the Beowulf cluster enjoys higher and more varied usage:

- Implement node failure recovery, including restarting terminated processes on failed nodes.
- Configure beorun to allow simulation run resumption after an interruption (e.g. power failure).
- Setting a limit on memory usage for external users, acceptable to both the external and cluster users.
- Using logon limitations to set an upper limit on the number of concurrent simulation runs by beouser.
- Enhancing the progress feedback to beouser from the process dispatcher.

Chapter 6

Conclusion

This thesis has investigated current link failure recovery schemes and developed a link failure recovery scheme using the Successive Survivable Routing (SSR) algoritm. We have looked at network theory and the development of the modified Dijstra algoritm which is a crucial component in the selection of optimal edge-disjoint OSPF (ED-OSPF) path pairs. Link disjointedness is a requirement for the SSR algorithm.

We investigated teletraffic engineering and specifically Multi-path Label Switching traffic engineering (MPLS TE). We also focus on MPLS TE global path protection.

The advantages of MPLS TE global path protection are:

- It is well suited for large-scale networks with a limited number of LSPs to protect. We specify the LSPs to protect in the SSR algorithm using the link failure matrix.
- The backup tunnels to be used as recovery LSPs are computed and signalled before the failure, which is an important part of a globally optimised backup path set. The SSR algorithm addresses the computation and optimal selection of the backup path set for the network.

The drwabacks of MPLS TE global path protection are:

- Global path protection requires the doubling of the number of TE LSPs, which has a significant impact in full mesh networks. In our investigation of "real-world" network topologies and the theories to why such topologies came into being, we have shown that preferential connectivity applies, which reduces the total number of network links.
- In international networks, generally global path protection recovery time is measured in tens of milliseconds, which is a problem with time sensitive traffic like voice and video data. This requires further investigation.

- The requirement of a bandwidth guarantee means that an external off-line tool is required for the computation of both the primary and secondary TE LSPs. We have shown how SSR is a well-suited tool for this purpose.
- The requirement for end-to-end diversely routed backup paths could imply the selection of a nonoptimal path for the primary TE LSPs. We have shown that the ED-OSPF algorithm ensures that we select the optimal pair of network paths, and have shown that it succeeds where the two-step (2SA) algorithm fails.

The above has shown that SSR is a viable tool for investigating global path protection. We investigated and extended the SSR algorithm to include capacity giveback and state-dependent SSR (SD-SSR) [35]. We have discovered that the computational requirements for SD-SSR are not scalable; therefore SSR cannot be deployed as a tool within larger sized networks. It could however be argued that the SSR algorithm is an off-line tool, which would permit it additional computation time. We also presented the Beowulf clustercomputer, which allows us to distribute the computation of the SSR algorithm across any number of cluster nodes to speed up the processing time.

To expand on the results of the SSR algorithm [20] which were tested on example networks, we developed "real-world" network topologies [3] and generated synthetic "real-world" topologies [4]. We discovered that the SSR-algorithm does not perform as well on these "real-world" topologies. We investigated the properties of the "real-world" topologies, and discovered that through network engineering we could gain a significant reduction of backup capacity requirement without any penalty to the SSR algorithm computation time. We mainly achieve this by enforcing a minimum node-degree of 3 for all nodes in the topologies under investigation.

6.0.1 Further work

Though the SSR algorithm is capable of dealing with multiple link failures, this thesis has focussed on single-link failures. The performance of the SSR algorithm with multiple link failure scenarios requires further investigation.

The gains made through network engineering can also be developed further. The "real-world" American, European and Australian models can be combined with African and Asian "real-world" network topologies to create a "real-world" global network for evaluation of the SSR algorithm performance.

Furthermore, the SSR algorithm and "real-world" network topologies could be incorporated into existing, well supported discrete event simulators. The ns [36] and OMNET++ [37] discrete event simulators offer built-in MPLS simulation models, and are well-suited for extension to include SSR algorithm based routing.

Appendices

Appendix A

Appendix

The appendix contains the scripts discussed in Chapter 5 on Page 59. To modify the working of these scripts, the reader can refer to the on-line manual pages of these commands by using the `man` Linux command. Type `man` followed the the Linux command you wish to use and it explains the command as well as it's parameter use.

```
clear
echo '/-----------------------------------\'
echo '| beorsakey                         |'
echo '|  - utility allows non-interactive  |'
echo '|    access from a master node to a  |'
echo '|    list of hosts as specified in a |'
echo '|    text file for the user that is  |'
echo '|    specified                       |'
echo '\-----------------------------------/'

if [ "$1" = "" ]; then
  echo No user account name given!
  echo Usage: rsakey user masternode node_list
  exit 1
elif [ "$2" = "" ]; then
  echo No master node host name given!
  echo Usage: rsakey user masternode node_list
  exit 2
elif [ "$3" = "" ]; then
  echo No node list given!
  echo Usage: rsakey user masternode node_list
  exit 3
fi

if [ -f $3 ]; then
  if [ -f $HOME/.ssh/id_rsa.pub ]; then
    echo 'Master node: id_rsa.pub file exists.'
  else
    echo 'Master node: creating id_rsa.pub file.'
    ssh-keygen -t rsa
  fi
if [ ! -f $HOME/.ssh/authorized_keys ]; then
  echo 'Master node: creating authorized_keys file.'
  cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys
else
  echo 'Master node: authorized_keys file exists.'
fi

for host in $(cut -f1 $3); do
  if [ $2 != $host ]; then
    ssh $1@$host "ssh-keygen -t rsa; cp__
      ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys"
  fi
done

for host in $(cut -f1 $3); do
  if [ $2 != $host ]; then
    cat ~/.ssh/id_rsa.pub | ssh $1@$host__
      "/bin/cat >> ~/.ssh/authorized_keys"
  fi
done
else
  echo 'Invalid node list file name given'
  exit 4
fi
```

Figure A.1: The beorsakey script.

```
clear
echo '/-----------------------------------\'
echo '|                                   |'
echo '| beouseradd                        |'
echo '|  - utility to add the same user   |'
echo '|    across the Beowulf cluster on  |'
echo '|    each node, with the same       |'
echo '|    settings configured all.       |'
echo '\-----------------------------------/'
echo

if [ "$1" = "" ]; then
  echo No parameters supplied!
  echo Usage: beouseradd user uid password__
    group gid node_list
  exit 1
fi

for host in $(cut -f1 $6); do
  echo "Adding user $1 in group $4 to node $host."
  ssh $host "groupadd -g $5 $4;__
    useradd $1 -u $2 -g $4 -d /udd1/$1"
  ssh $host "echo "$3" | passwd $1 --stdin"
done
```

Figure A.2: The beouseradd script.

```
# sudoers file.
#
# This file MUST be edited with the 'visudo'__
  command as root.
#
# See the sudoers man page for the details__
  on how to write a sudoers file.
#

# Host alias specification
Host_Alias BEOHOSTS = grendel1, grendel2,__
  onegin, beo1, beo2, beo3

# User alias specification
User_Alias BEOUSERS = beouser

# Cmnd alias specification
Cmnd_Alias NICE = /bin/nice
Cmnd_Alias RENICE = /usr/bin/renice
Cmnd_Alias SUDO = /usr/bin/sudo

# Defaults specification

# User privilege specification
root ALL=(ALL) ALL
BEOUSERS BEOHOSTS = NOPASSWD: NICE, RENICE, SUDO

# Uncomment to allow people in group wheel__
  to run all commands
# %wheel ALL=(ALL) ALL

# Same thing without a password
# %wheel ALL=(ALL) NOPASSWD: ALL

# Samples
# %users  ALL=/sbin/mount /cdrom,/sbin/umount /cdrom
# %users  localhost=/sbin/shutdown -h now
```

Figure A.3: The sudoers file.

```
#!/bin/sh
clear
echo '/------------------------------------------\'
echo '|                                          |'
echo '| beofilerep -                             |'
echo '|   - utility to add replicate file(s)     |'
echo '|     across the Beowulf to each node      |'
echo '\------------------------------------------/'
echo

if [ "$1" = "" ]; then
  echo 'No local file(s) given!'
  echo Usage: beofilerep files remote_dir node_list
  exit 1
elif [ "$2" = "" ]; then
  echo No remote directory given!
  echo Usage: beofilerep files remote_dir node_list
  exit 2
elif [ "$3" = "" ]; then
  echo No node list given!
  echo Usage: beofilerep files remote_dir node_list
  exit 3
fi

username=`whoami`
for host in $(cut -f1 $3); do
  scp $1 $username@$host:$2
done
```

Figure A.4: The beofilerep script.

```
#!/bin/sh
clear
echo '/-----------------------------------\'
echo '| beoprioritize                     |'
echo '|   - utility to lower or increase  |'
echo '|     the running process priorities |'
echo '|     all users except authorized   |'
echo '|     Beowulf users                 |'
echo '\-----------------------------------/'

if [ "$1" = "" ] || [ "$1" != "-u" ] &&__
  [ "$1" != "-d" ]; then
  echo 'Usage: beoprioritize -u | -d [value]'
  echo '       Where -u assigns a lower priority'
  echo '       Where -d assigns a higher priority'
  echo '       Value is optional. If not specified,'
  echo '       -u lowers to 19, -d increases to 0.'
  echo '       Process priority may range from'
  echo '       -20 (highest) to 19 (lowest).'
  exit 1
fi

users > users.txt; awk '
BEGIN {
  NF=" "
}
{
for (i=NF;i>=1;i--)
  print $i
}' users.txt > users2.txt

if [ "$2" != "" ] && [ "$2" -ge -20 ] &&__
  [ "$2" -le 19 ]; then
  priority=$2
else
  if [ "$1" = "-u" ]; then
    priority=19
  else
    priority=0
  fi
fi

for user in $(cut -f1 users2.txt)
do
  case $user in
    "beouser")
      echo Skipping user beouser
      ;;
    "root")
      echo Skipping user root
      ;;
    *)
      if [ "$1" = -u ]; then
        echo Lowering process priorities for $user
        sudo renice $priority -u $user
      elif [ "$1" = -d ]; then
        echo Increasing process priorities for $user
        sudo renice $priority -u $user
      fi
      ;;
  esac
done

rm -f users.txt; rm -f users2.txt
```

Figure A.5: The beoprioritize script.

```
#!/bin/sh
clear
echo '/--------------------------------\'
echo '| cronsched -                    |'
echo '|   - utility to schedule exec   |'
echo '|     of prioritize script every |'
echo '|     5 minutes. The file with   |'
echo '|     the actual crontab entry   |'
echo '|     is also specified          |'
echo '\--------------------------------/'
echo
if [ "$1" != "-a" ] && [ "$1" != "-d" ]; then
  echo No -a or -d command given!
  echo 'Usage: cronsched -a|-d [crontabfile]'
  echo '       Where -a adds entries,
  echo '       -d removes entries in crontabfile'
  echo '       optional crontabfiles specifies '
  echo '       filename which contains the entries'
  echo '       If ommited, file crontabfile used'
  exit 1
fi

if [ "$1" = "-a" ] && [ "$2" != "" ] && [ -f "$2" ]
then
  filename=$2
  echo using filename $filename
elif [ "$1" = "-a" ] && [ -f crontabfile ]; then
  filename="crontabfile"
  echo Using filename $filename
elif [ "$1" = "-a" ] && [ ! -f crontabfile ]; then
  echo 'No crontabfile exists, no filename supplied!'
  echo 'Priority script not scheduled'
  exit 2
fi

if [ "$1" = "-a" ]; then
  crontab $filename
  echo Successfully added user crontab entries.
elif [ "$1" = "-d" ]; then
  crontab -r
  echo Successfully removed user crontab entries.
fi
```

Figure A.6: The beocronsched script.

```
#!/bin/sh
clear
echo '/--------------------------------\'
echo '| beorun -                        |'
echo '|   - utility to start exec of    |'
echo '|     multiple simulation runs    |'
echo '|     across the Beowulf cluster |'
echo '\--------------------------------/'
echo

if [ "$1" = "" ]; then
  echo No project name given!
  echo Usage: beorun simrun_name cmd_list node_list
  echo '       Where project_name is name of sim run'
  echo '       Where node_list is file list of nodes'
  echo '       Where cmd_list is file list of commands'
  exit 1
elif [ "$2" = "" ]; then
  echo No command list given!
  echo Usage: beorun node_list cmd_list
  echo '       Where project_name is name of sim run'
  echo '       Where node_list is the file list of nodes'
  echo '       Where cmd_list is file list of commands'
  exit 2
elif [ "$3" = "" ]; then
  echo No node list given!
  echo Usage: beorun node_list cmd_list
  echo '       Where project_name is name of sim run'
  echo '       Where node_list is the file list of nodes'
  echo '       Where cmd_list is file list of commands'
  exit 3
fi

beoprioritize -u
beocronsched -a
date > /mnt/BeoResult/$1.status
/mnt/BeoStore/prun $3 $2
beocronsched -d
beoprioritize -d
date >> /mnt/BeoResult/$1.status
```

Figure A.7: The beorun script.

List of References

[1] World-Information.org: In search of reliable internet measurement data, 2009. Available at http://world-information.org/wio/infostructure/100437611791/100438658352.

[2] Consortium, I.S.: Iscdomain name survey, August 2009. Available at https://www.isc.org/solutions/survey.

[3] Spring, N., Mahajan, R., and Whetherall, D.: Measuring isp topologies with rocketfuel. In: *SIGCOMM 2002*. ACM, 2002.

[4] University, B.: Boston university representative internet topology generator, 2008. Available at http://www.cs.bu.edu/brite/.

[5] Taskforce, T.I.E.: Request for comments, 2009. Available at http://www.ietf.org/rfc.html.

[6] Vasseur, J.-P., Pickavet, M. and Demeester, P.: *Network Recovery: Protection and restoration of optical, SONET-SDH, IP, and MPLS.* Morgan Kaufmann Publishers, 2004. ISBN 0-12-715051-x.

[7] Hendrick, C.: Routing information protocol - rfc 1058, 1988. Available at http://www.ietf.org/rfc/rfc1058.txt.

[8] Kozierok, C.: The tcp/ip guide v3.0, 2005. Available at http://www.tcpipguide.com/free/t_RIPProtocolLimitationsandProblems.htm.

[9] Dijkstra, E.: Ewd-1166, 1993. Available at http://www.cs.utexas.edu/users/EWD/ewd11xx/EWD1166.pdf.

[10] Bhandari, R.: *Survivable Networks: Algorithms for Diverse Routing.* Kluwer Academic Publishers, 1999. ISBN 0-7923-8381-8.

[11] Iversen, V.: Handbook: Teletraffic engineering, May 2008. Available at http://oldwww.com.dtu.dk/teletraffic/handbook/telenookpdf.pdf.

[12] Zukerman, M.: Introduction to queueing theory and stochastic teletraffic models, 2008. Available at http://www.ee.cityu.edu.hk/z̃ukerman/classnotes.pdf.

[13] Rosen, E., Viswanathan, A. and Callon, R.: Multiprotocol label switching architecture, January 2001. Available at http://www.ietf.org/rfc/rfc3031.txt?number=3031.

[14] Braden, R., Zhang, L., Berson, S., Herzog, S. and Jamin, S.: Resource reservation protocol – version 1 functional specification, September 1997. Available at http://www.ietf.org/rfc/rfc2205.txt?number=2205.

[15] Awduche, D., Berger, L., Gan, D., Li, T., Srinivasan, V. and Swallow, G.: Rsvp-te: Extensions to rsvp for lsp tunnels, December 2001. Available at http://www.ietf.org/rfc/rfc3209.txt?number=3209.

[16] Heckmann, O., Piringer, M., Schmitt, J. and Steinmetz, R.: On realistic network topologies for simulation. In: *SIGCOMM 2003*. ACM, 2003.

[17] MCI: North america network topology map, 2004. Accessed at http://global.mci.com/about/network/global_presence/global.

[18] Barabasi, A. and Albert, R.: Emergence of scaling in random networks. pp. 509–512. October 1999.

[19] Zipf, G.: *Human Behavior and the principle of least effort*. Addison-Wesley, 1949.

[20] Liu, Y., Tipper, D. and Siripongwutikorn, P.: In: *Approximating Optimal Spare Capacity Allocation by Successive Survivable Routing*. February 2005.

[21] Medina, A., Matta, I. and Byers, J.: On the origin of power laws in internet topologies. April 2000. Available at http://www.cs.bu.edu/brite/publications/ccr00.pdf.

[22] Rayport, J.: Semiconductor industry and business survey, 1988.

[23] Spurgeon, C.: Ethernet (ieee 802.3) web site, 2004. Available at http://www.ethermanage.com/ethernet/.

[24] Online, L.: Linus torvalds biography, 2007. Available at http://www.linux.org/info/linus.html.

[25] van der Veen, V.: The minix3 operating system, 2008. Available at http://www.minix3.org/.

[26] Stallman, R.: Richard stallman's personal page, 2008. Available at http://www.stallman.org/#serious.

[27] Barr, T.: Linux nfs-howto, 2002. Available at http://nfs.sourceforge.net/nfs-howto/.

[28] Sterling, T., Salmon, J. and Becker, D.: *How to Build a Beowulf*. MIT Press, 1999.

[29] Siever, E., Spainhour, S. and Patwardhan, N.: *Perl in a Nutshell*. O'Reilly, 1998.

[30] Gropp, W., Lusk, E. and Skjellum, A.: *Using MPI: portable parallel programming with the message-passing interface*. MIT Press, 1994. ISBN 0-262-57104-8.

[31] Bovet, D. and Cesati, M.: *Understanding the Linux Kernel.* O'Reilly, 2000.

[32] Morgan, A.: A linux-pam page, 2004. Available at http://www.kernel.org/pub/linux/libs/pam/.

[33] Peek, J., O'Reily, T. and Loukides, M.: *Unix Power Tools 2nd Edition.* O'Reilly, 1997.

[34] Stapelberg, D.: Building a linux beowulf cluster, 2004. Accessed at http://www.cs.sun.ac.za/ dstapel/beowulf/.

[35] Göbel, J., Krzesinski, A. and Stapelberg, D.: A distributed scheme for responsive network engineering. pp. 2070–2075. June 2007.

[36] Fall, K. and Varadhan, K.: The ns manual, 2009. Available at http://nsnam.isi.edu/nsnam/index.php/Main_Page.

[37] Varga, A.: The omnet user manual, 2009. Available at http://www.omnetpp.org/doc/omnetpp40/manual/usman.html.