# QoS Routing and Pricing in Large Scale Internetworks

Diploma Thesis of David Schweikert

November 1998 – March 1999

Computer Engineering and Networks Laboratory, ETH Zürich
Supervisors: George Fankhauser and Burkhard Stiller
Professor: Bernhard Plattner

**Abstract**

The Internet has grown from a research network to a world-wide network which is used daily by millions of people. The increasing popularity is accompanied by increasing demands on consumer-oriented services such as telephony and video streams, known from circuit-switched networks. This type of application require a guaranteed *Quality of Service*, because of their real-time nature.

To satisfy these new consumer demands, the Internet must be adapted, since it was originally designed as a data packet network with best-effort packet forwarding capabilities. At that time the focus was on connectivity. If no special provisions are taken, no guarantees can be made on the duration that a packet will take to travel to it's destination, and thus no Quality of Service (QoS) can be assured.

In the future, Quality of Service will probably be assured in the Internet by using the *diffserv* framework proposed by David Clark and Van Jacobson. In the *diffserv* framework, packets are classified for QoS services only at the border of the providers according to fixed classes. *Aggregates* of flows are thus formed and the overhead is much lower than other QoS enforcing systems such as *intserv* where flows are treated independently by every router. Unfortunately no specific QoS routing system is provided in the *diffserv* framework and it is expected to use normal routing algorithms designed for *best-effort* traffic.

Between providers using diffserv, bilateral contracts on QoS services are made. These contracts, called *Service Level Agreements* (SLA), define how much and under what conditions traffic will be processed by a provider and which guarantees can be made on it's delivery.

A routing system for the diffserv framework is presented and analysed in this diploma thesis. The system automates the buying and selling process of Service Level Agreements between providers with *SLA Traders* which communicate using a *SLA Trading Protocol*.

*SLA Traders* are algorithms which analyse the current market, forecast future demands and buy resources accordingly in form of SLAs from other providers or by other external means such as buying additional links. It is also the SLA Trader's responsibility to construct from the available resources *SLA Bids* for other traders. *SLA Bids* are service proposals made by *SLA Traders*. A customer can accept such a bid and sign the contract with the service provider, for which it will be charged some money.

Competition between providers to sell services to a customer should force them to minimise the price of their SLA Bids thus increasing their attractiveness and probability of being accepted. The providers will try to optimise the utility/price ratio of their services by also minimising the usage of resources.

A provider who chooses to buy a service from a set of proposals does in fact make a *routing decision*, because it will then forward the packets for that service to the provider which sold him that service. For this reason we call the routing *SLA Trading-based Routing*.

This system was simulated on a new simulator written from scratch for this project. The simulator, called *Flowsim*, simulates packets on a high level as *flows* thus simplifying very much the simulation and making it much faster than packet simulators such as *ns-2*. It is written in Java and the classes specific to the SLA Trading System were put in a separate package to make it useful

also outside of this work. It is small and uses the object-orientation features of the Java language to make it easily understandable and extensible.

The system is shown to be *protocol independent*, because of the purely bilateral nature of the contracts. It is thus easily deployable by providers since no global protocol agreement is needed. It is a system which does *minimises network load* because more resources usage does represent more cost for the providers who will try to economise as much as they can. The providers will also try to provide the best possible service to customers to increase their gains in the selling of services, thus *maximising user utility*.

Future possible work on the SLA Trading System includes a better, more compact, SLA Trading Protocol, which would mean lower messaging overhead and more precise SLA Bids. Other possible future work is the implementation of *Assured Service* forwarding guarantees and the usage of a *diffserv* compatible DS-byte.

**Zusammenfassung**

Das Internet ist von einem Forschungsnetz zu einem weltweiten Netz gewachsen, das täglich von Millionen von Menschen benutzt wird. Diese zunehmende Popularität ist von der Erhöhung der Nachfrage nach Multimedia-Applikationen wie Fernsprechen und Videoconferencing begleitet. Sie sind bekannt von den leitungsvermittelten Netzen. Diese Art der Anwendungen benötigen aufgrund ihre Echtzeitnatur eine *garantierte Dienstgüte* (Quality of Service, QoS).

Um diese neuen Nachfragen zu erfüllen, muss das Internet angepasst werden, da es ursprünglich als Datenpaketnetz mit *best-effort* Dienstleistungseigenschaften entworfen wurde. Ein erstes Ziel des Internets bestand im Erreichen der Konnektivität. Wenn keine speziellen Voraussetzungen in bezug auf Verzögerungszeiten angenommen werden, können keine Dienstgarantien abgegeben werden.

Zukünftig werden vermutlich Dienstgüten im Internet möglich sein, indem man den *diffserv* Ansatz benutzt, der von David Clark und Van Jacobson vorgeschlagen wurde. In der *diffserv* Architektur werden Pakete, die spezielle QoS-Anforderungen aufweisen, nur am Rand der Anbietersubnetze durch entsprechend festgelegte Kategorien klassifiziert. Aggregate von Datenströmen werden folglich gebildet und die Komplexität ist viel niedriger als in anderen QoS-Systemen. Zum Beispiel im *intserv* wird jeder Datenstrom unabhängig voneinander behandelt. Leider wird kein spezifischer QoS-Routing-Algorithmus im *diffserv* zur Verfügung gestellt, und es wird erwartet, das normale Routing zu verwenden, welches für *best-effort* Verkehr entworfen wurde.

Zwischen Anbietern, die diffserv benutzen, werden bilaterale Verträge für QoS-Dienstleistungen gebildet. Diese Verträge, die in *diffserv Service Level Agreements* (SLA) genannt werden, definieren, wieviele und unter welche Bedingungen Pakete durch einen Anbieter verarbeitet werden sollen und welchen Garantien für Dienstleistungen gemacht werden können.

Ein Routing-System für *diffserv* wurde in dieser Diplomarbeit enwickelt und studiert. Das System automatisiert den Kauf und Verkauf der Verträge (SLA) zwischen den Anbietern mit *SLA-Traders*, die mit einem *SLA Trading Protocol* in Verbindung stehen.

SLA Traders sind Algorithmen, die den aktuellen Markt analysieren, zukünftige Nachfragen schätzen und die Ressourcen dementsprechend in Form von SLAs von anderen Anbietern oder mit anderen externen Mitteln, wie das Bereitstellen von zusätzlichen Links, vergrössern. Es liegt auch in der Verantwortung der SLA Traders, aus den vorhandenen Ressourcen SLA Angebote für andere Anbieter herzustellen. SLA-Angebote sind Dienstleistungsvorschläge, die von den SLA Traders gebildet werden. Ein Kunde kann ein Angebot annehmen und den Vertrag mit dem Anbieter unterzeichnen. Die im Vertrag verinbarte Dienstleistung wird finanziell verrechnet.

Konkurrenz zwischen Anbietern, Dienstleistungen an einen Kunden zu verkaufen, sollte diese zwingen, den Preis ihrer SLA-Angebote zu senken, um die Attraktivität und folglich die Wahrscheinlichkeit angenommen werden zu können zu erhöhen. Die Anbieter versuchen, das Kosten-Nutzen (utility-price) Verhältnis ihrer Dienstleistungen zu vergrössern, also müssen sie den Verbrauch der Betriebsmittel optimieren.

Ein Anbieter, der beschliesst, einen SLA von eine Gruppe von SLA-Angeboten zu kaufen, trifft tatsächlich eine Routing-Entscheidung, weil er dann die Pakete dieses Dienstes demjenigen Anbieter sendet, der ihm diesen Service verkaufte. Darum nennen wir dieses System *SLA Trading-based Routing*.

Dieses System wurde auf einem Simulator implementiert, der neu fuer diese Arbeit geschrieben wurde. Der Simulator, genannt *Flowsim*, simuliert Datenströme, welche eine höhere Abstraktionsstufe als Pakete darstellen. Dieses vereinfacht die Simulation und verschnellert sie im Vergleich zu Paketsimulatoren wie ns-2. Der Simulator wurde in Java geschrieben und die Teile, die spezifisch zum SLA Trading System gehören, wurden in einem separatem Package implementiert. Diese Modularität erlaubt es auch, den Simulator ausserhalb dieser Arbeit einzusetzen. Er ist klein und benutzt die objektorientierten Eigenschaften der Java-Sprache, um sie verständlich und leicht erweiterbar zu machen.

Es wird innerhalb dieser Diplomarbeit gezeigt, dass das SLA Trading System wegen der bilateralen Natur der Verträge protokollunabhängig ist. Es ist folglich leicht für Anbieter einsetzbar, da keine globale Protokollvereinbarung erforderlich ist. Es ist ein System, das die Netzwerklast minimiert, weil mehr Betriebsmittelverbrauch Mehrkosten für die Anbieter darstellen. Die Anbieter versuchen auch, den bestmöglichen Service den Kunden zur Verfügung zu stellen, um ihre Gewinne durch das Verkaufen von Dienstleistungen zu erhöhen.

Zukünftig mögliche Arbeiten für das SLA Trading System schliessen ein kompakteres SLA Trading Protokoll ein, welches niedrigere Netwerklast und exaktere SLA-Angebote bedeuten würde. Andere weiterführende Arbeiten betreffen die Implementierung der *Assured Capacity* Dienstleistung, wo statistische Garantien gegeben werden, und den Gebrauch eines diffserv-kompatiblen DS-Byte.

# Contents

**1 Introduction** **11**

1.1 Motivation . . . . . 11

1.2 Project Goals . . . . . 11

1.3 Structure . . . . . 12

1.4 Flowsim . . . . . 13

1.5 Acknowledgements . . . . . 13

**2 Related work** **15**

2.1 Quality of Service . . . . . 15

2.2 QoS Systems . . . . . 15

2.3 Routing Algorithms . . . . . 17

2.4 Quality of Service and Routing . . . . . 19

2.5 QoS Provisioning and Packet Classification . . . . . 20

2.6 Routing with Multiple Parameters . . . . . 20

2.7 Routing with Multiple Parameters . . . . . 21

**3 Experiments** **23**

3.1 Introduction . . . . . 23

3.2 Selective Probing based on Routing Hints . . . . . 23

3.3 Selective Probing with Probes Recollection . . . . . 25

3.4 Class Based Distance-Vector Routing (CBDV) . . . . . 25

3.5 SLA Trading Based Routing (SLATR) . . . . . 29

3.6 Preliminary conclusions . . . . . 29

**4 SLA Trading Concepts** **31**

4.1 Introduction . . . . . 31

4.2 Service Level Agreements . . . . . 32

4.3 SLA Trading . . . . . 33

4.4 SLA Traders . . . . . 33
4.5 SLA Bids Generation . . . . . 34
4.6 Routing based on SLA Trading . . . . . 34
4.7 SLA Trading based Routing Example . . . . . 35
4.8 Traders Competition . . . . . 37
4.9 User Competition . . . . . 37
4.10 Bid Guarantees and Bids Flooding . . . . . 37
4.11 Trend Analysis vs. Explicit Asks . . . . . 38
4.12 Profit Optimisation . . . . . 38
4.13 Service Loops . . . . . 38
4.14 Messaging Overhead . . . . . 39
4.15 Trading Styles . . . . . 40
4.16 Summary of Design Issues . . . . . 40

**5 The SLA Trading Protocol 43**
5.1 Introduction . . . . . 43
5.2 SLA Specification . . . . . 43
5.3 SLA Identification . . . . . 44
5.4 Messages . . . . . 44
5.5 Possible Improvements . . . . . 47

**6 Users Preferences 49**
6.1 Introduction . . . . . 49
6.2 Utility . . . . . 49
6.3 Willingness To Pay . . . . . 51
6.4 Bids Evaluation . . . . . 52

**7 Implementation 53**
7.1 Introduction . . . . . 53
7.2 SLA Trading Protocol Messages . . . . . 53
7.3 SLA . . . . . 54
7.4 SLABuyer . . . . . 54
7.5 SLASeller . . . . . 55
7.6 SLA Map . . . . . 55
7.7 SLATRouter Base Class . . . . . 57
7.8 SLATRouterImpl . . . . . 59

7.9 SLATRouter Implementations . . . . . 60
7.10 Greedy Trader . . . . . 60
7.11 Trendy Trader . . . . . 61
7.12 Profitable Trader . . . . . 61
7.13 Lazy Trader . . . . . 61
7.14 Users . . . . . 62
7.15 Voice User . . . . . 63

**8 Results 65**

8.1 Introduction . . . . . 65
8.2 Simulation limitations . . . . . 65
8.3 Load balancing . . . . . 66
8.4 User Competition . . . . . 69
8.5 Trader Implementations Comparison . . . . . 71

**9 Conclusions 75**

9.1 Advantages . . . . . 75
9.2 Problems . . . . . 76
9.3 Review of Goals . . . . . 76
9.4 Future Work . . . . . 77

**A Project Description 79**

A.1 Introduction . . . . . 79
A.2 Environment . . . . . 80
A.3 Tasks . . . . . 82
A.4 Remarks . . . . . 85
A.5 Results . . . . . 85
A.6 Bibliography . . . . . 86

**B `flowsim`: a network flows simulator 87**

B.1 Introduction . . . . . 87
B.2 Flows vs. Packets . . . . . 88
B.3 Simulator . . . . . 89
B.4 Scheduler . . . . . 89
B.5 Node . . . . . 90
B.6 Monitors . . . . . 91

B.7 Link . . . . . . . . . . 91
B.8 Flow . . . . . . . . . . 92
B.9 LinkManager . . . . . . . . . . 95
B.10 Router . . . . . . . . . . 96

**C Distance-Vector Routing in Flowsim 97**
C.1 Introduction . . . . . . . . . . 97
C.2 DV Update Messages . . . . . . . . . . 97
C.3 DV Table . . . . . . . . . . 98
C.4 Router Implementation . . . . . . . . . . 98
C.5 Class-Based DV Routing . . . . . . . . . . 99

**Bibliography 102**

# Chapter 1

# Introduction

## 1.1 Motivation

The Internet has grown from a research network to a world-wide network which is used daily by millions of people. The increasing popularity is accompanied by increasing demands on consumer-oriented services such as telephony and video streams, known from circuit-switched networks. This type of application require a guaranteed *Quality of Service*, because of their real-time nature.

To satisfy these new consumer demands, the Internet must be adapted, since it was originally designed as a data packet network with best-effort packet forwarding capabilities. At that time the focus was on connectivity. If no special provisions are taken, no guarantees can be made on the duration that a packet will take to travel to it's destination, and thus no Quality of Service (QoS) can be assured.

In the future, Quality of Service will probably be assured in the Internet by using the *diffserv* framework proposed by David Clark and Van Jacobson. In the *diffserv* framework, packets are classified for QoS services only at the border of the providers according to fixed classes. *Aggregates* of flows are thus formed and the overhead is much lower than other QoS enforcing systems such as *intserv* where flows are treated independently by every router. Unfortunately no specific QoS routing system is provided in the *diffserv* framework and it is expected to use normal routing algorithms designed for *best-effort* traffic.

This work should investigate the existing solutions to QoS routing problem, i.e. the problem of finding a route to a destination with additional QoS constraints. A new *routing and pricing system* that makes the QoS-guaranteed delivery of packets possible and efficient for inter-provider networks should be designed and analysed.

## 1.2 Project Goals

Goal is to develop and analyse a routing and pricing system with the following main properties:

- Loop-free, convergent routing
- Admission control based on available resources *and* pricing
- Services differentiation

- Quality of Service assurances
- User utility maximisation
- Network load minimisation
- Easy deployment in existing Internet

The full initial project description is reported in *appendix A*.

## 1.3 Structure

A description of the project phases is now reported and should serve as an overview of how the work was organised. This report follows in it's structure also these phases and this section can thus also be considered a document overview.

The first part of the project was devoted to literature studying and analysis. The goal of that literature work was having an overview of the related work made in QoS provisioning, routing and pricing. To facilitate the analysis work of the various algorithms a dichotomy of routing and QoS provisioning methods was made.

The result of this overview work is presented in *chapter 2*.

The existing problems of QoS provisioning, QoS routing and their relation to pricing are known and is what should be addressed in this work. What was not known before to start this work, was how to solve this problem.

To investigate different solutions and to avoid working only on one idea which then reveals to be not interesting, impracticable or inefficient, the first goal was to define four possible approaches and experiment with them. The most promising approach would then be chosen and form the basis for the rest of the diploma thesis.

After many brainstorming sessions the experiments were defined:

- Selective Probing based on Routing Hints
- Selective Probing with Probes Recollection
- Class based Distance-Vector Routing
- SLA Trading based Routing Protocol

See *chapter 3* for the description of the experiments, their results and the conclusions on which one was retained to be the most interesting to continue the work on.

The most promising proposal which did emerge from the experimentation phase was the *SLA Trading based Routing Protocol* experiment and it was taken as a basis for the rest of the diploma thesis. The experiment was studied and experimented more in detail in the design phase. A protocol was designed which does implement the investigated algorithm. See *chapter 5* for an explanation of the algorithm and protocol.

The final goal of all these QoS provisioning and routing algorithms is the maximisation of *user satisfaction* and *minimisation of costs* for both user and provider. To evaluate the user satisfaction

of the services provided by the network, a model of user preferences and behaviour was made. It is explained in detail in *chapter 6*.

The *SLA Trading based Routing Protocol* was implemented in a simulator so that the problems of the algorithm could be investigated and it's efficiency demonstrated. See *chapter 7* for an explanation of the protocol implementation in the simulator.

Using the simulator implementation, scenarios were simulated, emphasising special situation favourable or not to specific properties of the algorithm. The results of these simulations, presented in *chapter 8*, should show the advantages and problems of this algorithm.

Using the knowledge gained during the design and measurement of the algorithm, final conclusions on the efficiency are made in this last phase. Possible practical applications are proposed along with a feasibility analysis. See *chapter 9*.

## 1.4 Flowsim

As explained in chapter 7, no satisfying simulation environments were found for this type of work, and thus a new simulator was written from scratch. The time used to write that simulator was thought to be less than the time which would have needed an implementation of the SLAT Protocol (see chapter 5) in existing simulators such as *ns2*.

The resulting simulator, called `flowsim`, is a network simulator which is small, easily understandable and fast. It is written in java and uses the object-oriented features of the java languages to make the implementation as straightforward as possible. I think that `flowsim` could also be used outside of this project, because it was tried to keep it as generic as possible, by keeping the parts specific to this project in separate packages.

The documentation of `flowsim` is reported in *appendix B*.

## 1.5 Acknowledgements

This project was both interesting and entertaining and I would like to thank the TIK institute for giving me the possibility of doing it. I would like to especially thank George Fankhauser, the project supervisor, for the huge help and the many hours of discussions about the problems discussed in this report. Thanks to Burkhard Stiller, also project supervisor, for his precious hints.

# Chapter 2

# Related work

## 2.1 Quality of Service

A *Quality of Service* (QoS) specification for a network service, such as the transport of packets to a destination, does describe the guaranteed properties which will be kept by the network when delivering the packets to that destination and which the service user can pretend to be respected. See [Sti96] for further discussion of Quality of Service specification.

The Internet, as designed 20 years ago focused on connectivity and did only define one very simple QoS delivery guarantee: *best-effort*, i.e. every router in the network makes the best it can to deliver each packet the fastest possible. No differentiation on the application types is made: every packet is treated equally. It did make sense at that time, because of simplicity and efficiency. It does also make sense today, but only for those applications which don't pose constraints on the delivery delay of packets.

The QoS requirements of the network applications vary very much. For the File Transfer Protocol, for example, it is important to transmit the data fast, but it doesn't matter if the throughput isn't constant or if there is a big delay. On the contrary, for a telephony application, the data rate is low, but there must be a low delay and a very low delay-variation. It therefore makes sense to rather offer different QoS guarantees to different types of applications.

The telephony application is said to need *Quality of Service* assurances from the network, such as maximal packet transmission delay.

## 2.2 QoS Systems

The different QoS services can be implemented by giving different packets different priorities in the router queues and/or by choosing appropriate routes.

In order to make it possible to provide different QoS services, it must be possible to identify the type of service that a packet belongs to. In other words, it must be possible to say for example "this packet belongs to a data stream (or "flow"), which requires the lowest possible delay jitter.

### 2.2.1 Type of Service

The designers of the Internet Protocol [Pos81] already recognised the problem that packets of applications requiring different QoS delivery required a way to be differentiated and did reserve a byte in the IP header to specify the *Type of Service*. The TOS field was so described in the RFC791 (dated 1981):

"The Type of Service is used to indicate the quality of the service desired. The type of service is an abstract or generalised set of parameters which characterise the service choices provided in the networks that make up the Internet. This type of service indication is to be used by gateways to select the actual transmission parameters for a particular network, the network to be used for the next hop, or the next gateway when routing an Internet datagram."

The TOS field, which is 8 bits wide, does contain (in order from MSB to LSB) 3 bits to indicate precedence, 1 bit to indicate normal or low delay, 1 bit to indicate normal or high throughput, 1 bit to indicate normal or high reliability and 2 bits are reserved for future use. A file transfer packet would thus ideally have a TOS `xxx010xx` and a telnet session would ideally generate packets with TOS `xxx101xx`.

Unfortunately, this field in the IP header is today still ignored by most TCP/IP implementations.

### 2.2.2 Integrated Services

Another solution to the QoS problem is using a differentiation of packets on a *flow* level, such that packets in the same flow are treated equally. A *flow* is a stream of packets generated by an application, such as for example the packets generated by a Internet-telephony call.

In an Integrated Services (*intserv*) system, the network is re-configured to serve a particular flow, with it's QoS guarantees, each time a new flow is started. It is a connection-oriented system where the QoS provisioning is made on connection setup and is very similar in concept to circuit-switched systems.

A typical Internet telephony call in a intserv system would be made as follows:

1. User A asks the network to make a call to user B.
2. The network allocates the resources for a channel from A to B and acknowledges the request to A.
3. When the communication is finished, user A or user B request the network to shut down the connection.
4. The network frees the allocated resources.

The most used intserv signalling protocol in the Internet is the Resource reSerVation Protocol – *RSVP* (see [RZB$^{+}$97]). RSVP provides "receiver-initiated setup of resource reservations for multicast or unicast data flows, with good scaling and robustness properties" and is said to provide *soft-state* reservations, because, as opposed to for example ATM networks, they do need to be refreshed periodically. When these periodic refreshes are stopped, the resources are automatically released.

See [BCS94] for a generic discussion on the "Integrated Services Architecture".

### 2.2.3 Differentiated Services

The overhead associated with Integrated Services networks is huge for large networks. Imagine an Internet-telephony call from Zürich to New York using a resource reservation protocol: every router between Zürich and New York has to reserve explicitely for that call some bandwidth. The overhead posed on backbone routers is enormous and impracticable. Alternative solutions are therefore investigated and the today most promising technology is that of the *Differentiated Services* (*diffserv*) as first explained by D. Clark and V. Jacobson in two separate talks at the 39. IETF Meeting of 1997 in Munich. A differentiated architecture paper by Clark, published afterwards, is [CF98].

In Differentiated Services system, the traffic entering a network is classified and possibly conditioned at the boundaries of the network, and assigned to different QoS behaviours. The type of QoS behaviour is encoded in the *DS codepoint*, which normally is written the TOS byte field of the IP header. This QoS behaviour of a packet traversing a node is called *per-hop* behaviour (PHB) and is defined as a "description of the externally observable forwarding behaviour of a DS node" in [BBC+99].

The main idea of guarantees in the Differentiated Services approach is that of profiles and the marking of packets as *in profile* or *out of profile*. A profile is a description of the traffic which is accepted by a provider from an external source. It could for example say that this external source can transmit up-to 10 megabytes of delay-sensitive traffic and 50 megabytes of bandwidth-sensitive traffic. If the external source does exceed this specification, the provider will mark the packets thereafter as *out of profile*. This sort of *admission control* is made at the boundaries of the provider network.

In case of shortage of resources, the *in profile* packets will be treated with priority, because these are the packets which respect the profile defined in the contracts between the external entities (be it another provider or a user) and the provider.

The provider should provision it's network so as to quality of service delivered for *in profile* packets in the margins defined by the various profiles. This provisioning can be made by buying profiles from other providers, etc.

## 2.3 Routing Algorithms

The routing of packets on a network is made in three main phases common to all routing algorithms (see figure 2.1): the collection of the information needed to establish a correct route across the network (*Routing Information Collection*), the algorithm, which from the collected data computes the route (*Routing Algorithm*) and the method how the packets are transported from source to destination (*Packet Routing*).

See [Hui95] for a very complete discussion of all the routing protocols used in the Internet.

### 2.3.1 Routing Information Collection

A decision on how to route a packet across the network clearly needs information on the network topology to do so. The alternative would be to broadcast every packet on the whole network, which is obviously impractical.

We call the process of getting information on the network topology to make the routing decisions

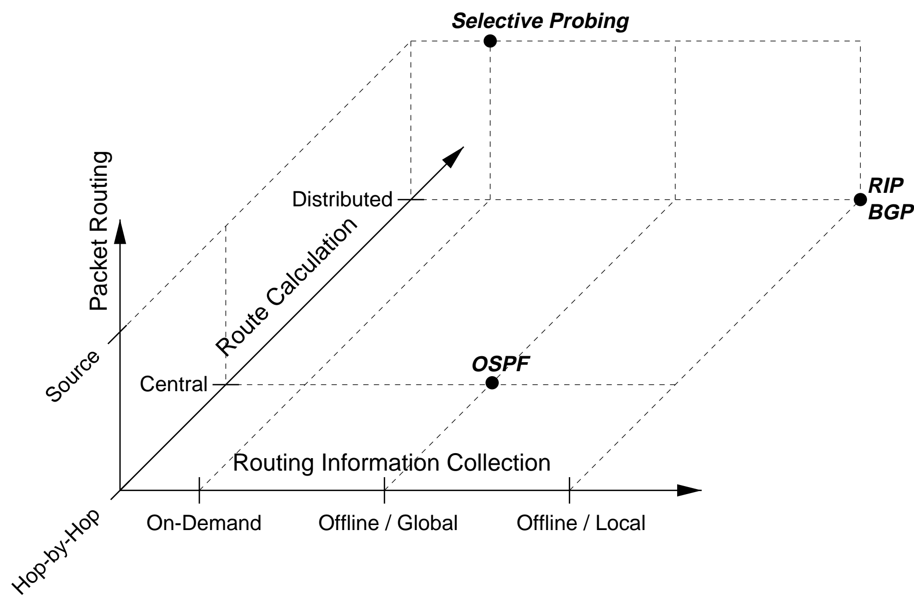



Figure 2.1: Routing method classification

*Routing information collection*. The methods how the information is collected can be grouped in three main categories:

- *On-Demand*: The information is collected whenever a routing decision should be made. When a packet with an unknown destination arrives on a router, the information collection process for that destination is started, which is normally done with a form of broadcasting. Example of is type is the *Selective Probing* algorithm, described in section 3.2.
- *Offline / Global*: A separate process, independent from the arrival of packets, collects information on the topology of the whole network. Every router in this group of routing methods has a global image of the network topology. Notable algorithm in this group is *Link-State* routing such as *OSPF*.
- *Offline / Local*: As in the 'Offline/Global' type of algorithms, the collection of information is done independently of the arrival of packets in a separate process. The image of the network topology in every router is however *local*. A router can only make a local decision on how the packets should be routed, i.e. on which link to forward the packet. The *Distance-Vector* algorithm for example collects local information.

### 2.3.2 Route Calculation Algorithm

We call the algorithm which, on basis of the collected information on topology, calculates the route that a packet should follow, the *Routing Algorithm*. These algorithms can also be classed in basic categories as follows:

- *Central*: The algorithm on a router doesn't need the partial results of the algorithm on other routers to produce a final result. Every router can make an independent decision on the routing of each packet from source to destination. The *Link-State* algorithm is calculated independently on each node.

- *Distributed*: The algorithm needs the proper functioning of the same algorithm on the other nodes and the computation of the route is made iteratively using partial results of each node. Such an algorithm is said to be a *distributed algorithm*. Prominent example using a distributed algorithm is the *Distance-Vector* algorithm.

### 2.3.3 Packet Routing and Policy

The routing systems can further be classified according to when the route of the packets is determined. There are two methods:

- *Source*: The complete route is calculated one time by the first router – the *Source*. Every router on the path from source to destination will follow the indications of the first router which can be inscribed by him in every packet for example. We put the *Selective Probing* in this category because the path is calculated by probes sent by from the first node and when the route is selected, it is fixed, and is not recomputed every time the packet traverses a router.
- *Hop-by-Hop*: The decision of which link to follow next when a packet arrives on a router is made anew each time. The *packet routing process* can be considered in this case a distributed algorithm.

The *hop-by-hop routing* is certainly less efficient because much more processing resources are needed, but is also much easier to implement than the *source routing*. Another advantage as opposed to source routing is the fact that it is considered more secure and it is possible for providers to establish *routing policies*.

Routing policies are used for example by non-commercial networks such as the Switch network to avoid the routing of packets originating from commercial entities and are a fundamental requirements of providers which want to make business in the Internet.

*Policy routing* is certainly more complicated in a source routing context. Note that policy routing is compatible with the *Selective Probing* algorithm because the path selection process already discards routes which aren't traversable, for example because of set-up policies.

## 2.4 Quality of Service and Routing

The Quality of Service problem has been traditionally solved by using various techniques along a fixed path chosen by normal shortest-path routing protocols. The techniques involved for example queueing algorithms on the routers which do take into account the various QoS classes of the incoming packets. Prominent example of these algorithms is the *Weighted Fair Queueing* (WFQ) algorithm (see [SW]).

Although ignored by most researchers in the field, it is legitimate to ask if the shortest path from a source to a destination for a flow with some QoS requirements is always the best path.

The best path for a bulk data transfer, where only the total transfer time is important and not the delay could be completely different of the best for a voice call flow where the delay is of primary importance. The former could take a many-hops route, which has however a large bandwidth and the latter is interested in the route with the shortest total delay, which would be probably the shortest path route.

## 2.5 QoS Provisioning and Packet Classification

A QoS networking system, defined as a system which does transport flows of data with Quality of Service guarantees, can be classified according to two main criteria: it's routing granularity and the type of Quality of Service enforcing technology (for the mostly part queueing technology). Figure 2.2 does present such a classification in a table.

Queuing / Quality of Service

| Routing Granularity | Best Effort | Differentiated Services: Prioritized Best Effort | Differentiated Services: Allocated Capacity | Integrated Services: Soft-State Reservation | Integrated Services: Hard-State Reservation |
|---|---|---|---|---|---|
| Destination | Traditional Internet | TOS Routing and Queuing | Traditional Alloc. Cap. | Traditional RSVP | Signaling System Nr. 7 |
| Destination + Class | | | | | ATM |
| Flow | | | | | |

Figure 2.2: QoS Provisioning and Packet Classification

By *routing granularity* the granularity of packet classification for the purpose of routing is understood. In other words, how are packets identified to choose the link they should follow next? This decision can be made only on the basis of the destination, in which case every packet with that destination will follow the same route, with destination and some sort of class or for each flow differently.

The arrows in the graph show the direction in which the existing technologies are pushed by the Quality of Service routing proposals.

The routing and pricing system proposed in this diploma thesis is based on the *diffserv* architecture ("Allocated Capacity" in the figure) and routing based on Destination and Class.

## 2.6 Routing with Multiple Parameters

The QoS routing problem can be resumed as "finding a route from source to destination, which respects the specifications of the needed QoS". It is mostly solved by using *routing algorithms with multiple parameters*. These algorithms choose, on the basis of multiple parameters (metrics) of the network, such as residual bandwidth, propagation delay, etc., and on the requested service, the best possible path. It is called *multiple parameters routing* because the algorithms are much more complicated than *single parameter routing* algorithms such as *shortest-path routing*.

### 2.6.1 Definition of Concave and Additive QoS Metrics

A *metric* is a characterisation parameter for a path or link in a network. A *QoS metric* is a metric which characterises the quality of the path.

Let $m(i,j)$ be a QoS metric for link $(i,j)$. For a path $P = (s,i,j,\dots,l,t)$, metric $m$ is

- *concave* if $m(P) = min\{m(s,i), m(i,j), \ldots, m(l,t)\}$
- *additive* if $m(P) = m(s,i) + m(i,j) + \cdots + m(l,t)$

(definition taken from [CN98a])

Bandwidth is for example a concave metric, because on a path, the link with the smallest bandwidth determines the total available bandwidth on the path. Propagation delay is additive because it is the result of the sum of all the propagation delays on each link.

### 2.6.2 Definition of link and path constraints

*Constraints* are used in QoS routing algorithms to define what properties the chosen path must have, i.e. to limit the search scope for the best route to the possible routes.

A *link constraint* specifies a restriction on the use of links and a *bandwidth constraint* specifies the end-to-end requirement on a single path. For example, a bandwidth constraint is a link constraint because it does limit the bandwidth of each link on the path. A delay constraint is however a path constraint because it does limit the end-to-end total delay.

A *link constraint* does limit a *concave metric* and a *path constraint* does limit a *additive metric*.

### 2.6.3 Definition of Optimisations

*Optimisations* are used in QoS routing algorithms to search, among the possible path, the best one according to some optimisation criteria.

Optimisation can be made with additive metrics or concave metrics. A optimisation on a additive metric is said to be a *path-optimisation* and an optimisation on a concave metric is said to be a *link-optimisation*.

For example, the problem of finding the path with the minimal delay which has a minimal bandwidth is said to be a *link-constrained path-optimisation* routing problem.

### 2.6.4 NP Completeness of Routing with Multiple Parameters

When multiple path-constraints and/or path-optimisations should be made, the problem becomes NP-complete, i.e. unsolvable in polynomial time. When doing QoS routing this is almost always the case, because of the many metrics involved (see [CN98b] for further discussion of the NP-completeness of these algorithms).

A path-constrained path-optimisation problem (PCPO) example is the delay-constrained least-cost routing problem. Another NP-complete problem example is a multi-path-constrained problem (MPC): propagation delay and delay-jitter constrained routing.

## 2.7 Routing with Multiple Parameters

Many algorithms were designed which try to solve the QoS routing problem in an efficient and good way. Shigang Chen and Klara Nahrstedt have written a very good summary of these algorithms in [CN98b].

The most interesting algorithms for this project are shortly presented in this section.

### 2.7.1 Shortest-Widest-Path Routing

In this algorithm, a path in a network is represented by its bottleneck bandwidth, referred as *width* and the total propagation delay, referred as *length*.

The algorithm finds the *shortest-widest* path from a given source to all other nodes, using *delay aided bandwidth search* by first searching the widest paths and then choosing from these the shortest. The algorithm guarantees that only the shortest path among the widest paths is selected, which resolves the problem of arbitrary paths and the loops problem of widest-only path algorithms. The complexity of the shortest-widest algorithms is equal to that of the shortest path algorithms.

Both a Distance-Vector and Link-State version of this algorithm were proposed by Zheng Wang and Jon Crowcroft (see [WC96]).

### 2.7.2 Widest-Shortest-Path Routing

This algorithm is analogous to the *shortest-widest path* algorithm but the search order is reversed: Of all the shortest paths to a destination, the widest is chosen. See [GOW96] for an analysis of the algorithm.

### 2.7.3 Selective Probing

Chen and Nahrstedt (see [CN98a]) proposed a routing algorithm called *Selective Probing* (further analysed in the experiments chapter 3). After a connection request arrives, probes are flooded selectively along those paths which satisfy the imposed constraints. The optimisation is done by choosing the fastest probe and with the introduction of artificial delays when a metric on a link is sub-optimal.

### 2.7.4 Sun-Landgendorfer Algorithm

This algorithm tries to solve the NP-complete delay-constrained least-cost routing problem. It is described in [SL97]. Every router in the network does maintain a separate cost and delay distance-vector table. Because two tables are maintained, two routes from each node are possible for a destination: the least-cost-path and the shortest-path.

A control message is sent to construct the routing path. The message travels along the least-delay path until reaching a node from which the delay of the least-cost path satisfies the delay constraint. From that node on, the message travels along the least-cost path all the way to the destination.

# Chapter 3

# Experiments

## 3.1 Introduction

As said in the first chapter, to investigate different solution methodologies and to avoid working only on one idea which possibly reveals to be not worthwhile pursuing further, impracticable or inefficient the first goal was defining four possible problem solutions, experiment with them and choose the best one, which would then be the basis for the rest of the diploma thesis.

In this chapter the four experimented routing algorithms are presented along with the analysis which was made to select the most promising. At the end of the chapter a comparison between all the experiments is presented and the reasons for having chosen one of them are explained.

The four experiments are:

- Selective Probing based on Routing Hints
- Selective probing with Probes Recollection
- Class based Distance Vector Routing
- Service-Level-Agreement based Routing

## 3.2 Selective Probing based on Routing Hints

### Description

Finding a path for a packet or a flow of packets which can satisfy it's Quality-of-Service (QoS) requirements is a difficult problem. Problematic is the fact that the QoS specifications vary very frequently and it is thus impossible to distribute precise QoS state information to every router in the network. There are fundamentally two ways by which information can be collected: either ”off-line”, where periodic link state or distance information is exchanged independently of the incoming packets, or ”on demand”, where when the route for a flow has to be determined, ”probes” are sent and analysed hop-by-hop. Such a system using probes was described by Shigang Chen and Klara Nahrstedt in [CN98a]. Probing without any knowledge of the network does mean a broadcast on the whole network for each incoming packet, which is not sustainable for big networks. This

experiment is a combined approach which does send probes only to probable paths, based on connectivity information collected off-line.

### Results

After the experiment was conceived, and at the time it was started, a paper of Seok-Kyu Kweon and Kang G. Shin emerged which did already implement very efficiently this idea [KS98]. It was thus decided to not experiment it further. A short description of the algorithm follows.

Hop-count information for each destination and each outgoing link on a node is collected via a Link-State or a Distance-Vector routing algorithm. Based on this information, the search-scope of the probes is limited.

The search-scope is defined as a maximal hop-count which the probe can take to reach its destination. A tradeoff between message overhead and blocking rate has to be found: large search-scopes will generate much overhead but will also find large detours which are perhaps better and small search-scopes will have less impact on the network load but will also fail to find a route more frequently. The right measure has been set in this algorithm by setting the search-scope to the second smallest hop-count to the destination on the source node.

For example in figure 3.1 the hop-count between source A and destination E via B is 3, via D is 2 and via F is 4. The search-scope is thus set to 4. For source A and destination G, hop-count via B is 3, via D is 2 and via F is 2, and thus the search-scope is 2. The algorithm could be slightly modified by adding a small number to the search-scope value.

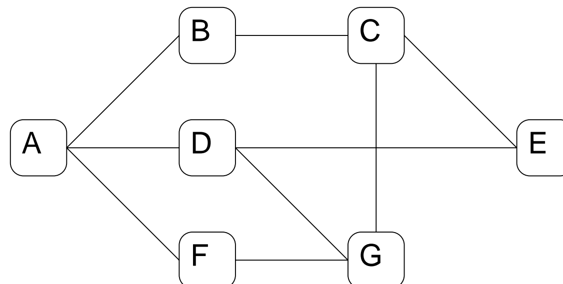



Figure 3.1: Example topology for the first experiment

Each router which receives a probe can check if it is possible for the probe to arrive in less hops than that specified in the search-scope field. If it is possible, it decrements the search-scope by one and sends it further on that interface.

### Conclusions

This algorithm, which is straightforward to implement and which is also light on processing complexity for the router, is shown in the paper to reduce dramatically the message overhead while keeping very low blocking rate values.

## 3.3 Selective Probing with Probes Recollection

### Description

The selective probing method of finding a QoS parameters satisfying route, as described in [CN98a], is very efficient in finding a good route and can be also efficient in terms of messages overhead (see ”Selective probing based on routing hints”). The probes are sent across multiple paths, and the first that comes to the destination does determine the selected route. To make sure that better probes arrive first, an artificial delay is introduced, based on the QoS parameters of the link to which the probe is forwarded. This method of selecting the best probe has the merit of being simple, but it can become imprecise and difficult to administer because of the real delay which is introduced and which is non deterministic. An alternative solution is proposed in this experiment, which should alleviate that problem by collecting more than one packet at each hop before selecting the best to forward.

### Results

The already mentioned paper by Seok-Kyu Kweon and Kang G. Shin did implement such a technique too [KS98]. A short summary is here presented.

In this algorithm, a time-to-live (TTL) limit is set for each generated probe. When a probes-router sees that the TTL of a probe has expired, it discards it. The TTL limit is set by the source using the search-scope of the connection. Since the search-scope indicates the maximum hop-count of candidate routes, the timeout is set larger or equal to the link-delay-maximum multiplied by the search-scope.

The destination doesn’t send the confirmation as soon as the first probe arrives. It does wait and keeps the best received probe in memory (for example with the higher residual bandwidth). When the TTL of the probe expires, it does send a confirmation based on that one.

### Conclusions

This is also a very simple and straightforward implementation, which limits the added confirmation-delay to the minimum, while giving all the probes a chance to arrive in time to the destination.

## 3.4 Class Based Distance-Vector Routing (CBDV)

### Description

In an heterogeneous network with many QoS parameters, the best route for a delay sensitive traffic can be non optimal for a bulk data transfer, where the delay is of secondary importance. In a conventional QoS system (intserv or diffserv) the route for a destination is always the same for each QoS class and it is does optimise only the hop-count. Going from a single-class routing as it is today current to a multi-class routing environment is straightforward by keeping a per class routing table for each served QoS class. Doing so is rather simple and dumb but certainly interesting because of that simplicity and reuse of existing algorithm implementations. This experiment

analyses the efficiency of a possible multi-class version compared to the traditional single-class routing. The comparison should be made based on the call blocking probability for the reservation of flows.

### Results

To make this experimentation possible, a simulator was written from scratch in java (see chapter B). The simulator (`flowsim`) can simulate efficiently a high number of network flows which follow the paths indicated by routers that implement the experiment algorithms.

The experiment consisted of running a large amount of flows on a simple heavily connected topology (we call this topology "Tanenbaum" because it was taken from [Tan96] of A. Tanenbaum, see figure 3.2) and measure the blocking rate when using the traditional distance-vector routing algorithm (DV) and when using the new class-based distance-vector routing algorithm (CBDV). Result of the experiment was then how much better the CBDV algorithm was, compared to the DV algorithm.

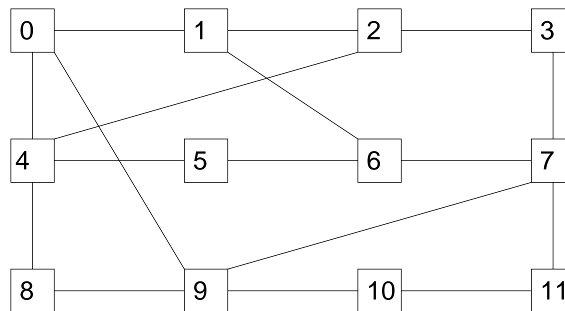



Figure 3.2: Tanenbaum topology

CBDV is only meaningful when used with different classes of flows. It was chosen to implement four different QoS classes:

1. no constraints or best-effort
2. bandwidth constrained
3. bandwidth and delay constrained
4. delay constrained

The class of a QoS-flow is determined by using threshold values for delay and bandwidth constraints. For example if the threshold are set to 60 ms for the delay and 16 kbps for the bandwidth as reported in table 3.1, a QoS flow which wants 20 kbit and supports a delay of up-to 40 ms would be in class 4.

The four flow classes were routed independently by four different routing algorithms implemented with four variations of the normal DV router. Each DV router variation (called CBDV router) did use a different single DV metric (which in the traditional DV is normally the hop-count). That single metric is for the CBDV routers a *composite metric* and is built with local link parameters such as delay, bandwidth and load, which are then combined according to the preferences of the served flow-class.

| Class | Bandwidth [kbps] | | Delay [ms] | | Distrib. |
|---|---|---|---|---|---|
| | min | max | min | max | |
| No Constraints | 0 | 16 | 60 | ∞ | 24% |
| Bandwidth Constrained | 16 | ∞ | 60 | ∞ | 26% |
| Bandwidth And Delay Constrained | 16 | ∞ | 0 | 60 | 26% |
| Delay Constrained | 0 | 16 | 0 | 60 | 24% |

Table 3.1: CBDV Example Classes

For example the CBDV router for delay constrained flows does use as a metric only the delay on the link. The used formula is:

$$invbw = (MAX_{LOGBW} - \log_2(rbw))/MAX_{LOGBW};$$
$$cost = FACTOR_{delay} \cdot delay + FACTOR_{invbw} \cdot invbw + FACTOR_{hop-count}$$

Where: $rbw$ is the residual-bandwidth on the link, $MAX_{LOGBW}$ is the the value of $\log_2(rbw)$ when rbw has the highest possible value and the $FACTOR$s are used to weight the significance of the delay, bandwidth and hop-count parameters.

The delay metric is straightforward, but because of the nature of the bandwidth parameter, how it is used in the composite metric formula is a little complicated. Since the less is better for bandwidth, the inverse is taken. Another problem is that the bandwidth (more precisely the residual-bandwidth, or "rbw") can range from 300 bps to 10 Gbps and thus a logarithm is also made. It should be noted that it is only intended as a hint to find a good route in terms of bandwidth. Because the bandwidth is a concave metric (see section 2.6.1) for a definition of concave metrics) and here it is used as an additive metric, the hop-count does also determine the value. The three $FACTOR$s vary according to the preferences of the served class.

The first scenario did create flows with a Poisson-distributed arrival-time and an exponential-distributed duration. The bandwidth was randomly chosen with uniform distribution from 1 kbps to 32 kbps and the maximal delay also uniform distributed from 20 ms to 100 ms. The bandwidth of the link is fixed at 128 kbps and the delay is random/uniform from 10 ms to 20 ms (the resulting distribution of classes is shown in table 3.1. The source and destination nodes were chosen uniformly random from all the nodes. The results are shown in figure 3.3.

The second scenario was like the first but with the source node was always node number 2 and the destination node always node number 10.

Both scenarios did run exactly the same scenario (with the same sequence of random numbers), which consisted of 60 seconds of simulation with an average inter-arrival time of 10 ms, for different numbers of average concurrent flows in the system. The average number of average concurrent flows was attained by changing the average duration of the flows.

## Conclusions

The first graph (figure 3.3) shows a very similar behaviour for both routing algorithms (DV and CBDV), and thus no big win for CBDV. This is the scenario with the uniform distribution of source

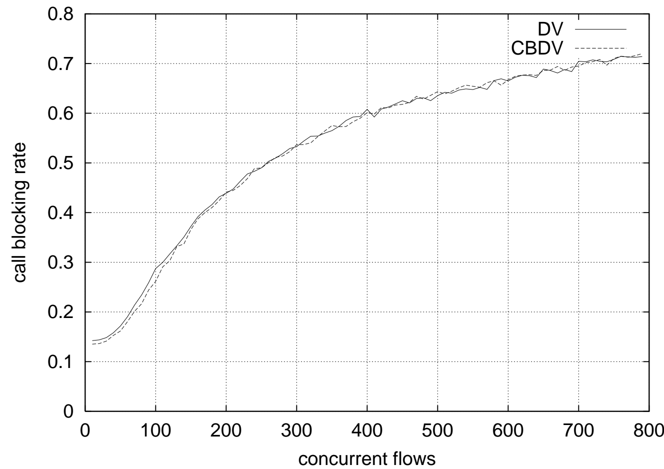



Figure 3.3: CBDV scenario 1

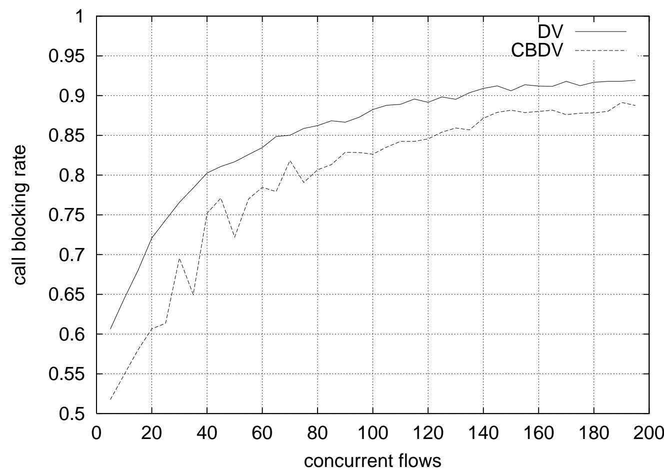



Figure 3.4: CBDV scenario 2

and destination node. Because of randomness, this distribution leads to an homogeneous load on all the links, and thus the CBDV routing algorithm can't make better routing decisions than the traditional DV routing algorithm.

If however a fixed source and destination node are always selected, as in the second scenario (figure 3.4), CBDV can choose alternate paths for the different flow classes, thus making an effective load balancing on the links. The improvement is visible, but even for this very-special case unfortunately not very high (5-10%).

In addition to these discouraging results we must also mention the fact that the messaging overhead for the protocol is much higher than traditional DV, because all the CBDV routers run independently and also send DV updates when dynamic data such as the residual bandwidth changes. It is also very difficult to fine-tune the used *FACTOR*s, the updates-period and threshold.

The biggest problem is however the fact that because the composite metric is directly sent as updates to the neighbours, a *global agreement* on the composition of metrics must be made, which

would be impossible in the heterogeneous Internet of today.

A better alternative, and field of possible research, would be to send not the composite metric, but all the basis metrics to the neighbours, which then build the composite metrics according to local preferences (this technique is already used in the proprietary IGRP from cisco). An algorithm such as “shortest-widest” [WC96] could be used. To alleviate possible counting-to-infinity problems, a path-vectors based approach such as in BGP (see [LR89]) could be used.

## 3.5 SLA Trading Based Routing (SLATR)

### Description

Much of the work on the new differentiated services architecture has been about admission control, queueing, traffic shaping and packet dropping techniques. The routing of the differentiated services packets on the network was completely ignored, even though it is evident that the traditional hop-count based routing can be sub-optimal. Initial work has also been done to implement automatic bilateral agreements on services for a dynamic provision of resources (see [NJZ97]). The idea of this experiment is a natural extension to the bilateral agreements about services (Service Level Agreements or SLA), which does select routes for diffserv flows based on proposals of the neighbour domains (a.k.a. *AS*, Autonomous Systems). Each domain calculates the costs of the various services (identified by aggregate-destination and QoS-class) based on the availability of resources, internal costs and costs of the agreements made with other domains, in order to propose a value-added service to the neighbour domains.

### Results

This is an experiment which required too much time to achieve results such as blocking rate. Instead, a proof-of-concept was implemented. The protocol used to exchange the SLA was designed (see example in section 4.7) and implemented in the `flowsim` simulator (see chapter B).

A very dumb SLA Trader was implemented which does accept every incoming bid, and it was shown, that already with such a simple implementation, correct routing was possible.

### Conclusions

This is certainly a very new and interesting idea of solving the routing problem. With the increasing popularity of diffserv, it could become the method of choice to route the flows between providers.

The fact that an agreement on the protocol has to be achieved only on a peer-relations basis, makes the deployment of it easy and certainly compatible with current routing technologies.

## 3.6 Preliminary conclusions

Table 3.2 summarises the application domain, advantages, problems and possible future work of the experimented routing protocols.

The “SLA trading based routing protocol” (SLATR) experiment was chosen as a basis for the investigated algorithms in this diploma thesis. I think it is the most interesting experiment because

| | **Selective Probing** | **CBDV** | **SLATR** |
|---|---|---|---|
| **Application Domain** | • Int. Serv.<br>• Local Diff. Serv. provisioning | • Generic | • Inter-AS for Diff. Serv. |
| **Advantages** | • Very low block-rate<br>• Load balancing | • Modest overhead<br>• Lower block-rate than pure DV<br>• Easy to implement | • Minimisation of costs<br>• Local preferences<br>• Local protocol<br>• Easy to deploy |
| **Problems** | • High overhead<br>• Not usable as a generic routing alg. | • Block-rate gain low<br>• Global agreement on composite metrics needed<br>• Very difficult to deploy | • SLAT intelligence difficult to implement<br>• Complexity |
| **Future work** | • Most research work already done | • BGP (path vectors)<br>• Shortest-Widest algorithm | • Very new<br>• Further research |

Table 3.2: Experiments comparison table

of the fact that it is intended to work in a diffserv context, which is very new and has gained much momentum recently. Intserv based solutions are now seen as inappropriate for inter-AS domains and will probably in the future only be used as an intra-AS system to provide services to internal or external customers.

The fact that the SLA trading system is easily deployable because of the local decision on the protocol and preferences, makes it also a possible solution to the existing routing problems in the inter-provider (inter-AS) space. Appealing is also the simplicity of how the whole system works.

# Chapter 4

# SLA Trading Concepts

## 4.1 Introduction

The *diffserv* architecture uses *Service Level Agreements* (SLA) to define formally bilateral services quality and admission control.

Because of the heterogeneity of the Internet and it's services, it is almost impossible to establish manually each of these contracts, and it would be certainly an advantage to introduce automatic *traders* which would establish service level agreements independently with peer traders using some criteria.

In the figure 4.1, already shown and explained in chapter 2, the location of the SLA Trading System is shown with a point. *Allocated Capacity*, because it uses *diffserv* and *Destination+Class* because, as will be explained in the following sections, routing is done on the basis of destination *and* class.

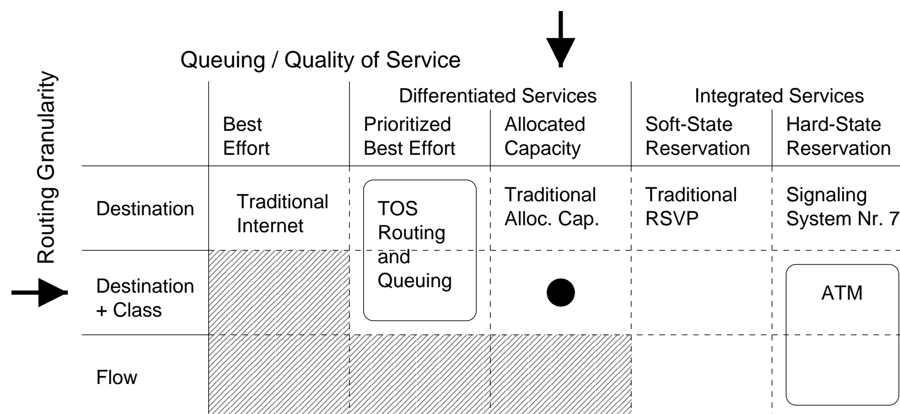



Figure 4.1: QoS Provisioning and Packet Classification

## 4.2 Service Level Agreements

**Definition:** A *Service Level Agreement* is a bilateral service contract between a customer and a service provider that specifies the forwarding service with QoS guarantees the customer should receive.

An example SLA could be a contract for the delivery of traffic with maximal bandwidth 64 kbit and guaranteed less than 200 ms propagation delay for destination New York and for a total of 10 Mbit. A SLA can be seen by the customer as a sort of channel with some guarantees on what it does put in it.

The SLA can use different degrees of assurance, i.e. they can be made more or less strict on the guarantee of the delivered service. It could be possible to also represent the actual best-effort delivery of packets as SLAs: connectivity would be assured in the contract. Even in best-effort SLAs, some guarantees could be made, such as fairness for example.

### 4.2.1 Premium Service SLA

*Premium Service* does provide strict guaranteed Quality of Service by allocating bandwidth in a way that does no over-commitment. In other words, when a provider sells a Premium Service to a customer, it must allocate it's resource so as to be able to deliver the guaranteed traffic even at congestion moments.

Premium Service was first described by Van Jacobson in a talk at the 39. IETF Meeting in Munich and is explained in [NJZ97].

Note that the fact that over-commitment (i.e. selling more resources than what is available) is forbidden, because of the hard guarantees, does make the network utilisation very sub-optimal in case only Premium Service was used. The problem is that resources are allocated even if they aren't used.

### 4.2.2 Assured Service SLA

David Clark, author of the initial *diffserv* architecture proposal (with Jacobson), did propose *Assured Service* as basic diffserv service (see [CF98]).

With Assured Service, *statistical guarantees* are made. In other words, it is said that if the contract is taken by the customer, it's packet will be forwarded with a quality of service that will almost always be within the specifications of the contract.

The provider does provision it's network and sell services so as to make it all work within the specification if the users behave in a "statistical" way, i.e. if not everybody wants at the same time to make a connection to the same destination for example.

This type of service does solve very well the problem of network usage inefficiency and is certainly much better than best-effort. The problem is that it is difficult to quantify the level of assurance which can be made to the users (what does "almost always" mean?).

To make an analogy, the postal delivery of letters could be called an assured service: the guarantee is made that the letters will arrive the next day at destination. If however everybody did send many letters the same day, the postal delivery offices wouldn't be capable to deliver the letters within the specification.

Van Jacobson proposed a system with both service classes, Premium and Assured, proposed in a "Airline" way, where First, Business and Economy classes are provided. The advantage of such a hybrid system is that some applications *need* to have hard guarantees on the delivery, while most others will be adequately served by *assuring* the service.

## 4.3 SLA Trading

Service Level Agreements do constitute *goods* in a market of service providers. A provider does buy a link to a destination and it can thus issue Service Level Agreements for that destination. It can propose to others a guaranteed delivery under some conditions to that destination and sell such services in form of SLAs.

Consider the services proposed to it's customers by a provider. It will propose services in form of *SLA bids*, which will be evaluated by it's customers and perhaps bought, depending on their interests and on the attractivity of the bids.

How does a provider provision it's resources to be able to make SLA Bids for a destination to it's customers? This could be made for example by buying a physical connection to the destination in question. Another possibility would be to buy, as customer, that service from another provider, which proposes that service.

The SLA bids will have in the typical case also a *cost* to make the contract. As in any other trading system, you trade a good for another one, and in this case it will be mostly SLAs for money.

A system where SLAs are sold and bought in such a way is a *SLA Trading System*. This *SLA Trading System* is a very typical trading system known from basic economic courses with elements and properties such as buyers, sellers, demand, supply, etc.

## 4.4 SLA Traders

This trading of SLAs could be made manually but it would be very impractical because of the amount of information to be taken into account and on the quantity of SLAs which are required to be bought or prepared for selling. An automatic *SLA Trader* would be therefore advantageous.

Without considering the external provisioning of resources (for example by leasing a telephone line), the SLA Trader (be it human or a computer program) responsibilities are:

- **Bookkeeping**: Keep track of sold and bought SLAs
- **Provisioning**: Buy SLAs so as to be able to propose services in form of SLA bids to customers
- **Pricing**: Prepare SLA bids for customers

These three responsibilities will be further discussed in the following sections.

### 4.4.1 Bandwidth Brokers

Van Jacobson did in [NJZ97] also propose a *Bandwidth Brokers* architecture which are similar to *SLA Traders*. Bandwidth Brokers did inspire the SLA Traders concepts, but are somewhat

different:

- Only *bandwidth allocations* are managed as opposed to SLA contracts
- Trading is done only on a fixed path, and thus no competition is possible (see section 4.8)
- Routing isn't taken into account (see section 4.6)
- The protocol is two-way (see chapter 5 for a description of the SLA Trading Protocol)

## 4.5 SLA Bids Generation

### 4.5.1 Price

The ultimate goal of commercial providers is to make money and this is also reflected in how the SLA traders will behave: They will try to make money from the selling of services.

The price of the sold services will be calculated to cover the costs of the provisioning of services and to make some profit.

How to price the services is, with the decision on which services to buy for the provisioning, the most difficult task of the SLA trader and further discussion will be made on that problem in the following sections.

### 4.5.2 QoS Guarantees

When deciding which quality of service guarantees to give to the customer, the quality of service of the resources on which that service is based should be taken into account.

For example, if a provider has bought a SLA to a destination from another provider with maximal delay of 20 ms and wants to make a bid to a customer which is based on that bought SLA, the guaranteed maximal delay will be at least 20 ms plus it's own propagation delay.

## 4.6 Routing based on SLA Trading

If a provider wants to buy some service to a destination and more than one of it's neighbours does send him bids, he will have to make a decision on which one to take. Making such a decision is doing a *routing decision* because the services path for a destination across the network is so chosen.

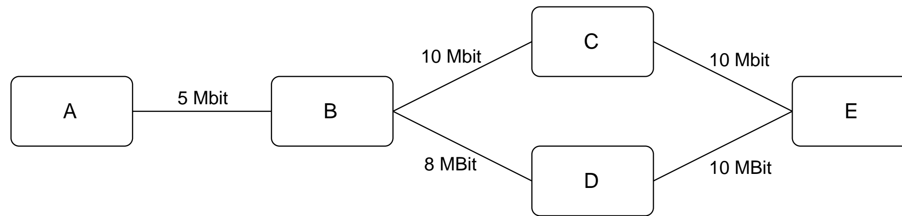
Distance Vector routing could also be considered a form of SLA Trading: every router proposes *best-effort* SLAs to it's neighbours and the best bids are the bids with the shortest *hop-count* indication.

When a packet for a destination should be forwarded by a provider, it will look at it's bought SLAs and will forward the packets accordingly.

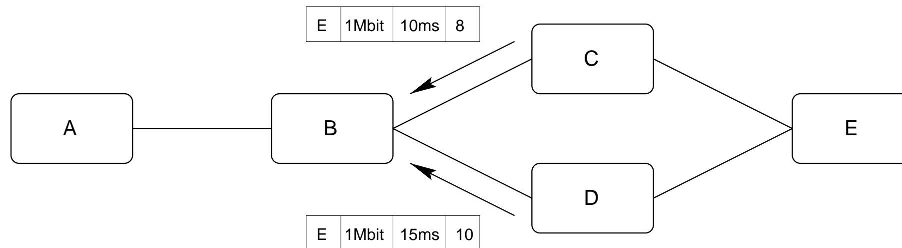
A short example is presented in the next section on how the routing is done and the effects of doing it so.

## 4.7 SLA Trading based Routing Example

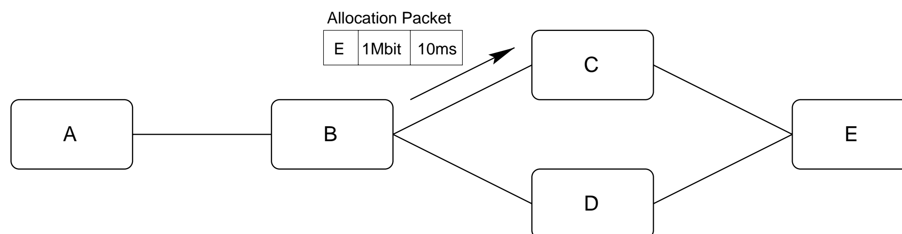
We will now describe a short example to show how SLA Trading based Routing works. With a simple 5-nodes topology the trading mechanism is shown.



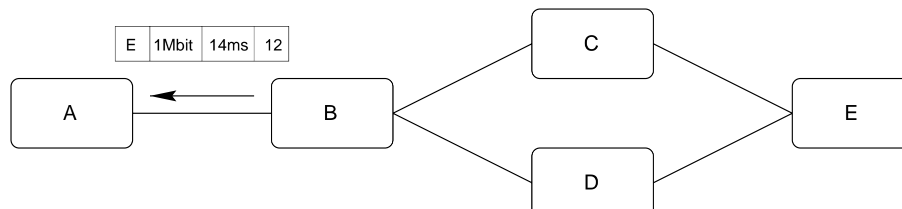

We concentrate ourselves on the bids relevant to a possible 1 Mbit flow which would be set-up from A to E. Node B receives from nodes C and D two different proposals (in form of ”bids”) for accepting 1 Mbit flows. Node C makes a better proposal (for example because it has a faster path from B to E): cost of 8 for a 1 Mbit flow to E with maximal 10 ms delay. Node D's bid has a higher cost and higher maximal delay.



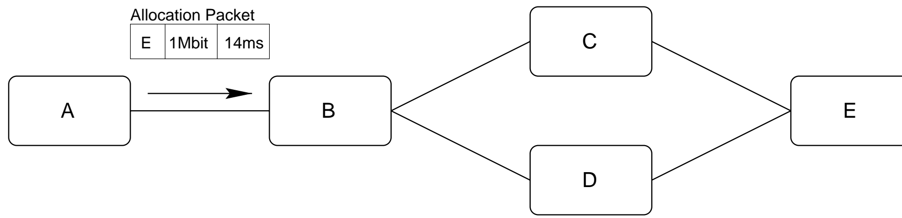

Node B now known that it can send in the next time-frame (also specified in every bid, but omitted for clarity) up to 1 Mbit for E through C with a cost of 10 and a maximal delay of 8 ms. It knows that probably someone will be interested in sending a flow to E, so it does buy the service.



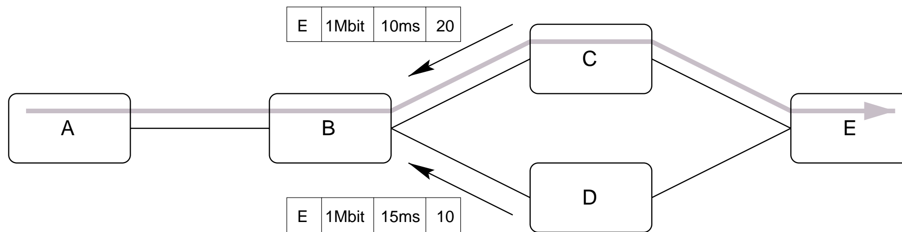

It can now make a proposal of service to A based on the resources it has. In this example it has for example an assured-service to E through C. It adds a margin of delay, which becomes 12 ms and it increases the cost to 14 to cover it's internal costs and to make some profit.



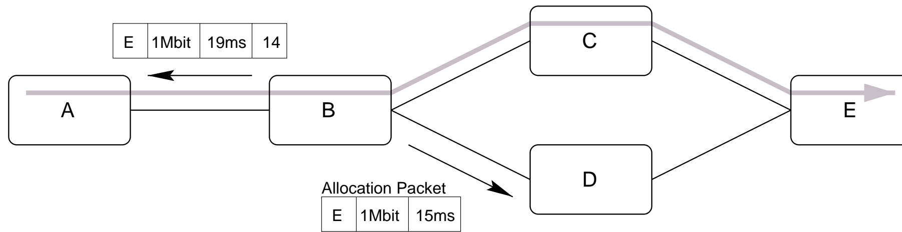
A can now allocate the resources for the flow it needs to setup.



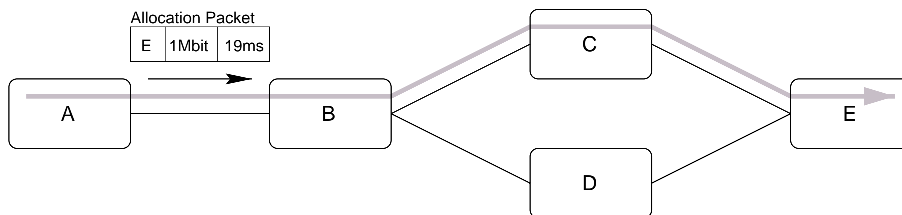

The flow does occupy some resources and the proposal from C to B for the next time-frame changes according to the modified residual bandwidth. The price is increased from 8 to 20.



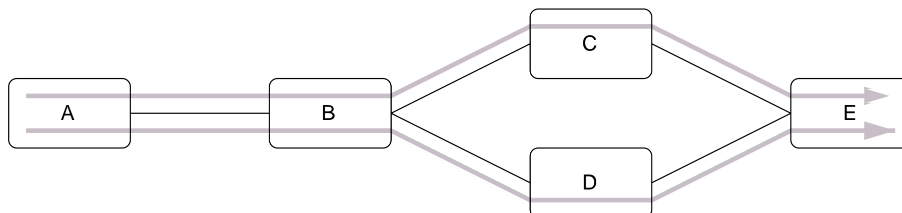

B has now to compute the next bid for A. It estimates that a new flow is likely for destination E which doesn't have very-low-delay constrains, and thus takes the bid from D which costs less.




A buys the service.




The result is a network utility which is maximised according to local preferences.



## 4.8 Traders Competition

In the previous example, it is clear that when more than one neighbours propose a service to a trader, these neighbours *compete* to sell that service. As said earlier, it is from this competition that the routing is done, i.e. with the selection of the best bids.

Competition is of primary importance for the SLA Trading System, because, as in a capitalistic economic system, it is the competition that guarantees the minimisation of the costs. If a provider has a much faster and less expensive “connection” to a destination, it is better for the overall efficiency that his services are being preferred over the others.

If it is possible for everyone to setup the required resources and make SLA proposals, it also probable, that if at a given time the price for a service is much higher than what could be made, another provider will come and take the customers.

To the horrified readers it should be remained that we are speaking of inter-provider services with guarantees, something that will never (in our society) be provided for free. That form of competition is just the technical transposition of already current processes.

## 4.9 User Competition

When a resource is scarce or when it is very expensive, the price of bids will likely be high. A sort of *competition* arises between the users: the user willing to pay more for services is likely to get more services. Users which find a resource more valuable get it.

This *user competition* does also improve network usage, because expensive resources won’t be used for applications that aren’t worth it. A satellite link won’t be established to transfer Usenet news.

## 4.10 Bid Guarantees and Bids Flooding

There are two basic ways a provider can make *bids* to it’s customers: it can give them the exclusive right to sign them or it can propose the same service to many customers and serve the first come.

*Exclusive signing rights* are a guarantee that if the customer does respond before some expiration time the provider is guaranteed to give him that service to the indicated price. See the protocol specification chapter 5 for a possible implementation. The provider could even give the customer that service for free up to that expiration time. This type of bid would be a *try and buy* offer: if you are satisfied, you buy it.

Note that in both cases the provider must not over-commit the bids but treat the bids as temporary agreements. This is certainly expensive for the provider and it is imaginable that such a business model could be used for low cost services.

The big advantage for that type of bids is the fact that the providers can verify customer demands using those bids and decide later to buy them or not. No trend analysis is needed. The provider could even issue that type of bids with guarantees based on bid guarantees it has got from other providers! Such a system results in a *flooding of bids*.

That *flooding of bids* does permit a very fast setup of resources for new demands. Distant providers do know about the possibility to take a service before some provider in the path does think there

could be interest in providing that service.

## 4.11 Trend Analysis vs. Explicit Asks

How does a trader decide which bids to take? This can be done it two basic ways: using *Trend Analysis* or waiting for *Explicit Asks*.

In *Trend Analysis*, the trader does already have some traffic, and on the basis of the evolution of it, it can decide what the most probable needs in the future will be and buy services accordingly.

A trader can also send *Explicit Asks*, messages containing a service request indication, to another trader asking for a particular service, because it wants it and because it doesn't have received a bid for that. Traders could be designed which doesn't make any pre-provisioning of resources but which wait *Asks* from customers.

Note that if a trader does receive a "Ask" for a destination which it already did provision in expectation of future demands, it will respond much faster than a trader which does first need to buy the resources for it.

## 4.12 Profit Optimisation

The competition makes life much more difficult for traders. They are faced, when calculating the price of the SLA bids, by two opposing optimisation goals:

- Maximise the profit made from the selling of SLAs
- Minimise the costs of SLA bids to improve the bids attractiveness

The difficulty arises from the fact that the higher the price of a bid, the higher the profit when selling it, but the less the probability that it will be sold.

Another difficulty is at the provisioning level. Here too, there are diverging optimisation goals:

- Maximise the available resources to be able to make good bids
- Minimise the resources costs

Should the provider buy many SLAs such as to propose to it's customers many diversified services or should it only buy the SLAs which will be certainly sold? Proposing more services does improve the probability of selling services but does also increase costs. The optimisation goal is finding the right balance.

## 4.13 Service Loops

When designing a routing system it is always of fundamental importance to provide a *loop-free* system. Using SLA Trading terminology, this does mean that no service should be sold and bought by the same provider to provision it.

Consider the delivery service to A in figure 4.2 from provider C. Suppose that the link from A to B is full: B will generate no more bids to C. Suppose further that C did (before the link was full) buy some SLAs from B for destination A and that these SLAs are almost completely used because of generated SLAs sold to D and E. C, seeing that it is becoming low on resources to A and seeing that that service does sell well, does try to buy some more service to A. E, which for some reason did buy some service for A from D and not from C, will make an offer to C for going to A. Here comes the service loop: C, which is really bad at commerce, does buy the proposal of E for destination A! It will be able to further make offers which are based on this new bought service and so on. A infinite SLA loop is made.

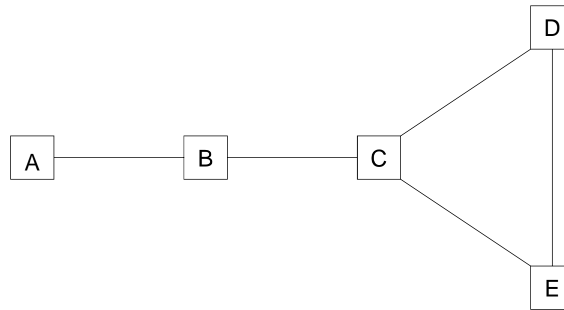



Figure 4.2: SLA Service Loops Example Topology

Let's further analyse the bought SLAs in this service loop with the notation [destination provider bandwidth max_delay price]:

1. C sells [A C 1MBit 20ms $20] to D. C has no more bandwidth to A.
2. D sells [A D 1MBit 30ms $25] to E.
3. The link A-B is full ...
4. E sells [A E 1MBit 40ms $30] to C. C has 1MBit to A.

What are the consequences after the first service loop? C has lost $10! At each loop, if every trader adds $5, C will spend $10 more for nothing. Note also the progression of the maximal delay guarantee.

Are such loops dangerous for the overall network functioning? No. The guarantees are always maintained. The only consequence is a loss of money for C.

Such service loops are the consequence of poorly designed traders, which will be automatically taken out of competition from the system because of it's losses. This can be seen as the network punishing bad traders and rewarding well designed ones.

## 4.14 Messaging Overhead

Every Trader does own each link to it's neighbours in the direction from him. Every bid does obviously occupy some bandwidth on that link, if in-band signalling is used. The trader is thus faced with this further optimisation problem:

- Make the most bids possible so as to find exactly what the user wants
- Make the less bids possible so as to minimise the link utilisation and be able to sell services which use the link's bandwidth

## 4.15 Trading Styles

In this diploma thesis, four basic trading styles were designed and implemented. The implementations are further discussed in chapter 7.

### 4.15.1 Greedy Trader

This Trader does buy every bid it does receive. This is clearly very dumb and the implementation is used as a proof of concept that bad traders are eliminated from competition.

### 4.15.2 Trendy Trader

The *Trendy* Trader does buy some basic connectivity services to each destination node, by choosing the best bid and tries to sell services based on these. As soon as a bought service begins to be full because of sold services based on it, it tries to buy some more of that by choosing a similar bid with the smallest price.

This trader was used in the SLA loops example to show how services loops are made by bad traders.

### 4.15.3 Profitable Trader

The *Profitable* Trader does behave like the Trendy Trader but does only buy more services when one is almost full if the best bid passes a *profitability* test which says if based on the price by which services were sold in the past and by evaluating the expected usage it would be profitable to buy it or not. This test would have eliminated the offer of E in the SLA loop example.

### 4.15.4 Lazy Trader

The *Lazy* Trader doesn't buy anything. Another trader has to first *Ask* him a service and then he will try to buy services to that destination by buying received bids or by generating itself "Asks". See the next chapter for a definition of *Asks*.

## 4.16 Summary of Design Issues

When designing a SLA Trader, the following problems should be solved in the better way possible.

- SLA and external provisioning cost minimisation
- Bid utility maximisation and price minimisation

- Bid precision and messaging overhead

The trader which solves these problem better will gain more than the others. This is a very good incentive to stimulate the development of very good traders which use the minimum of resources while giving the maximum of utility to the users.

# Chapter 5

# The SLA Trading Protocol

## 5.1 Introduction

The previous chapter did present the SLA Trading concepts. Among the properties of the system, there is the *protocol independence* which stems from the bilateral-only relations. This *protocol independence* does guarantees an easy deployment of the algorithm, because no global agreement on the protocol must be attained to make it work.

A simple three-way handshaking protocol for SLA trading was designed and is here presented to show a possible working implementation.

The protocol is implemented using five message types, which can be shortly translated in English:

- **Ask**: (Buyer) "I want this service. Could you make a proposal?"
- **Bid**: (Seller) "I propose you this service agreement."
- **Accept**: (Buyer) "I want to buy this proposal."
- **Confirm**: (Seller) "I confirm to have made a contract with you for that service"
- **Reject**: (Seller) "Sorry but it isn't possible to make that contract anymore"

## 5.2 SLA Specification

A *SLA Specification* should define the full contract and is using that specification that the customer will choose to buy it or not. The specification should contain two basic group of definitions:

- Admission Conditions
- Quality of Service Guarantees

The *Admission Conditions* does define the conditions that the customer traffic should respect so that the QoS guarantees are respected by the provider. Admission Conditions can also be seen as the identification of the traffic which will be treated by the provider in a way to respect the QoS

guarantees. Typical specifications will include *destination* and traffic quantity in form of *traffic volume* in Mbytes for example or a service expiration.

The *Quality of Service Guarantees* does define the guarantees that the provider makes to the customer for the traffic which falls within the Admission Conditions. Typical specifications will include *maximal delay*, *minimal bandwidth*, etc.

## 5.3 SLA Identification

A way to identify SLAs is needed so that the protocol can refer to it in a unique way when doing the hand-shaking for buying it.

Each SLA is uniquely identified by a provider with the following specifications:

- Destination
- DS Byte
- SLA Id

Note that in the *Flowsim implementation*, discussed in chapter 7, the identification is only done with Destination and DS Byte, because the DS Byte is incremented for each new SLA for that destination. See chapter 7 for further explanations on the subject.

## 5.4 Messages

### 5.4.1 Ask

With a *Ask* a customer (the *buyer*) does request the services provider (the *seller*) to make a bid for a destination and for some Quality of Service specifications. This message should be regarded as a *hint* for the services provider and is not required. The provider receiving the Ask will normally answer this Ask by issuing an appropriate *Bid*, but it is his own decision to do so or not.

An Ask message should contain:

- Destination
- Quality-of-Service specification (for example bandwidth and maximal delay)
- Expiration

The *Expiration* field does specify for how long the service is desired.

Figure 5.1 is the *Message Sequence Chart* (MSC) of how the message is sent. Note that the box surrounding the figure does represent the system controlling the messaging process. The messaging process, is controlled by it's input (arrows going from the box) and does provide some output to the system. See [MSC] for descriptions of MSCs. The *ASKreq* does indicate the request by the buyer to it's messaging process to make that ask to the buyer. The buyer is then notified by it's messaging process of the customer ask.

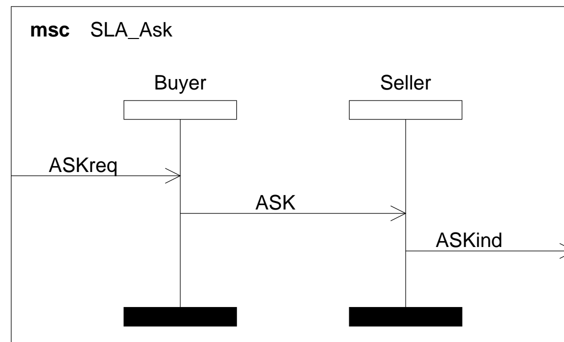



Figure 5.1: SLA Trading Protocol: Ask

### 5.4.2 Bid

A *Bid* is a SLA proposal that the seller does to the buyer. A Bid is the first of a three-way handshaking necessary for the establishment of valid SLAs between a customer (buyer) and a provider (seller). The MSC of the handshaking is drawn in figure 5.2. Note in the MSC that the protocol is started by a *BIDreq* request by the seller, which does so indicate the messaging process to make a bid to the customer. This request could follow a previous *Ask* made by the buyer.

The buyer will receive a *BIDind* indication from it's messaging process with the contract proposal specification.

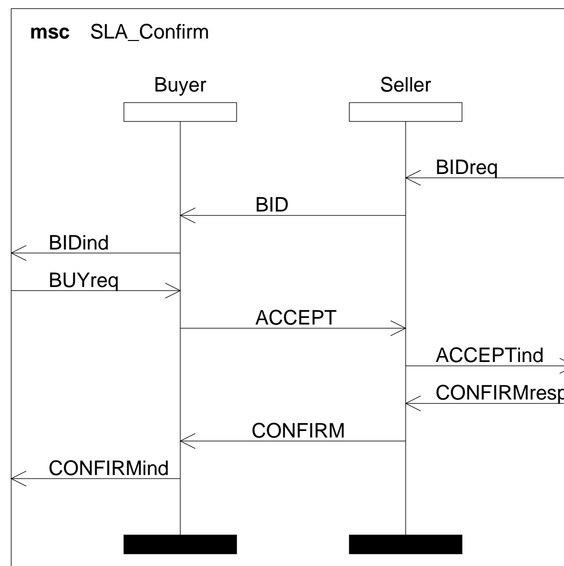



Figure 5.2: SLA Trading Protocol: Bid, Accept and Confirm

The contents of a bid are:

- SLA Identification
- SLA Specification
- SLA Price

- Guaranteed-Confirm Expiration

The *SLA Identification* should be used as a way to identify the SLA in the other messages of the handshaking and was already discussed in section 5.3.

The *SLA Specification* does define exactly the conditions of the contract. This is the only message in which these conditions are given and they will be referenced only with the usage of the SLA Identification. SLA Specification was described in section 5.2.

*SLA Price* defines the cost of the service if taken by the customer.

The *Guaranteed-Confirm Expiration* does define for how much time the customer is guaranteed a positive answer (i.e. a *Confirm*) if it does accept it. This specification can be omitted, depending on the business-model used (see 4.10). Figure 5.3 shows a time diagram for the *guaranteed confirm expiration*: if the customer responds with an accept before the *accept limit* (the guaranteed-confirm expiration), it should receive a confirm from the provider.

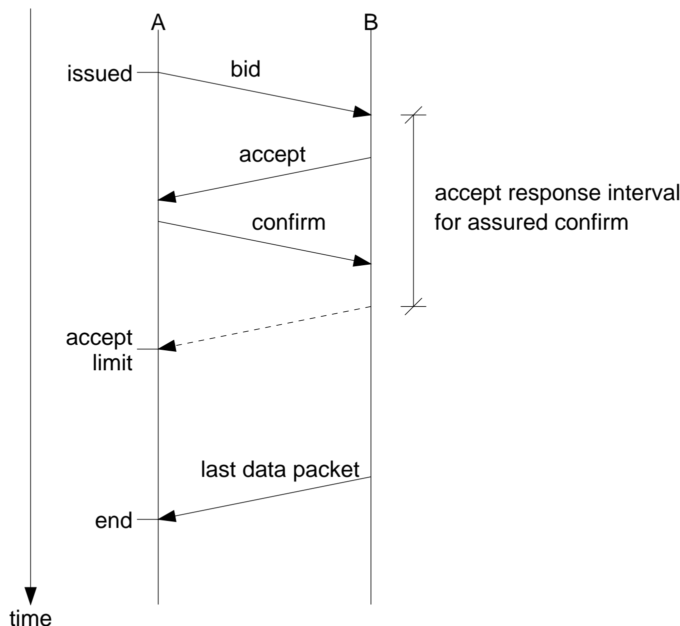



Figure 5.3: Guaranteed Confirm Expiration

### 5.4.3 Accept

When the customer wants to buy a service, it does send an *Accept* message containing only the *SLA Identification* of the SLA in question.

A customer may receive many Bids without responding, choose the best one and send an *Accept*. In other words, it is not required that the customer respond immediately to each bid.

Note that with the *Accept* message, the customer does only *indicate* that it would like to buy that service. The customer shouldn't make any assumption about the contract until the *Confirm* message is received.

### 5.4.4 Confirm

To the *Accept* message, the provider must give (as soon as possible) the confirmation to the customer if it is still possible to make the contract, thus rending it *valid* from this moment, or if the conditions have changed and the contract can't be made.

The *Confirm* message, which does also only contain the *SLA Identification*, is the positive answer from the provider. The provider does, by sending it, make the contract valid.

Note that if a *Guaranteed-Confirm Expiration* was given in the Bid and the customer did send an *Accept* in before that expiration specification, the provider is *expected to confirm* the Accept.

### 5.4.5 Reject

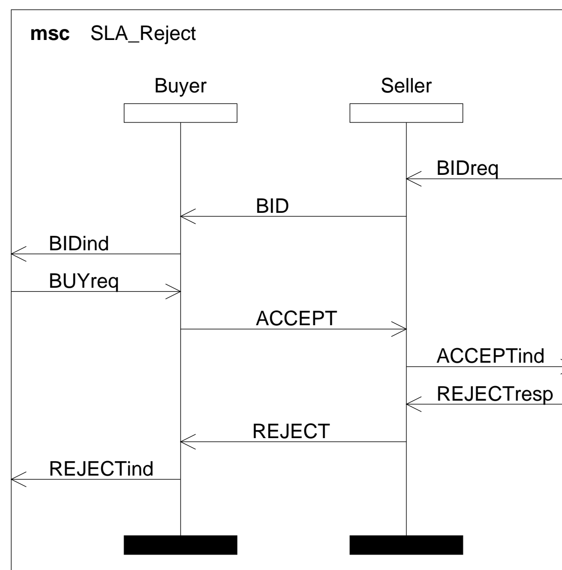



Figure 5.4: SLA Trading Protocol: Bid, Accept and Reject

If the provider can't confirm the *Accept* it must send a *Reject* message, as shown in the MSC of figure 5.4. The Reject message does also contain only the SLA Identification.

## 5.5 Possible Improvements

The possible improvements which can be made in such a protocol are made to improve the *flexibility and utility* or to reduce the *messaging overhead*.

*Flexibility and utility* is probably difficult to improve, because these basic constructs are sufficient for almost every imaginable business model.

On the contrary, in the area of *messaging overhead*, many improvements and tricks can be used to reduce it. The messages which makes for the most overhead are certainly the bids: the providers are interested in specifying and diversifying their service proposals the most possible so as to find exactly what the customer wants.

Possible ways to reduce the Bids overhead are:

- A way to reduce the Bids overhead is to group them. For example, a common price to many destinations could be found and only a bid, specifying the destination group and the price could be made.
- Specify a *pricing function* for a group of bids for example in function of bandwidth and maximal delay so that the customer can calculate the price of the service it wants. The *Accept* message would then specify exactly the desired service.
- Send only differences (deltas) to the customer. For example with update message on the expiration, on the price, etc. See [Jac90] for such a technique used for IP-header compression.

Since the SLA Trading System is independent of the protocol used, it is an incentive for provider peers to reduce at the minimum the messaging overhead while keeping good bid attractiveness.

# Chapter 6

# Users Preferences

## 6.1 Introduction

To produce meaningful results, users of the network must also be simulated. Different type of users such as a 'voice user', a 'web user' and so on with their differing preferences should be modelled. It will then be possible to measure the performance of the algorithms and protocols presented in this work by the "user satisfaction", which is the ultimate goal.

A model of user preferences and selection of services was designed with micro-economics ideas adapted to the problem.

## 6.2 Utility

Utility is a way to describe preferences and to recognise a product that is better than another one. Hal R. Varian defines it so (see [Var96]): "A utility function is a way of assigning a number to every possible consumption bundle[1] such that more-preferred bundles get assigned larger numbers than less-preferred bundles". This is the ordinal property of the utility function, because the utility value of a service has no significance by itself. It has significance if compared to the utility value of another service, thus making it possible to "order" them according to their utility.

We further extend this definition to include also a cardinal property, so that it can be said "this bundle is twice as good as that one, because it's utility is the double". What "twice as good" means is clearly very subjective and it does need a definition, so that we agree on a common utility function for every situation. We define that a good is twice as good as another one if you would pay twice as much for it, three times better if you would pay three times as much, ...

The utility will be used to classify the perceived quality of a received service when a specific request is made to the service provider. To give a meaning also to the absolute value of a utility value, utility 1.0 is defined as "maximal utility", i.e. you are completely satisfied by the service (bundle) you received in comparison to what you asked.

[1] a "bundle" is, using networking terminologies, a "service"

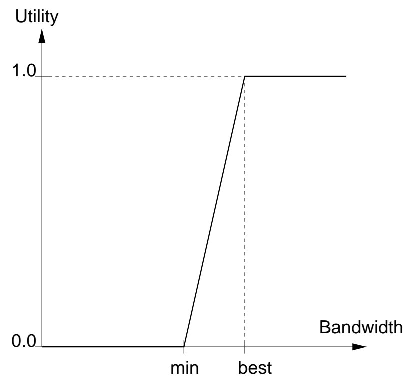



Figure 6.1: Utility function for voice flows in function of bandwidth

### 6.2.1 Voice Application

Consider for example a voice application user. Possible utility in function of bandwidth as depicted in figure 6.1. The bandwidth does have a “best” value, which corresponds to the normal rate of transmission of the voice application in full quality, therefore the utility is 1.0 because it is for you “ideal”. It more bandwidth than “best” was given to the application, nothing would change and it wouldn’t make sense to pay for that additional bandwidth, which explains the behaviour of the curve after the “best” point.

If the bandwidth is less than “min” the application can’t work and therefore it’s utility is 0 (i.e. you would pay 0 times the full price for that service). Between these two extremal points we do make the simplifying assumption that the behaviour is linear. This does mean for example that you would pay half of the full price for best quality if the received bandwidth is $(best - min)/2$.

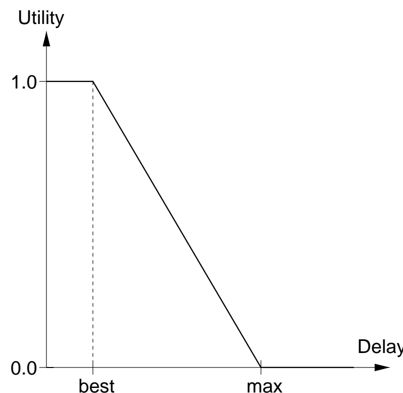



Figure 6.2: Utility function for voice flows in function of delay

In figure 6.2 the curve for the utility in function of delay is drawn. The behaviour is very similar to the bandwidth relation to utility but is inverted because higher delay means lower quality. A delay lower than “best” isn’t perceived in a conversation and a delay higher than “max” is unacceptable.

### 6.2.2 File Transfer Application

Another example is file-transfer application user. This user wants to transfer files and the only thing that does matter to him is the total required time to transfer those files. The utility is thus only dependent on the transfer-time.

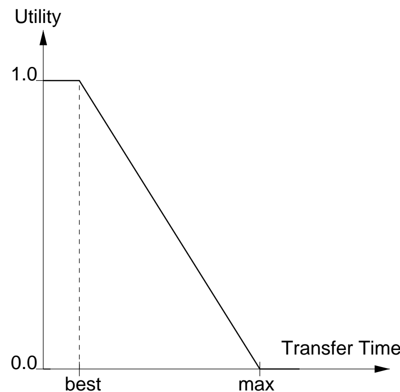



Figure 6.3: Utility function for file-transfers in function of transfer-time

In figure 6.3 the curve for the utility in function of transfer-time is drawn. The user won't even note the difference for transfer-times below *best* and does consider transfer-times above *max* unacceptable.

## 6.3 Willingness To Pay

Let's consider more in detail the relation between utility and what you are willing to pay for that utility. We have defined the utility so that the relation is *linear* and so that for example you are willing to pay half of full price for utility 0.5. The question now is: what are you willing to pay for the full price (i.e. for full quality)? This is clearly dependent on the requested service, and we just define it as *WTP* (*willingness to pay*).

The graph of what you are willing to pay in function of the received utility is a line such as the one labelled *indifference line* in figure 6.4. Consider a random point on this line with utility $U$ and cost $C$: that point represents what you would pay for a service of utility $U$. If a service of that utility was proposed to you but with a higher cost $C+dC$, that would be more of what you are willing to pay and you wouldn't take it, i.e. if you were forced to take it, you would be worse than not taking it. If on the other hand the same service was proposed to you for a lower cost $C-dC$ then you would happily take it and would feel better than not having taken it. We can then say that buying that service at the price $C$ is to you indifferent as not buying it.

We call this line *indifference line* so, because it does represent points were it is for you indifferent staying. Being over that that line is for you "worse" (you pay more than what you are willing to pay) and being under is "better" (you pay less than what you are willing to pay).

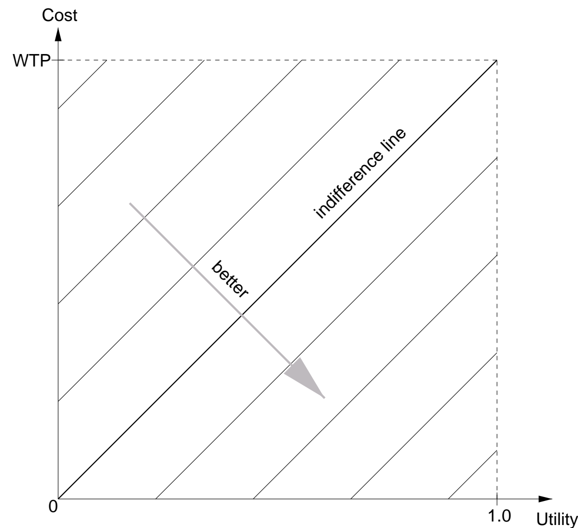



Figure 6.4: Willingness to pay diagram

## 6.4 Bids Evaluation

Indifference (see figure 6.4) can be relative to not buying any services or can be relative to buying other services so that you can say these services are for me equal good to take. Indifference lines can be drawn parallel to the one originating in $(0,0)$.

Consider the situation where the user wants a service from the service provider and it receives different proposals in form of $(utility, cost)$ pairs. An algorithm should be used to select the *best proposal* (or *best bid*) according to it's willingness to pay for the service.

For every bid, the corresponding cost of it's utility on the WTP line is taken so that it can be said that that cost is what you are *willing to pay*. In figure 6.5 that cost is labelled $A$, and the real cost of the bid is labelled $B$. It can be said that if you take that bid, you will save the difference of cost between $A$ and $B$. We call that difference in cost *gain*. The gain of every bid is calculated, and the bid with the highest gain is selected.

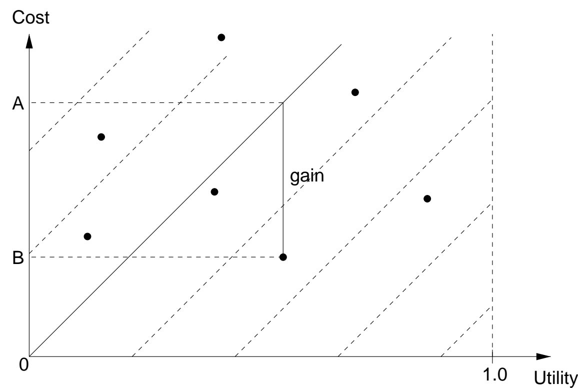



Figure 6.5: Bids evaluation by the user

# Chapter 7

# Implementation

## 7.1 Introduction

A simulation of the SLA Trading System was built with `flowsim` for the following reasons:

- Functional verification and experimentation of the SLA Trading Protocol.
- Study the feasibility of SLAT-routers which make good decisions of which services to buy and evaluate different business models.
- Show that the whole system can function reasonably. For example show that it can be used to route flows, i.e. that good paths from every source to every destination with no loops are found.
- Show the advantages of this system in comparison of other systems, such as Distance-Vector routing with Int. Serv. reservations, and that it is indeed good for inter-ISP routing.

A description of the SLA Trading System implementation in `flowsim`, which was put in the `flowsim.slatr` package, is presented in this chapter. Consult appendix B for the documentation of `flowsim`'s core classes. This is not a supplement to the `javadoc` documentation which should be regarded as a reference.

## 7.2 SLA Trading Protocol Messages

The five messages of the SLA Trading Protocol, described in chapter 5, share the same base class: `SLATP_Message`. Every `SLATP_Message` is sent from `Service` to `Service` (and not from `Node` to `Node` such as in the Distance-Vector implementation, described in appendix C). The base `SLATP_Message` implementation stores the source and destination `Service` and does provide some commodity functions used by all messages.

Every message is implemented in a separate class such as `SLATP_Ask` for *Ask* messages. The object does contain all the data pertaining to the message and provides a method to construct and send in a easy way messages of that type called `send`. For example the `SLATP_Ask` implementation's `send` method interface is so:

```
public static void send(Simulator s, SLABuyer from,
                SLASeller to, Node dest, int bw,
                int max_delay, long expiration)
```

Source must be a `SLABuyer` (described in a following section) and destination must be a `SLASeller`. `dest` is the destination for the wanted service, `max_delay` and `expiration` are other service specifications.

## 7.3 SLA

SLAs are used almost everywhere in the `flowsim.slatr` package. Great flexibility is thus wanted from the SLA implementation. Many additional data is needed in relation to specific SLAs. A very clean and nice implementation would have defined the SLA class as containing only what is pertaining to SLAs and would have used composite classes to expand them with implementation specific data. Since, unlike the `flowsim.core` implementation, this sources were only written for the purpose of simulating SLA Trading scenarios in this diploma thesis, efficiency of simulation and minimisation of complexity was chosen.

For that reason the SLA class does contain many fields irrelevant to SLAs such as *residual bandwidth*, *cost*, *interface number*, etc. The SLA fields are:

| | |
|---|---|
| `dest` | service destination |
| `ds` | ds-byte of this SLA |
| `bw` | bandwidth |
| `max_delay` | maximal delay |
| `time` | timestamp when this SLA was issued |
| `expiration` | expiration of service |

Every SLA object does also contain, as explained before for commodity reasons, it's *price* or *cost*. Methods are also implemented which calculate the *volume* of the SLA in *bits* (using the timestamp and the expiration field). *cost per volume* methods calculate the cost divided by bits and the *cost per time* the cost divided by total time.

## 7.4 SLABuyer

A `SLABuyer` is a object which can buy SLAs, i.e. receive *Bid*, *Confirm* and *Reject* messages. The interface is so defined:

```
public interface SLABuyer extends Service
{
        void receive_bid(SLATP_Bid msg);
        void receive_confirm(SLATP_Confirm msg);
        void receive_reject(SLATP_Reject msg);
        SLABuyerProfile get_profile();
}
```

The profile is an indication that a `SLASeller` can ask so as to serve better the buyer. It is so defined (this could be extended):

```
public class SLABuyerProfile
{
        public final boolean send_not_asked_bids;
        public final boolean is_end_user;
        public final boolean is_also_seller;

        public SLABuyerProfile(boolean send_not_asked_bids,
                boolean is_end_user, boolean is_also_seller);
}
```

`send_not_asked_bids` tells the seller that it can also send bids when specific services weren't asked. This is the case in this implementation of `SLATRouter` but not of `SLATUser`.

`is_end_user` tells if it is a user and `is_also_seller` tells if the buyer is also a seller, i.e. a trader.

## 7.5 SLASeller

A `SLASeller` can receive *ask* and *accept* messages. It is also expected that it generates *bid*, *confirm* and and *reject* messages.

The interface is as follows:

```
public interface SLASeller extends Service
{
        void receive_ask(SLATP_Ask ask);
        void receive_accept(SLATP_Accept msg);
        void add_buyer(SLABuyer buyer, SLABuyerProfile profile);
}
```

`add_buyer` is used by buyers to register themselves with this seller so that they do receive bids. Note that this registration process could be implemented in the protocol, but isn't because the associated messaging overhead is so little that it isn't worth simulating.

## 7.6 SLA Map

Some terminology: The *taken slas* are the SLAs which a SLA Router did buy and do represent it's *resources*. *taken bids* are received SLA Bids from other SLA Routers which can become taken slas if they are accepted and confirmed.

*given slas* are sold SLAs and *given bids* are SLA bids sent.

These four SLA groups are stored in a SLA Router as a *SLA Map*. *Given bids* and *given slas* are further divided in function of the SLA Buyer to which they were given (or *sent*).

As said in the previous chapters one of the principal functions of the SLA Routers (also called in this document *SLA Traders*) is creating and proposing SLA bids to customers (a.k.a. *SLA Buyers* or *peers*). Most SLA Router implementations will do this on a one-to-one basis, i.e. they will propose services on the basis of only one taken sla.

A typical SLA Router will examine a *taken sla* and construct from it *given bids* which it will then send to it's neighbours. When a bid is accepted, it will know on which *taken sla* this *given bid* was based and will be able to know if it should confirm or reject it. It will also lower the *residual bandwidth* of that *taken sla*, i.e. the amount of bandwidth of this SLA which can be used for other services to be sold.

A SLA Map is thus a list of *taken slas*, *taken bids*, *given slas* and *given bids* with mappings from *given slas* and *given bids* to *taken slas*.

The `SLAMap` interface is here reported (some public methods are omitted because unimportant):

```
public final class SLAMap
{
        public void put_taken_sla(SLA sla);
        public void put_taken_bid(SLA bid);
        public void put_given_sla(SLA sla);
        public void put_given_bid(SLA bid);

        public SLA get_taken_sla(int peer, Node dest, int ds);
        public SLA get_given_sla(int peer, Node dest, int ds);
        public SLA get_taken_bid(int peer, Node dest, int ds);
        public SLA get_given_bid(int peer, Node dest, int ds);

        public boolean remove_taken_sla(SLA taken);
        public boolean remove_taken_bid(SLA taken);
        public boolean remove_given_sla(SLA given);
        public boolean remove_given_bid(SLA given);

        public Iterator taken_slas_to_dest(Node dest);
        public Iterator taken_bids_to_dest(Node dest);
        public Iterator given_bids_to_dest(Node dest);
        public Iterator given_slas_to_dest(Node dest);

        public Collection given_slas(int peer);
        public Collection given_bids(int peer);
        public Collection taken_slas();
        public Collection taken_bids();

        public void map_given_to_taken(SLA given, SLA taken);
        public SLA get_taken_from_given(int peer, Node dest, int ds);
        public SLA get_taken_from_given(SLA given);

        public SLA move_taken_bid_to_sla(int peer, Node dest, int ds);
        public SLA move_given_bid_to_sla(int peer, Node dest, int ds);

        public SLAMap(Simulator s, int peers);
}
```

You can *store*, *move*, *map* and *search* SLAs in the `SLAMap`.

## 7.7 SLATRouter Base Class

Every implemented SLA Trading Router does subclass the `SLATRouter` base-class. This class does provide the following facilities:

- Database of SLAs using `SLAMap`
- Integrity check routines on the SLAMap
- SLATP Messages Methods
- SLA Numbering
- `SLABuyer` and `SLASeller` interface
- Link bandwidth management
- Traffic Monitoring
- `Router` interface

Some of these are shortly explained in the following subsections.

### 7.7.1 SLA Numbering

Every SLA, *taken* or *given*, must be uniquely identified by the Router so that the protocol can work correctly and so that the SLA database is consistent. In this implementation each SLA is identified (taken and given are two separate groups) with these parameters:

- **peer**, i.e. buyer/seller which received/sent the SLA
- **ds byte**

The DS-Byte is used as a SLA identification. This does mean that unique DS must be given to each created SLA. `DSNumbers` is a class managing the numbering of the DS-Byte. Note that although it is called DS-*Byte*, the number can be higher than 255.

Such a DS-Byte usage is incompatible with the *diffserv* framework. That compatibility was however for the purpose of the simulation unimportant and was more difficult to implement.

### 7.7.2 SLATP Messages Methods

These methods do simplify the construction of SLATP messages. Example of such a function is:

```
protected void send_bid(SLABuyer to, SLA taken_sla, SLA bid);
```

This method will:

- Set the DS-byte of the bid

- Set the `peer` field in the bid `SLA` so that it can be stored in the `SLAMap`
- Put it in the `SLAMap`
- Map it to the `taken_sla`
- Create a *expiration event* which will remove the bid when it has expired
- Send it

### 7.7.3 Link Bandwidth Management

The `link_rbw` array contains the space available for SLAs on the links departing from this node. In the `SLATRouter` two methods are provided to modify this value:

```
protected final boolean reserve_link_for_sla(SLA sla);
protected final void release_link_for_sla(SLA sla);
```

Note that the interface number on which to modify the residual bandwidth is taken from the SLA. Also note that the `link_rbw` aren't equal to the real residual bandwidth which can be accessed with `get_iface(i).get_rbw_notbe()`. The former measures the former takes into account the *reserved bandwidth* while the latter the *real flowing bandwidth* on the link. The relation `link_rbw<rbw_notbe` should be respected though (this is tested in the integrity check routines).

### 7.7.4 Router interface

When the SLATRouter sells a given sla based on a taken sla, it does in fact establish a *route* for packets coming from that given sla. It is a sort of *pipe* from given sla to taken sla. When a flow should be processed, the DS-byte of the flow is used to retrieve the given sla to which the flow belongs. Using `get_taken_from_given` of the SLA Map, the taken sla is retrieved, where the interface number is stored. The DS-Byte is changed to the DS-Byte specified in the taken sla, which will be a given sla for the SLATRouter which sold it and which will do exactly the same to route the flow.

Changing the DS-Byte isn't that simple. Consider a (unicast) flow which starts from a user A and which traverses SLATRouters B and C. B did sell A a SLA to C with the DS-Byte set to 15. B did sell that service based on a SLA to C it does have with DS-Byte 20. What should be made is that A sets the DS-Byte to 15 and forwards it to B, then, B sets the DS-Byte to 20 and forwards it to C. Let's say A and B are on the same node (in other words A is a user by provider B). A sets the DS-Byte of the `UnicastFlow` instance to 15 and B modifies it to 20 without A knowing it! To circumvent this problem, a local *child* (see the `UnicastFlow` documentation in appendix B) is created whenever the DS-Byte is changed, so that when someone forwards the flow, it can assume that nothing won't be changed in it.

### 7.7.5 Traffic Monitoring

The `SLATRouter` base class does also monitor the incoming routed flows to verify that the conditions fixed in the given sla are respected. In case they aren't, the flows are blocked. Note

that this is different from *diffserv* where only the *out of profile* bit would be set, thus changing the service quality.

## 7.8 SLATRouterImpl

Other methods that are common to all trader types implemented were put in a `SLATRouterImpl` class, sub-class of `SLATRouter`. These method are less generic and very sophisticated SLATRouter implementation are likely to not subclass this class. The `SLATRouter` base class should however be usable for every implementation.

Facilities provided by this class are:

- Pricing
- Bids generation in response to Asks
- Confirm or Reject messages in response to Accepts
- Periodic Bids generation
- Neighbour SLAs management

### 7.8.1 Pricing

The price of bids is so calculated in function of destination, bandwidth, maximal delay and volume if the bid (in bits):

$$margin = margin_{min} + bw_{used}/bw_{total} * (margin_{max} - margin_{min})$$
$$price = fixed + margin * cost * volume$$

$bw_{used}$ and $bw_{total}$ are respectively the used and total bandwidth for the group of taken slas for that destination and with that delay constraint. *cost* is the total cost per volume of that group of taken slas. $margin_{min}$ and $margin_{max}$ are configurable and should be so tuned as to make some profit while having good prices. Typical $margin_{min}$ and $margin_{max}$ values are 1.2 and 1.5 respectively. *fixed* is the fixed price and should make the big volume bids more attractive.

### 7.8.2 Confirm Or Reject Messages In Response To Accepts

The `SLATRouter` does call the protected method `send_confirm_for_accept(bid)` which returns if the implementation wants to confirm or reject that accept. The following is checked in this implementation of `send_confirm_for_accept`:

- Verify if the bid to which the buyer refers exists in the SLA Map (it would be an error if it doesn't)
- Verify if the resources are sufficient to guarantee that service
- Verify if the selling price of that service hasn't changed too much

### 7.8.3 Periodic Bids Generation

`SLATRouterImpl` will generate on a periodical basis (every `BASIS_CYCLE_TIME`) bids for buyer and send them to the buyers which want "not asked" bids (see `SLABuyerProfile`).

The bids are so generated:

- A different bid is generated for every *buyer* (peer) and for different bandwidths (64kbps, 128kbps, 256kbps, etc.).
- The bid is constructed, the price set, and if it is *probably interesting* for the buyer, it is sent.

The *probably interesting* check does verify if a similar bid wasn't already sent. Note when an Accept is received for a bid, a new one is automatically generated.

### 7.8.4 Neighbour SLAs Management

To simplify the implementation, services to neighbours are simulated also as taken slas, even though they aren't really received from the neighbours. Every service sold is a service from the provider which sells the service, going to that provider is responsibility of the buyer. For that reason it doesn't make sense for providers to send bids for themselves to neighbours ...

A provider wants to sell services to it's neighbours though, and that services take bandwidth from the links.

This is easily done by simulating received *Bids* of various sizes to neighbours, which are automatically re-generated when they are *Accepted* by the provider. In other words, the SLATRouter implementation does treat *Neighbour Bids* as normal bids. The only difference is that *Accept* messages are treated in a specific way (no *Confirm* should be sent!).

## 7.9 SLATRouter Implementations

What remains to be defined for a functioning SLATRouter is mainly what bids to buy. The `bid_received` method is called when a bid is received and it is expected that the SLATRouter implementation do define this method such as to be informed when bids arrive. The implementation can consult the SLA Map (which is filled by the base classes) and decide from that too.

## 7.10 Greedy Trader

The `SLATRouterGreedy` is a very simple implementation. The full implementation is here reported:

```
package flowsim.slatr;
import  flowsim.core.*;
import  java.util.Random;

public class SLATRouterGreedy extends SLATRouterImpl
{
```

```
        final static long ACCEPT_MARGIN = 2*BASIS_CYCLE_TIME;

        protected void bid_received(SLA bid) {
                if(bid.expiration<s.now()+ACCEPT_MARGIN) return;
                if(bid.iface<0) return;
                if(bid.cost>budget) return;

                if(has_link_space(bid)) {
                        accept_bid(bid);
                }
        }

        public SLATRouterGreedy(Simulator s, Node node, int port,
                int local_peers, long budget, long fixed_costs,
                Random random)
        {
                super(s, node, port, local_peers, budget,
                        fixed_costs, random);
        }
}
```

This SLATRouter does try to accept every bid it receives. It will stop when the budget isn't enough or when the link capacity is used.

## 7.11 Trendy Trader

The `SLATRouterTrendy` does only buy bids upon their reception if they are SLA to a new destination to ensure connectivity. On a periodical basis, the taken slas are scanned and, if the residual bandwidth is lower than a trigger value, a new SLA to that destination is bought. To buy a SLA, the list of bids is scanned and the most convenient (in term of price per volume) is accepted.

## 7.12 Profitable Trader

`SLATRouterProfitable` is similar to `SLATRouterTrendy`. The principal difference is that new SLAs are bought because of low residual bandwidth if that service is considered *profitable*.

*Profitability* for a bid is so determined: The average price of the sold given slas which resulted from buying the last taken sla is calculated. If this bid was bought and the expected amount of it (`TAKENSLAS_EXPECTED_RBWBW`) was sold at that price, would the SLATRouter make profit?

## 7.13 Lazy Trader

`SLATRouterLazy` doesn't buy anything automatically. It does only buy new SLAs upon reception of *Asks*. In such a case, it does buy a bid if what was asked is already there, or asks it itself to it's neighbours. If many `SLATRouterLazy` are used, Asks are flooded in the network. This is inefficient for messaging overhead and setup time but guarantees that only what is needed is bought.

Note that to lower the messaging overhead, SLAs larger than what is asked are bought or asked so that other following Asks can perhaps be directly served. If *real laziness* is wanted, this behaviour can be changed though.

## 7.14 Users

The `User` interface is so defined:

```
interface User extends Service
{
        long get_budget();
        long get_budget_incr();
        void add_to_budget(long money);
        void remove_from_budget(long money);
        void print_stats();
        float get_max_utility();
        float get_utility();
}
```

Every `User` has a `budget` which is periodically increased such as to simulate money spending in the system. They also have an accumulated *maximal utility* and *received utility* so that their satisfaction can be measured.

### 7.14.1 Asks generation

When a user wants a service, it normally “Asks” it (see 5.4.1) to it’s service provider, which will then generate a bid of service susceptible to be accepted by the user.

Consider the scenario where a user wants a service $A$ with a minimal bandwidth $A_{bw}$ and a maximal delay $A_{delay}$. To get a set of bids upon which the bid selection algorithm chooses the best one, there are two possibilities:

- One Ask is sent and a group of probable bids is generated by the server according to server’s estimation of user preference.
- More Asks are sent and the server generates one Bid for each of them.

The first option is impractical because the server can’t know exactly the user’s preference, willingness to pay, etc. (and would be probably bad for the user if it did), which makes the generated bids sub-optimal. The second option however let’s the user decide what bids it does want to make it selection upon and is certainly more flexible.

Figure 7.1 shows the Asks generated for a service requirement of a user. Every Ask specifies the minimal bandwidth and maximal delay (more parameters are specified, but let’s focus on these two). For every Ask, the server can choose the bid which satisfies these conditions and which costs the less (the regions in grey are what the server is able to generate).

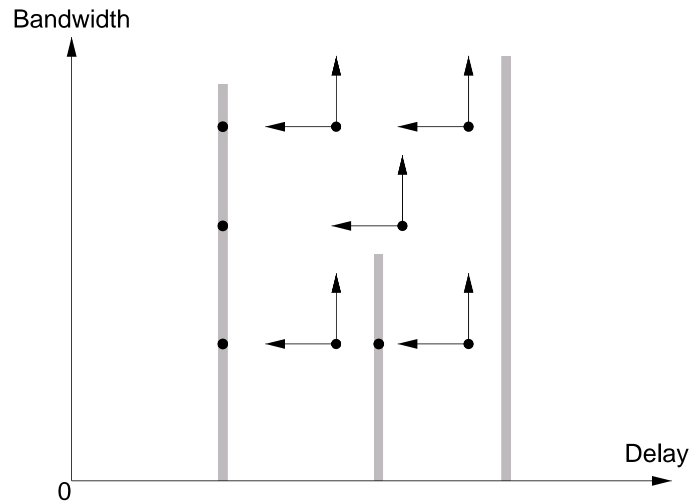



Figure 7.1: Asks generated by the user

## 7.15 Voice User

The Voice User is a user which wants to establish 16kbps voice connections with the utility figures discussed in chapter 6. Voice Users are implemented in two different ways, depending on the fact if they are users of the SLA Trading System or of the Distance Vector Routing System used as comparison. The former is `SLATUserVoice` and the latter `UserVoice`.

`UserVoice` simply starts with Poisson distribution UnicastFlows with 16kbps and does monitor the blocking of it. `SLATUserVoice` are a bit more complicated because they first try to by the service, and then start the flow setting the DS-Byte accordingly.

# Chapter 8

# Results

## 8.1 Introduction

As said in the previous chapter, goal of the simulation was to show that the SLAT Routing system does function correctly for routing and that it is better than other system, such as Distance-Vector for QoS constrained flows.

To measure the performance of the implementation, several scenarios have been simulated to stress each time particular properties which should be analysed.

It is also useful to compare the results to some other known system. Where such a comparison makes sense the same scenario was simulated using what is commonly used in the today's Internet: shortest-path routing and integrated-services's reservations over these paths.

## 8.2 Simulation limitations

Because of resources limitations and complexity, these simulations are an approximation of multiple ISPs across the globe which use SLA Trading based Routing. This section will examine these limitations, to help evaluate the results of the simulations. These limitations will be further discussed in the conclusions chapter.

### 8.2.1 Scale of simulation

The SLAT-Routing system is thought to work with very big aggregate flows between country-wide providers. This very high number of flows in each direction of a provider, make it possible for him to evaluate accurately what the future demands will be and thus buying exactly what is needed.

In this simulation, only twelve providers are simulated. A major problem is the low quantity of users on each provider. Consequence of this limitation is that the SLAT-Routers in the simulation will make bad predictions on the future needs and thus buy unnecessary services or not enough of them. Said in other words, the level of aggregation is too small and the statistical behaviour is thus less predictable.

### 8.2.2 Simulation Period

The variation of the needed bandwidth of the providers is very slow and the SLA Trading Protocol is designed to sell services of duration from minutes to days.

Because of the simulation run-time, which would become prohibitively high, simulations in the order of seconds upto minutes were made. Short-term services are sold and the overhead is thus much higher.

### 8.2.3 Randomness

In the simulations, the arrival of traffic flows was generated randomly with Poisson distribution. Sources and destinations, if not fixed, were also chosen randomly. In the Internet reality the traffic patterns are, if seen at the aggregated-flows level, non-uniform, because of common user preferences and popular services. This macroscopic behaviour is very difficult to simulate, but is however what the SLAT-Routing proposal tries to optimise best. It was tried to show the behaviour for such cases for example in the load-balancing experiment with non-uniform scenarios.

### 8.2.4 Intelligence of SLAT-Routers

Because of the mentioned limitations and of the complexity of the problem, it was also difficult to design SLAT-Routers which make good predictions on the future needs to decide what to buy. This is a typical economic problem which couldn't be resolved in a very efficient way in the scope of this diploma thesis.

## 8.3 Load balancing

### 8.3.1 Motivation

The load balancing property of the SLAT routing technique is based on the selection of alternate routes when there is congestion on the previously selected route to a given destination.

The mechanism by which this load balancing is automatically done was already shown with an example in section 4.7. The neighbour in the path of the congestion will have very high prices or make no more bids at all and thus, if additional bandwidth is needed to a destination past this congestion, a bid of one of the neighbours, which normally cost more, will be taken.

### 8.3.2 Scenario

The very heavily connected grid topology of figure 8.1 was taken, so that there are many alternate routes with small hop-count variations. To show the load balancing effect, a voice user (see section 7.15) with very high demands is simulated on node 0 and wants to make calls always to node 15. No other traffic than the voice traffic of this user was generated. Every link has a capacity of 10 Mbit and the total user bandwidth demand is 20 Mbit (made of in the average 1250 flows of 16 kbit each).

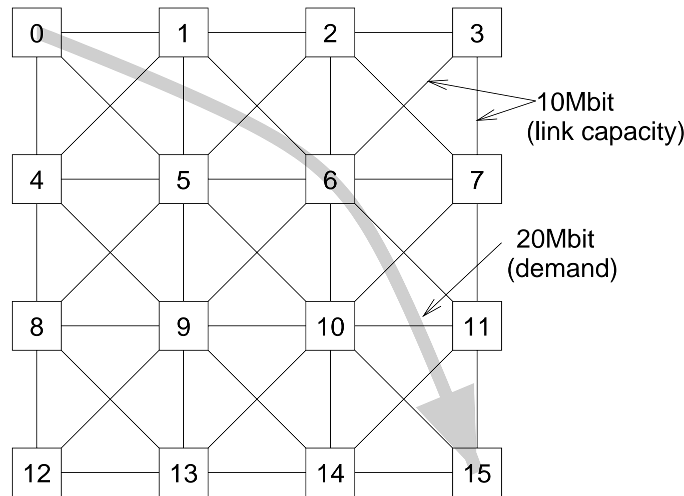



Figure 8.1: Load Balancing experiment

As a comparison, the same experiment was done with normal DV routing (shortest-path) and RSVP-like reservations.

### 8.3.3 Measurement: Accepted Calls

Figure 8.2 shows the *users call demand*, *SLAT call success rate* and *DV call success rate*.

The curve at about 500 calls/s is the *demand curve*, i.e. what the users request from the network. The *DV-curve* begins by following the *demand curve*, because at the beginning no resources are used, and thus every resource reservation (à la RSVP) is successful. After about two seconds, the links bandwidths begin to be fully used and the success rate decreases rapidly. Distance-Vector routing chooses only one route, the shortest path, and every flow will follow it. This means that the maximal traffic bandwidth which can be transported is the bandwidth of that route, i.e. 10 Mbps. The demand is 20 Mbps and it thus follows that only half of the request can be served as shown in the DV-curve.

The *SLAT-curve* shows a *slower setup time* but is able to serve almost all 20 Mbps traffic. The setup time follows from the fact that SLAs must be bought by the Traders to provision the resources such as to serve the users. The `SLATRouterProfitable` implementation was used (see chapter 7), which does analyse the *trends* of traffic demands and buys accordingly. This process does obviously require time but it is clear in this example that it does produce better results than DV-Routing. Note that it is a system for *providers* and for *large flow aggregates* which do have a slow changing behaviour and which the provider will be able to forecast much better.

### 8.3.4 Measurement: Bandwidth Usage on Links from Node 0

Figure 8.3 shows the bandwidth usage of the links going from node 0 to it's neighbours.

Link to node 5 is the link on the *shortest-path*. On the shortest path, there are less traders which do inflate the price for that service such as to make profit and therefore the price is lower. This can be seen at time 0: all the traffic goes through the shortest path because the trader on node 0 will only buy SLAs from node 5.

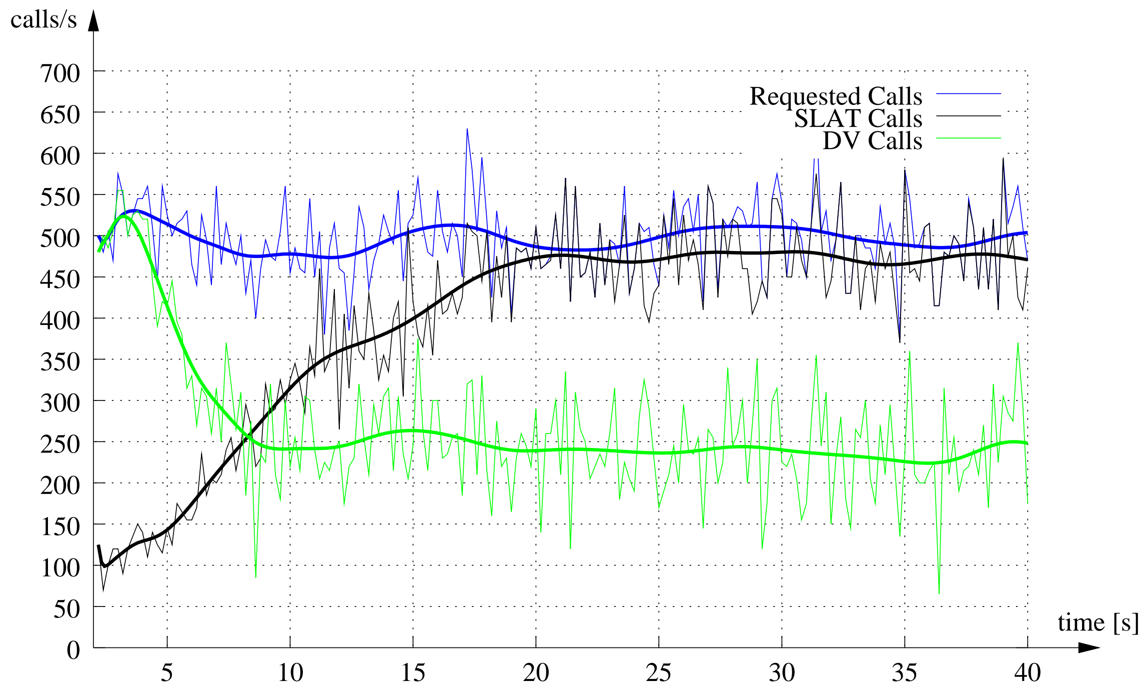



Figure 8.2: Load Balancing Experiment: Call Success Rate

The prices of the traders on the shortest path will go up as the resources are more used. They will go up until the price of the *almost-shortest-paths*, i.e. in this case the paths with one hop-count more, will become lower than the price on the shortest-path. The trader on node 0 will begin to buy SLAs also from node 4 and 1 and traffic will begin to flow on the corresponding links, as seen in the graph at time about 5 s.

The two paths going though node 1 and node 4 have the same hop-count and the price behaviour in function of traffic should be about the same. The consequence is that the SLA bought by trader on node 0 from the traders on node 1 and 4 should be balanced and the traffic should also follow this balance. There is however a little difference of about 128kbps in the graph.

That difference can be explained by the fact that the trader on node 1 did buy one more SLA from a neighbour of 128kbps and maintains that advantage. This is the result of the randomness of the system: node 1 did sell first it's service, thus did see first the need for other services, and so on. What is important is that it is only one SLA difference, i.e. the load is effectively balanced within the SLA granularity.

### 8.3.5 Nam Display

Figure 8.4 shows the output of *nam* showing the traffic at about 20 seconds from the start. Note the three main independent paths. It wouldn't make sense for an alternate path to merge back in the main (shortest) path, because the bottleneck problem would be the same as taking only one route. Note that the drawn packets aren't real packets but only an indication of the usage of the links by the flows.

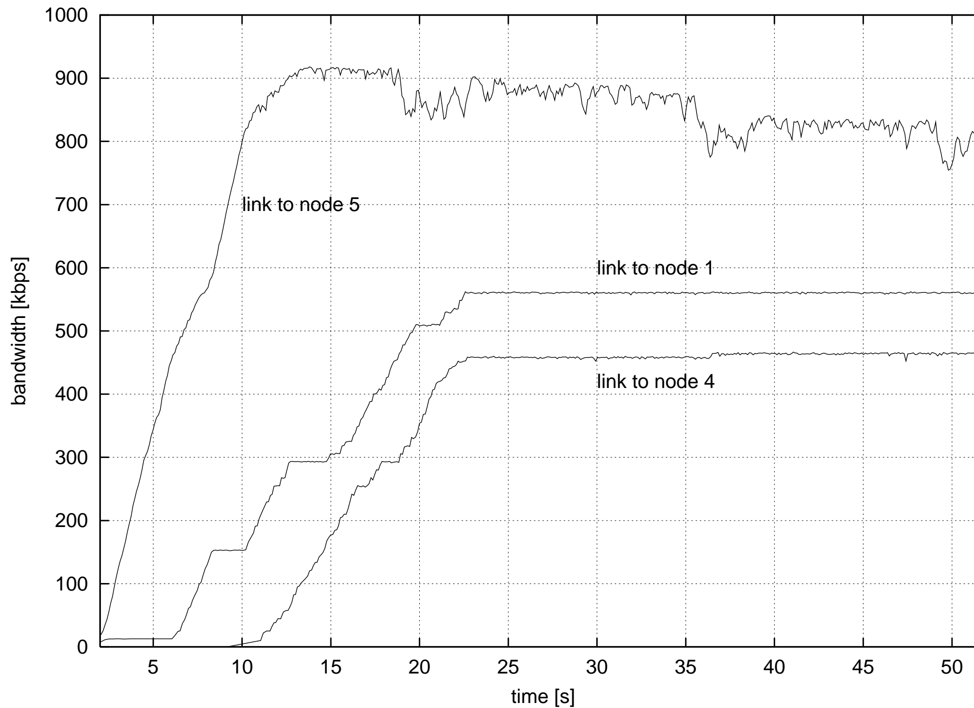



Figure 8.3: Load Balancing Experiment: Bandwidth on Links of Node 0

### 8.3.6 Results Comments

This experiment showed a *weighted load balancing* behaviour: the load is balanced on the links according to their network-costs. Using a local profit optimisation algorithm, a very good network resources usage optimisation and a much better utility for users than with shortest-path routing is attained.

## 8.4 User Competition

### 8.4.1 Motivation

When some resources cost very much or are limited, only some users can use them. The selection of the users which can access these resources is made in the SLA Trading System automatically by *user competition*. See section 4.9 for a more detailed explanation of this behaviour.

This experiment should show that the user competition is effectively done and that the users with the highest willingness-to-pay for the service are admitted.

### 8.4.2 Scenario

The very simple topology of figure 8.5 is taken. The links from A to C constitute the limited resources. On node A there are three users: one which is willing to pay little for the service (User 3), one willing to pay somewhat more (User 2) and the third willing to pay very much for it (User 1).

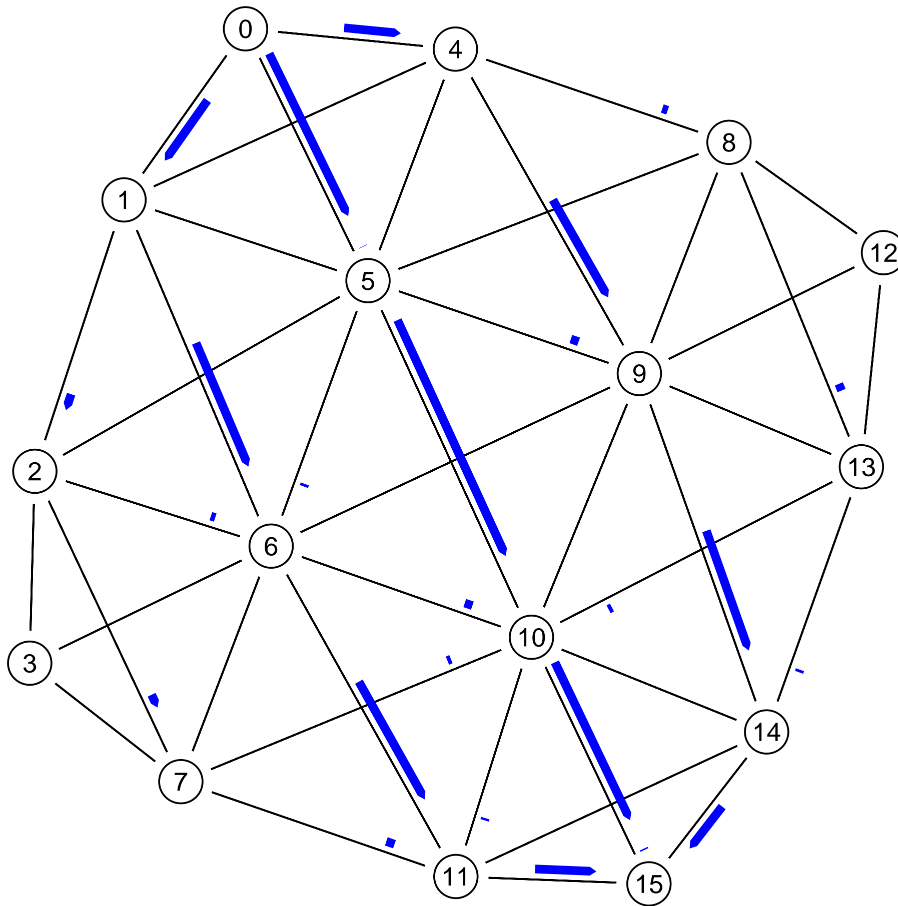



Figure 8.4: Load Balancing Experiment: Nam Output

Every user wants in total 5 Mbps of bandwidth to destination C and a selection is clearly needed, because it does make for the three users 15 Mbps, while the links are only 10 Mbps. Every user is started at the same time.

### 8.4.3 Measurement: Utility

In figure 8.6 the maximal utility receivable by one user (if it could transfer all it's flows) is plotted along with the perceived utility by each user.

At the beginning, the links are unused and thus there is space for every request. You can see that every user receives about the same service from the net. At time about 4 s, the price for the service, because of resource usage has become too high for User 3 which will rapidly give up buying services.

What is visible is that User 1, which is the user willing to pay more for the service, will receive almost all the resources it does ask, User 2 follows (and takes the rest) and User 3 doesn't take anything.

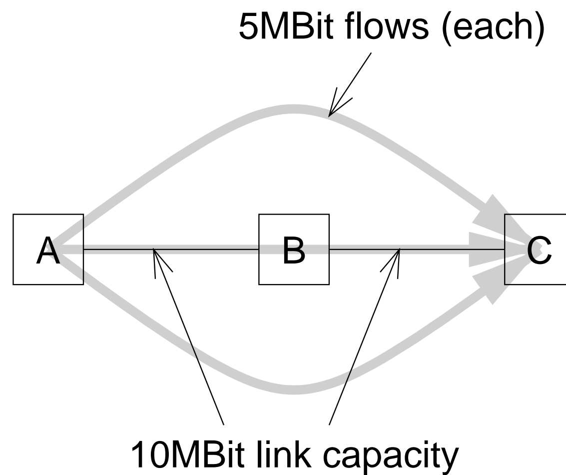



Figure 8.5: User Competition Experiment Topology

### 8.4.4 Results Comments

From User Competition, it is clear in this experiment that when the resources become scarce, only the users willing to pay more will be allowed to use them.

## 8.5 Trader Implementations Comparison

### 8.5.1 Motivation

In this experiment, a comparison between the SLATRouter implementations, as described in chapter 7, is made. The question that it should answer is "how much do implementations make a difference for the SLA Trading System efficiency ?"

### 8.5.2 Scenario

The same scenario as in the *Load Balancing* experiment was used: The very heavily connected grid topology of figure 8.1 and the voice user with high bandwidth demands. The only difference is that the demand is somewhat lower: 5 Mbps instead of 20 Mbps.

On every node except node 5 there is a `SLATRouterProfitable`. On node 5 the SLATRouter implementation is variated: Greedy, Trendy, Profitable and Lazy implementations are tested.

Node 5 was chosen because it is on the shortest-path of the demand and it's implementation should thus make a big difference for the route selection.

Note that in such an environment, the Lazy trader won't buy anything because it won't receive any *Ask* message. It was inserted in the experiment as a comparison to a *passive router*, i.e. a router which doesn't make anything. The result of such a passive router is that alternate routes should be selected.

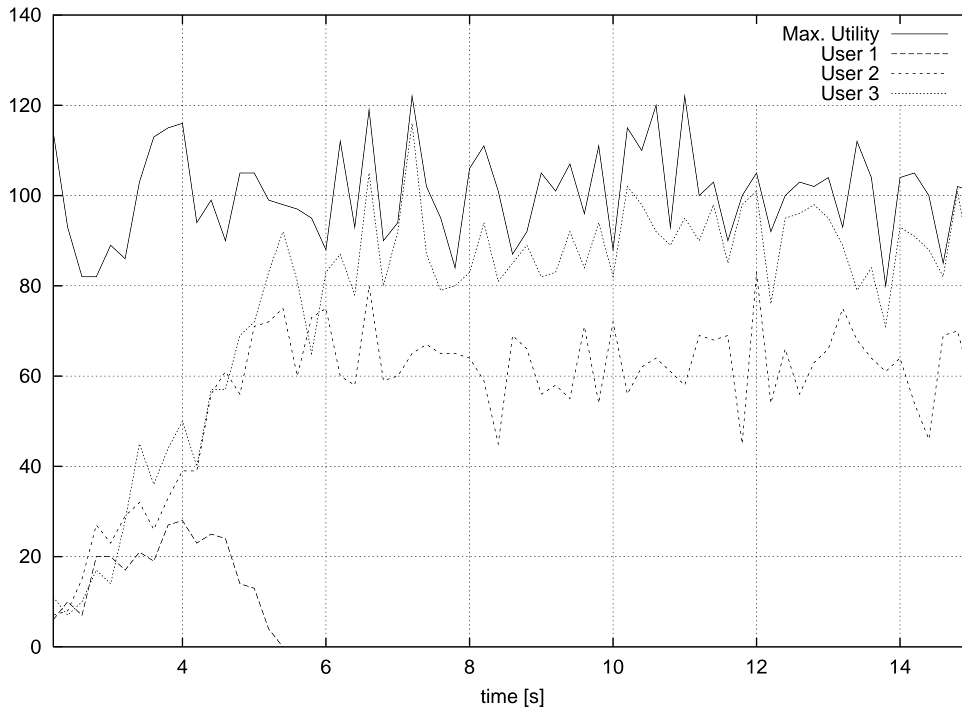



Figure 8.6: User Competition Experiment: Utility

### 8.5.3 Measurement: Trader Budget

The first measurement is how much the trader at node 5 does spend or gain. If it does make good predictions and decisions on which services to buy, it should make profit. If it does make inefficient decisions, it will spend more as it gains and thus will loose money. Figure 8.7 shows the budget of the four router implementations.

The *Lazy* doesn't buy or sell anything and thus it's budget remains at what it was initially. Greedy makes very bad decisions, looses money immediately and is thus in little time eliminated from the market as a service provider. *Trader* makes also bad decisions although much better than *Greedy*. Note that every neighbour is a *Profitable* router, which is more clever, and it is thus expected that it looses some money. The *Profitable* router does make profit. It is clear that money from the user flows to him, one of the main service providers for that service (going from node 0 to node 15).

Note the loss of money at time about 4.1 seconds: at that time, the provider can serve the user fully, i.e. the resources that it did buy are all used and no more are required. The Profitable trader will always try to keep it's resources used at 70 percent and what happens at time 4.1 is that it does buy additional resources to go from 100 percent usage to the wanted 70 percent usage.

### 8.5.4 Measurement: User Utility

The second measurement is the utility that the user at node 0 receives from the network in function of the implementation of the trader at node 5.

What is interesting to measure is how time it does take to setup the resources such as to satisfy the user demands. The setup time is summarised in the following table:

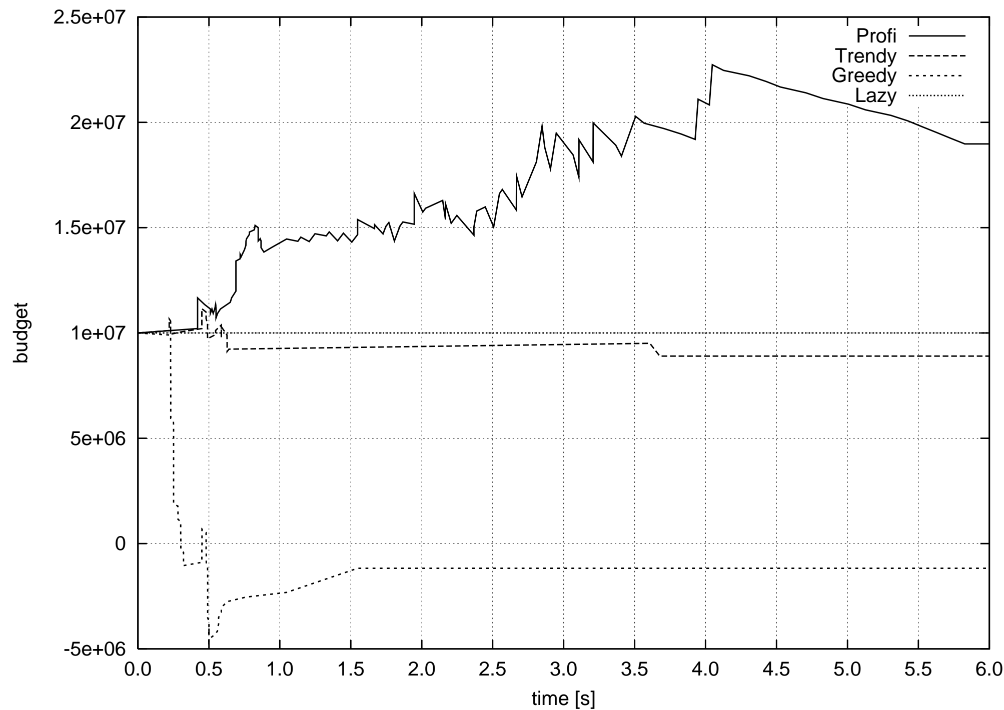



Figure 8.7: User Competition Experiment: Budget

| **Trader** | **Setup Time** |
|---|---|
| Profitable | 2.7 s |
| Trendy | 3.6 s |
| Greedy | 4.4 s |
| Lazy | 5.2 s |

The *Profitable* Trader is clearly the better implementation in this case, which is also reflected by it's high gains. *Trendy* follows, because it makes sensed decisions based on service usage although not always good. *Greedy* makes very bad decisions on the services to buy, and does impose an unneeded network load, but the fact that it buys something does make that this route can also be used and is better of *Lazy* which doesn't buy anything at all and which does require the setup of alternate routes though node 4 and 1.

### 8.5.5 Results Comments

The profit of an implementation should reflect it's efficiency in terms of utility of the offered service and resources usage. It is demonstrated that a trader which does make bad decisions, for example the *Greedy* trader, is eliminated from the services market. There is also a direct relation from efficiency of a single trader to utility of a external user, but it is shown that the network can adapt itself very well to every type of implementation by for example finding alternate routes to circumvent bad implementations.

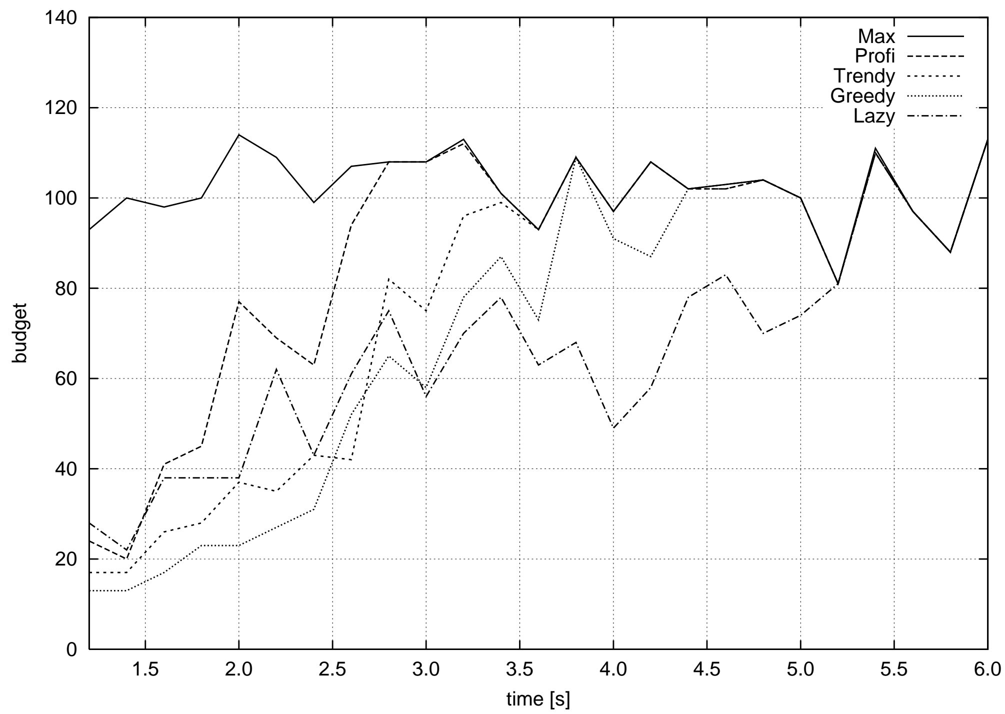



Figure 8.8: User Competition Experiment: Utility

# Chapter 9

# Conclusions

## 9.1 Advantages

### Price Per Utility Minimisation

The most important advantage of this system is it's capacity to minimise the *price* of services while maximising their *utility*. The *providers competition* does select the providers which do minimise their cost while providing attractive services to the customers.

### Quality of Service Guarantees

With this system, QoS guarantees can be made by the providers in a very straightforward way, because the QoS specifications are part of the system and not an addition to it.

Also, because of the *customers competition*, the services with the higher QoS demands will be given to the customers willing to pay more for that service, thus making a selection on the customer which can use scarce or expensive resources.

### Route Selection According To Local Preferences

Every trader does make the selection on which services to buy independently. This is made according to profitability but some provider preferences or policies can be enforced without requiring any modification in the other traders.

### No Global Protocol Agreement Necessary

The purely *bilateral nature* of the service level agreements does imply a protocol independence for the establishment of agreements. This is a very big advantage, because it does mean that no global agreement on a protocol for the signing of SLAs must be attained before the effective deployment of the system. Another consequence is the easy transition to new protocols which are more compact or more flexible.

#### Income Distribution Where Services Are Used

This system does provide a distribution of money to the providers that do provide services. They provide services because they have sold them in a pay-per-use manner. On the other side, customers (which can also be providers) with high demands for the network will pay more than small customers.

## 9.2 Problems

#### Good SLA Trading Routers Are Difficult To Design

It is difficult to implement traders which do the right decisions on which services to buy and which choose good prices for the bids. The future market demands must evaluated and forecasted so that the resources will be there when they will be there. Opposing this resources availability goal is the cost of unused resources which must be kept to a minimum.

Since better implemented traders are rewarded by better profitability, there is a very strong incentive for the providers to build the most clever traders as possible, and it can be safely assumed that very good traders will be designed.

#### Messaging Overhead Is Difficult To Keep Small

The providers will want to given to their customers the most precise and diverse service bids as possible trying to serve accurately the customer demands. The problem is that these bids do involve a messaging overhead which reduces the availability of the link's bandwidth as service to be sold.

Because of the protocol independence the providers will be free to build very compact protocol implementation which reduce the messaging overhead to the minimum. As in the previous problem, this problem does also represent a gain of money if solved in a better way than the others and it is thus expected that much research will be made by the providers.

## 9.3 Review of Goals

#### Loop-Free, Convergent Routing

If a provider can make the hard guarantee on the maximal delay of delivery packets, it can also be assured that the path that the packets will follow is loop free. In fact, because of these hard guarantees, it is completely of no importance for a customer the path that the packets will follow: only the compliance to the QoS specification does matter to him. This goal was fully reached.

#### Admission Control Based On Available Resources *and* Pricing

Every SLA proposal does (in the commercial provider case) come with a price. The admission control based on pricing is made by the selection of which customer do take the bids at that price. This price can also be dependent on the amount of remaining resources thus making an admission control system based also on resources availability. This goal was also reached.

#### Services Differentiation

The services differentiation goal is provided by the *diffserv* framework. Goal reached.

#### Quality Of Service Assurances

This does also come from *diffserv* and can be done in two basic ways: using *Allocated Capacity* (statistical assurance) and *Premium Service* (strict assurance). Goal reached.

#### User Utility Maximisation

Because of competition, the customer will receive many proposals of service from which it can choose the best one, i.e. the one with the highest utility for him. If there are customers which want a service, but no provider to sell it, a new provider is encouraged to setup the required resources and provide that service thus gaining money.

#### Network Load Minimisation

Also because of competition, the provider will want to render it's proposals the most attractive as possible. This is done by using the available resources cleverly and buying what is needed and no more, which would be a loss for the provider. Global network load is thus kept to a minimum, because more load is directly associated with more costs for the providers.

This and the previous goal were solved in an efficient and good way, solving it the *best* way is a NP-complete problem.

#### Easy Deployment In Existing Internet

The already mentioned protocol independence and unnecessary global agreement on a standard does make this system easily deployable. A provider could for example at the beginning provide SLA trading services only to it's neighbours.

## 9.4 Future Work

#### Diffserv-Style Services Classification

For compatibility reasons, the *diffserv* classes should be reused. This is for the moment a problem because they are actively being defined by the *diffserv working group*. The implementation (see chapter 7) uses very simple classes (delay bound and not delay bound) and a ds-byte numbering scheme for the SLA identification which is incompatible with diffserv.

#### Assured-Service (Statistical Guarantees, Over-booking)

*Premium Service* only was implemented (see chapter 7), i.e. no over-commitment is made. It is highly inefficient for the network usage to use only that type of service. *Assured Service*, which

was thought to be too complicated to implement in the scope of this diploma thesis, will certainly be a big improvement on the quantity of service that can be delivered.

### Destination-Aggregation

The current system does treat each destination independently. This could become a problem for large networks because the messaging overhead and database of SLAs would be too big. This problem can be partially solved with the usage of *Assured Service* along with *Destination-Aggregation*. Assured Service does permit to make *estimated delivery guarantees* which could be made for groups of destinations. It is certainly more difficult to make good *strict guarantees* for group of destinations, because the destination in that group with the worse QoS metrics would determine the overall metrics of the group.

### Better (More Compact) SLA Trading Protocol

Reducing the Bids overhead is also possible by reducing the SLA Trading Protocol overhead. This is possible with the use of pricing functions, multiple bids aggregation and delta messages (see section 5.5).

# Appendix A

# Project Description

## Diplomarbeit für Herrn David Schweikert

| | |
|---|---|
| **Aufgabenstellung**: | George Fankhauser |
| **Thema**: | QoS Routing and Pricing in Large Scale Internetworks |
| **Beginn der Arbeit**: | 17. November 1998 |
| **Abgabetermin**: | 27. März 1999 (arbeitsfreie Weihnacht) |
| **Betreuung**: | George Fankhauser, Burkhard Stiller |
| **Arbeitsplatz**: | ETZ G69 |
| **Umgebung**: | ns-2, PC/Linux, Sun/Solaris |

## A.1 Introduction

The Internet has grown at a very fast pace over the recent years. In parallel more and new services were introduced to this network. Originally designed as a data packet network with best-effort packet forwarding capabilities, services with different application and service characteristics, such as streaming voice and video, known from circuit-switched networks, were experimentally introduced to the Internet.

Due to its best-effort approach the Internet cannot handle such real-time services, when overload situations (congestion) occur at network nodes. To remedy this problem and to enable a multi-service capable Internet, two approaches are currently researched:

- Adaptive applications in cooperation of queueing mechanisms: Here, the original characteristics of the Internet should be restored as much as possible but the applications requesting high-end services should respond to quality-of-service (QoS) changes in the network. Moreover, such applications can tag (mark with a label) packets in order to give them priority etc.
- Another approach resembles more the traditional telephone network and uses signalling along data paths to reserve resources (link bandwidth, buffers). In order to be compatible with the Internet design goals a special form of signalling, called soft-state, was introduced. Also, in addition to strictly guaranteed serviced, more relaxed forms of service (e.g. controlled load) have been introduced.

These two extreme approaches can also be operated in conjunction and a current field of research (diffserv) investigates the suitability and interworking.

In this work, we focus on the approach using reservations. Although proposals for resource reservation systems are quite old and numerous, they still lack some important functionality and critical questions still remain unanswered. For example, today's most popular resource reservation protocol (RSVP), solves the following problems:

- It transports information about resource reservation requests by signalling flowspecs.
- It offers some basic security enhancements (like integrity and authentication)
- It is a flexible and extensible protocol that can transport any kind of signalling information

  Some of the points not yet solved are listed here:

- There is currently no policy applied to resource reservations. Whenever the resource is available you get it. One of the most interesting policies to be applied is pricing; for an initial approach see [3].
- Resource reservations have to be made immediately. There is no way to make long-term or in-advance reservations.
- Resource reservations are made along a path determined by standard routing protocols that are optimized for best-effort packet forwarding. We suspect this approach to be suboptimal for several reasons (overloaded shortest paths, no interactions between reservations and routing, etc) [1].

The last point of this (incomplete) list is the major target of this work. First we will have to demonstrate how resource reservations and routing interact. For that reason convincing scenarios have to be simulated, demonstrating standard topologies of ISPs with access and backbone networks. Moreover, it is difficult to predict how users will use their applications and thus the network. Here we have to investigate a whole palette of configurations and traffic mixes to get an idea how routing reacts.

Once we have established a model of the status quo, modifications and improvements can be considered. Two possible approaches focus on the modification of the RSVP protocol by adding QoS-routing information to the signalling messages or on the modification and interaction of standard routing protocols such as OSPF and BGP to include information about reservation state. In any way, routing can be summarized as a cycle of the following actions:

- collection of information about topology and load
- re-calculation of new routing decisions
- redistribution of this newly aquired knowledge to other routers.

## A.2 Environment

As introduced, the target of this work are currently deployed Internet configurations. Since the Internets shape constantly changes we have to work on reasonable abstractions that are simple

enough but show the basic problems. Also for testing, we can work on specialized and in the literature well-known topologies, like 4 by 4 full meshes, etc.

The basic elements of such a setup are pictured in the diagram above: Hosts (end-systems) and routers, traffic sources and sinks, links between nodes (nodes are any of hosts or routers) and 'agents' which may operate on routing or signalling protocol or affect pricing or other policy decisions.

### A.2.1 Network Flows and Reservations

A network flow is simply defined as a stream of packets with the same header information. In our context, we usually require the same source/destination addresses and ports plus the same protocol identifier (i.e. UDP or TCP). On such flows, implicit or explicit reservations can be made. An implicit reservation is made locally whenever certain conditions hold (i.e., 4 packets of the same flow within 500 ms); this mechanism is called flow detector. Explicit reservations are initiated by hosts using signalling protocols. For example, RSVP defines such a signalling protocol where senders advertise flows and receivers may reserve resources on such data streams. Flows are usually simplex flows, for duplex communication we need two flows, such as depicted between hosts A and B with the two flows 1 and 2.

### A.2.2 Routing with Multiple Objectives

Basic routing has been defined as collection of routing information, calculation of paths and redistribution of such information. In traditional routing protocols there is a single metric defined (distance, delay, etc.) and used to calculate a shortest path using an algorithms such as Dijkstra's. Note that this calculation can also happen in a distributed manner.

When additional metrics are introduced the problem becomes np-complete. Therefore, large networks feature usually such additional metrics (e.g., cost, bandwidth, and delay) but do not perform calculations on all parameters. They rather use heuristics to combine metrics to speed up calculation.

An important Internet routing protocol that supports multiple metrics and is even extensible with policy decisions is BGP (Border Gateway Protocol).

### A.2.3 Working Environment: ns-2 Simulation Platform

The network simulator 2 (ns-2) is an event-driven simulator that specializes for Internet protocol applications. Its most extensive use was for the development of various TCP enhancements. Recently, other Internet protocols were added to ns-2 (such as RSVP, link state routing, real-time transport protocols, etc.)

The following gives some pointers on how to get started with ns-2. There is an ns-2 homepage[1] and mailing list, please subscribe to both lists!

A 'must read' is Marc Greis' ns-2 tutorial[2].

Examples can be found in all ns-2 installations in subdirectories tcl/ex and tcl/ex/rsvp

[1] `http://www-mash.cs.berkeley.edu/ns/`

[2] `http://titan.cs.uni-bonn.de/~greis/ns/ns.html`

Some facts what ns-2 is:

Protocols supported: IP, UDP, TCP (many implementations), RSVP, LAN-Protocols (IEEE MACs), DV Routing, LS Routing, RSVP, IP Multicast, Mobile IP, RTP, RTCP, HTTP

Many applications and/or traffic generators are already supported (e.g., ftp, telnet, Inria Video System, Video audio tool VAT), simple CBR sources, HTTP/1.0 traffic generator for ns-2 as an example for complex traffic sources [3].

Router functionality: Many queueing algorithms (class-based, FIFO, fair, weighted-fair, statistical-fair, deficit round robin, random early drop (RED). Link properties: error and delay models.

#### More Tools and Graphics

There is basic support for monitoring links, nodes, and queues. A library of statistical functions is available (written in oTcl). A graphical tool called nam (Network Animator) can be used to play back trace files and animate protocol behavor down to single packets.

For documentation we will need plotting tools like gnuplot. This program supports plotting trace data in tabular form (e.g. a textfile of two columns of data with the time in the first and the queueing delay at some point in the second). Furthermore, functions can be plotted (e.g. to compare measurements to the complexity measure of an algorithm by plotting the datafile and the function n*log(n)).

#### Practical Tips

Simulation of complex models is slow. Distribute your tasks, think ahead and use multiple machines at the same time (e.g. with the help of rsh). Fast machines can be found across the TIK network. If you use other CPUs do it mainly at night and nice your ns process (by running nice ns my-script.tcl or doing a renice 10 process-id). In addition, ask the main users of the machines for permission (do a finger @machine — grep -i console). Fast machines are Ultra-30 on the kom- and tik-subnetworks: kom21, kom26, kom31, kom32; tik37 .. tik40.

Compilation and linkage is mostly I/O-bound and works best where your files reside (minimizes network traffic). A df $HOME shows where your home directory is attached.

## A.3 Tasks

The objective of this work is to compare traditional routing used for RSVP calls with an enhanced call setup protocol that modifies routing decisions based on resource availability (or prices). Since there are already QoS-routing enhancements made to OSPF (QOSPF) using the widest-shortest path algorithm, and BGP might support improvements, those protocols have to be evaluated with respect to improvements for call routing too. Among these traditional or enhanced protocols this project focuses on the exterior gateway protocols (i.e. BGP).

[3] `http://http.cs.berkeley.edu/~tomh/wwwtraffic.html`

### A.3.1 Theory, Overview and Related Work

This part of the project serves as a foundation in theory and summarizes related work in relevant fields. A good overview of Internet routing is given in [2]. Current QoS routing problems and related work is described in [1]; it outlines also why QoS routing (as best-effort routing) may profit from hierarchical implementations.

#### Overview on Internet Routing and QoS Routing

This part has to grasp the essence of today's Internet routing algorithms, their hierarchical layout and possible attempts to bandwidth management or optimization.

It has to include a short routing classification including all major criteria (on-demand vs. pre-computation, centralized vs. distributed, complexity/scalability (messages, memory, computation), metrics used, basic algorithms used, etc.).

Most practical solutions for global networks use a hierarchical approach to routing. This fact has to be discussed separately.

Examples of relevant routing work include:

- Path QoS Collection [11]
- Selective Probing QoS routing framework [5]
- (Dynamic) Alternative Routing
- Widest-shortest and Shortest-widest path algorithms [8], [6]

As a different but related aspect, routing dynamics and quantifications of message overhead of best-effort routing protocols have been investigated in practice (i.e. work by V. Paxson, supported by lots of measurements).

#### Overview of Exterior Gateway Routing Protocols

This part of the project takes a closer look at the few proposals dealing with interdomain QoS routing. Where relevant, related work on ATM QoS routing should be reviewed (e.g. [9] or [10]).

A first step to quantification of QoS-based routing protocols is given in [7].

Another important and difficult point is policy routing: arbitrary policies invalidate usually the desired optimization goals (high utilization, low cost). As an example of such incentive incompatible policies, Earliest Exit should be mentioned [12].

### A.3.2 Design of an Interdomain Path Selection Algorithm and Protocol

As described in [1] traditional best-effort datagram routing focuses on connectivity, operates on single metrics, and it uses always best paths according to policy and metric. The increase in utilization of QoS routing algorithms stems from the distribution of traffic flows among various paths, including those of inferior quality. The point here is that a path must satisfy certain criteria but it doesn't need to be the best choice ('quality as good as needed').

Based on existing routing and path selection protocols (including resource reservation protocols with routing interfaces) a solution is to be designed that supports routing decisions at exterior gateways. Such decisions can be based on different criteria/metrics, such as pricing, load measurements etc. In this context, the QoS routing framework described in [5] might provide a basis for this project.

As an interesting point, it should be mentioned that it is possible to transport partial information about network load and/or pricing (even combined with data traffic) to support informed routing decisions. See also [13].

Suitable algorithms have to be found or developed that support the correct operation of the routing system (i.e. finds always a path, satisfies specified criteria, does not produce loops, etc.). Furthermore, the solution should have a low complexity (focus on message overhead, then memory and time complexity).

### A.3.3 Implementation of an Interdomain Path Selection Protocol

The design of the algorithm and protocol has to be translated into an implementation running on ns-2. Only the basic characteristics have to be supported; it is not intended (nor possible) to implement a simulation of a 'ready-to-deploy' protocol. However, care should be taken to preserve the basic properties of the algorithms used.

### A.3.4 Evaluation/Measurements of Routing Protocols

In general, the most important measure to look at is (1) messaging overhead. However, for completeness we should also look at (2) processing costs and (3) memory requirements.

#### Complexity Measures

Before doing complex simulation runs, applying complexity measures to the routing protocols give insight about the general properties (actually, this should be done already in the design phase, when selecting or developing algorithms).

#### Benchmarks

When comparing new solutions we usually find upper bounds, given by theoretically optimal solutions. When choosing topologies and traffic in a controlled environment it is possible to benchmark routing decisions against optimal network states. For example, using exhaustive search or other optimization techniques, we can find the best possible utilization for a topology/traffic configuration (such algorithms ususally differ from real routing algorithms by (1) having all information available at a certain point in time (but no knowledge about the future) and (2) by using very complex path calculations). Such optimality criteria can be used to show whether it is worth to send more updates of routing information etc.

On the other hand, existing solutions can be used to compare to lower bounds. For example, we might ask, what is the improvement over a pure OSPF-based network or what is the routing overhead compared large scale networks using BGP.

In general, the metrics we are interested in, include congestion/packet loss, utilization, and call-blocking rate, depending on the type of network and traffic.

Evaluation is a time consuming task and should be coordinated with the semester projects by Markus Foser and Gabriele Giambonini.

## A.4 Remarks

- Mit dem Betreuer sind wochentliche Sitzungen zu vereinbaren. In diesen Sitzungen soll der Student mündlich über den Fortgang der Arbeit berichten und anstehende Probleme diskutieren.
- Am Ende der ersten Woche ist ein Zeitplan für den Ablauf der Arbeit sowie eine schriftliche Spezifikation der Arbeit vorzulegen und mit dem Betreuer abzustimmen.
- Am Ende des zweiten Monats der Arbeit soll ein kurzer schriftlicher Zwischenbericht abgegeben werden, der über den Stand der Arbeit Auskunft gibt (Vorversion des Berichts).
- Bereits vorhandene Software kann übernommen und gegebenenfalls angepasst werden. Neuer Code soll möglichst sauber in den Bestehenden integriert werden.
- Die Dokumentation ist mit dem Textverarbeitungsprogramm łFrameMaker zu erstellen.

## A.5 Results

Neben einem mündlichen Vortrag von 15 Minuten Dauer im Rahmen des Fachseminars Kommunikationssysteme sind die folgenden schriftlichen Unterlagen abzugeben:

- Ein kurzer Bericht in Deutsch oder Englisch. Dieser enthält eine Darstellung der Problematik, eine Beschreibung der untersuchten Entwurfsalternativen, eine Begründung für die getroffenen Entwurfsentscheidungen, sowie eine Auflistung der gelösten und ungelösten Probleme. Eine kritische Würdigung der gestellten Aufgabe und des vereinbarten Zeitplanes rundet den Bericht ab (in vierfacher Ausführung).
- Ein Handbuch zum fertigen System bestehend aus Systemübersicht, Implementationsbeschreibung, Beschreibung der Programm- und Datenstrukturen sowie Hinweise zur Portierung der Programme. (Teil des Berichts)
- Eine Sammlung aller zum System gehörenden Programme.
- Die vorhandenen Testunterlagen und -programme.
- Eine englischsprachige (deutschsprachige) Zusammenfassung von 1 bis 2 Seiten, die einem Aussenstehenden einen schnellen berblick über die Arbeit gestattet. Die Zusammenfassung ist wie folgt zu gliedern: (1) Introduction, (2) Aims & Goals, (3) Results, (4) Further Work.

## A.6 Bibliography

[1] E. Crawley et al., RFC 2386, A Framework for QoS-based Routing in the Internet, August, 1998

[2] C. Huitema, Routing in the Internet, Prentice Hall, 1995

[3] G. Fankhauser et al., Reservation-based Charging in an Integrated Services Network, 4th INFORMS TCOM, Boca Raton, USA, March 1998.

[4] J. Moy, OSPF 2.0, RFC 1583

[5] S. Chen and K. Nahrstedt, Distributed QoS Routing in High-Speed Networks Based on Selective Probing, 23rd LCN, Boston, October 1998.

[6] Q. Ma and P. Steenkiste, On QoS Path Computation, INFOCOM 98.

[7] George Apostolopoulos, Roch Guérin, Sanjay Kamat, Quality of Service Based Routing: A Performance Perspective, SIGCOMM 98.

[8] Z. Wang and J. Crowcroft, Routing Algorithms for Supporting Resource Reservations, IEEE JSAC, 1996.

[9] A.V.Vasilakos, K.G.Anagnostakis, A.Pitsillides, An Evolutionary Fuzzy Algorithm for QoS and Policy-Based Inter-Domain Routing in Heterogenous ATM and SDH/SONETNetworks, In proceedings of EUFIT97, Sept. 8-10, Aachen, Germany

[10] IBM PNNI Whitepaper, http://www.networking.ibm.com/pnni/pnniwp.html

[11] M. Ohta et al., Path QoS collection, INET 97

[12] S. Savage et al., Detour: A Case for Informed Internet Routing and Transport, University of Washington, TR UW-CSE-98-10-05, Seattle, 1998

[13] C. Vögtli, Auktions-basierte Reservationen und Verrechnung für das Next Generation Internet, Diplomarbeit, TIK/ETHZ, März 1998.

# Appendix B

# `flowsim`: a network flows simulator

## B.1 Introduction

`flowsim` is a network simulator written in Java which is very small and simple, therefore easily understandable and extensible.

The *design-by-interfaces* approach was taken, which consists in designing separately the interfaces and the implementation. The implementation is only used through the corresponding interface, which is a very good design practice. The implementation can be easily changed, if the need arises and it is also conceptually easier to comprehend the relation between the various objects.

Another property of this network simulator is that the simulation scenario is written also in java and therefore must be compiled each time a modification to the scenario is made. This is different from simulators such as `ns2` [ns2] where the simulation engine is compiled and the scenario is interpreted. We think this dual approach only makes the whole system much more complicated and inefficient. The scenario compilation time is negligible when compared to the total design-time and run-time of the scenario. The gained simplicity is enormous: everything is implemented the same way and is interoperable.

The principal difference of `flowsim` is however it's *flow* focus as opposed to the *packet* focus of simulators such as `ns2`. The difference is explained in the next section.

The `flowsim` classes are organised in various packages:

| | |
|---|---|
| `flowsim.core` | The simulator engine described in this chapter |
| `flowsim.util` | Various utility classes not specific to `flowsim` such as probability distributions, etc. |
| `flowsim.dv` | The Distance-Vector routing implementation described in appendix C. |
| `flowsim.cbdv` | Class-Based Distance Vector, also shortly described in appendix C. |
| `flowsim.slatr` | SLA Trading Based Routing: implementation goal of the simulation, described in chapter 7. |

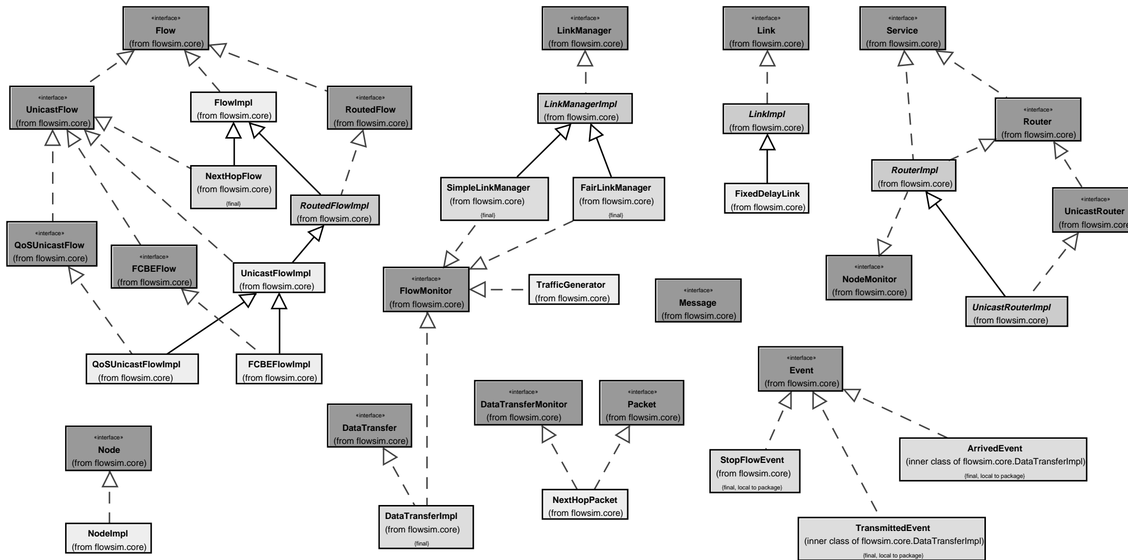



Figure B.1: Class Hierarchy of the flowsim.core package

## B.2 Flows vs. Packets

When a high number of packets are transmitted from a source to a destination their behaviour can be represented and often analysed as a *flow*. When the inter-provider traffic is considered you no more see the traffic as packets which are forwarded, but you see flows.

Consider the bandwidth used by packets transmitted on a link: an example usage in function of time for two flow of packets A and B is drawn in figure B.2.

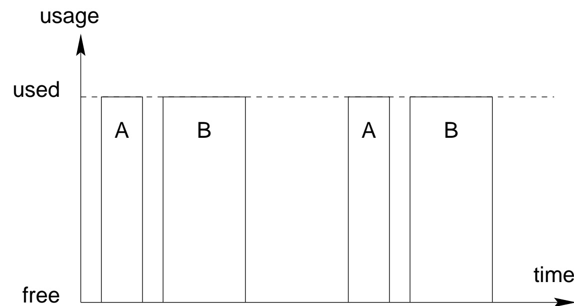



Figure B.2: Packet model of link usage

The figure B.2 is certainly a correct representation of the reality: a link is normally used or not used: two packets aren't transmitted at the same time (i.e. time multiplex). What is however important to us is only how much resources are used by the packets of flow A and B. The representation of the bandwidth usage is displayed in figure B.3.

These are in fact two ways to look at the same thing. This dual vision results also in two possible implementations: you can implement, as in ns2 the low-level view of packets or the high-level view of flows.

The flows view was implemented because it is, when you aren't interested at the low-level details

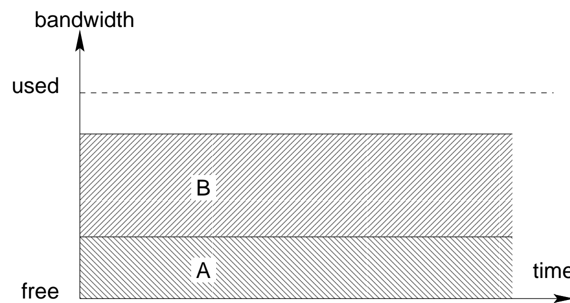



Figure B.3: Flow model of link usage

of protocols, much more easy to analyse and much more faster.

## B.3 Simulator

The *Simulator* object does contain all the services which are needed globally by every element such as the scheduler.

```
public class Simulator
{
        public long now();
        public void run(long how_long);
        public EventQueueElement schedule(long when, Event e);
        public EventQueueElement schedule_absolute(
                long when, Event e);
        public void error(Object who, String str);
        public void debug(Object who, String str);
        public void set_verbose(boolean v);
        public Simulator();
}
```

## B.4 Scheduler

The *Scheduler* does maintain the time of the system. It is responsible for maintaining the list of events to be executed in the future and triggering them when the time has come.

It does use a *EventQueue* to maintain the list of events and pickup the next event to trigger.

Event can be scheduled in *absolute* or *relative* time. Scheduling events in the past is forbidden.

```
public class Scheduler
{
        public long now();
        public void run(long how_long);
        EventQueueElement schedule_absolute(long when, Event e);
        public EventQueueElement schedule(long when, Event e);
```

```
        public String toString();
        public Scheduler(Simulator s);
}
```

### B.4.1 EventQueue

A *EventQueue* is responsible for the storage of *Events* and for the retrieval of the Event with the smallest time-stamp.

```
public interface EventQueue {
        EventQueueElement put(long time, Event event);
        long first_time();
        EventQueueElement get();
        public int size();
}
```

### B.4.2 Event

An *Event* in flowsim is a command to be executed. The Event interface is so defined:

```
public interface Event
{
        void trigger();
}
```

Defining a new Event is very easy: you simply write a class which implements the Event interface.

## B.5 Node

A node represents a router or a provider in a network which is connected to other nodes by links. It consists of an id/name, interfaces and services on its ports. Every interface is a LinkManager instance.

```
public interface Node
{
        int    get_id();
        String get_name();
        int  get_max_ifaces();
        int  get_ifaces_count();
        void set_iface(int n, LinkManager f);
        int add_iface(LinkManager f);
        LinkManager get_iface(int i);
        int get_iface(Node n);
        Link get_link(int n);
        Node get_peer(int n);
        Service get_service(int port);
```

```
        void set_service(Service service, int port);
        public void attach_monitor(NodeMonitor nm);
        public void remove_monitor(NodeMonitor nm);
}
```

### B.5.1 Service

Entity which receives messages on a specific port-number of a node. The Message object asks when it arrives on a Node the Service object at a port number and then converts this object to what it thinks it should be and calls the appropriate method.

```
public interface Service
{
        Node get_node();
        int  get_port();
}
```

## B.6 Monitors

In `flowsim`, the monitor/observed pattern is heavily used to simplify the implementation of complex causal relations and to facilitate extension. For example every `Router` implementation should register itself to the node where it resides as a `NodeMonitor` so that when an interface changes it is notified and can make what is necessary to correctly implement the routing algorithm.

Another example are the `LinkManager` which will normally register themselves as `FlowMonitors` to every flow they manage so that when the desired bandwidth of the flow changes they can respond by restarting their allocation algorithm.

Currently, five monitors are defined: `DataTransferMonitor`, `LinkManagerMonitor`, `NodeMonitor`, `FlowMonitor`, `LinkMonitor` and `TotalBandwidthMonitor`.

As an example, the `FlowMonitor` interface is here reported:

```
public interface FlowMonitor
{
        void flow_started(Flow f);
        void flow_stopped(Flow f);
        void flow_blocked(Flow f);
        void flow_changed_bw(Flow f, int was_bw);
        void flow_changed_desired_bw(Flow f, int was_bw);
        void flow_changed_available_bw(Flow f, int was_bw);
}
```

## B.7 Link

A link object simulates the physical behaviour of a real-link. It does have a total bandwidth, which can be used by flows which circulate on it. The unused bandwidth is called 'residual bandwidth'. A link is in flowsim like a bucket which can be filled up-to it's capacity.

The delay (for the added packet, maximal and minimal) can be also asked from the link object.

The packet queueing implementation can be simulated in flowsim by modelling the delay with a function of the residual bandwidth.

```
public interface Link
{
        Node    get_from();
        Node    get_to();
        int     get_bw();
        int     get_rbw();
        int     get_delay(); /* in microseconds (us) */
        int     get_min_delay();
        int     get_max_delay();
        boolean add_flow(int fbw);
        void    remove_flow(int fbw);
        void    attach_monitor(LinkMonitor lmo);
        LinkMonitor get_monitor();
}
```

## B.8 Flow

A *flow* simulates a stream of data packets in the network. Flow objects have a source node (`from`), bandwidth (`bw`), flow-id (`flowid`) and differentiated-services-id (`ds`). They can be started, stopped, blocked and forwarded to an interface.

Each flow has three type of bandwidths:

- *desired bandwidth*: This is what the flow would like to have (for example a best-effort flow could have this set to infinity. This is the only bandwidth which it is allowed to modify outside of a `LinkManager`.
- *current bandwidth*: The `LinkManager`, on the basis of the desired-bandwidth will decide how much real bandwidth to give to this flow on this link and will set this value accordingly (this value is called often only 'bw').
- *available bandwidth*: The `LinkManager` will also tell the flow, when it does set the current-bandwidth on the basis of the desired-bandwidth, how much bandwidth it would have received if it would have set it's desired-bandwidth to infinity. This value is important to flow-control behaviour emulations (see `FCBEFlow`).

Each flow can be monitored and it's monitors will receive notifications when the flow was started, stopped, blocked or when it's current, available and desired bandwidths change.

```
public interface Flow {
        Node    get_from();
        Service get_from_service();
        void    set_from_service(Service from);
        int     get_bw();
```

```
        int     get_desired_bw();
        int     get_available_bw();
        void    set_bw(int available_bw, int bw);
        void    set_desired_bw(int bw);
        int     get_flowid();
        void    set_flowid(int flowid);
        int     get_ds();
        void    set_ds(int ds);
        void    start();
        void    stop();
        void    block();
        boolean is_blocked();
        double  utility();
        void    attach_monitor(FlowMonitor monitor);
        void    detach_monitor(FlowMonitor monitor);
}
```

### B.8.1 UnicastFlow

UnicastFlows are a sub-type of Flows, which do also define a destination. Which nodes are traversed by the UnicastFlow must also be stored. Therefore, UnicastFlows are implemented in `flowsim` by using a linked-list of UnicastFlows (see figure B.4: the linked-list relation is indicated by the thin black arrows). Every UnicastFlow object in the list represents the flow which start from that node. This form of representation permits to have flows which have different bandwidths on different links.

We call the previous UnicastFlow of a UnicastFlow it's *parent* and inversely it's *child*.

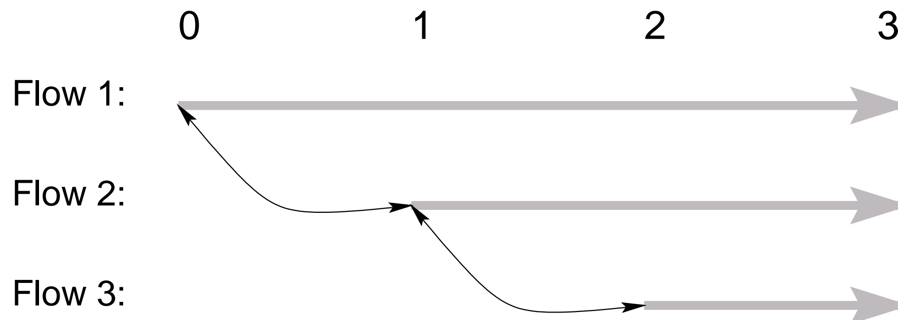



Figure B.4: Unicast flow child/parent relation

```
public interface UnicastFlow extends Flow
{
        void next_node(Node child_node);
        Node get_to();
        UnicastFlow get_parent();
        UnicastFlow get_child();
        LinkManager get_iface();
        UnicastFlow add_child_same_node();
```

```
        void set_parent_bw(int bw);
}
```

### B.8.2 FCBEFlow

FCBEFlow is a UnicastFlow which does simulate *best-effort* flows with *flow control*. The flow control is implemented by limiting the parent's bandwidth after the delay on the link between parent and child.

```
public interface FCBEFlow extends UnicastFlow
{
        void set_child_available_bw(int child_abw);
}
```

### B.8.3 DataTransfer

Very frequently the transport of some data through flows must be simulated. Such cases are for example the transmission of DV update messages or of a file with a file-transfer protocol. This *data transfer* is simulated by a flow which is appropriately stopped when the data was transmitted. The implementation class (`DataTransferImpl`) does implement that by starting a `UnicastFlow` and watching if it does change bandwidth. If it does change bandwidth, the transmission time is changed accordingly.

A DataTransfer object can also be monitored by a `DataTransferMonitor` object, which will for example be notified when the transfer is completed.

```
public interface DataTransfer
{
        long get_start_time();
        int get_from_port();
        int get_to_port();
        int get_total_bits();
        int get_remaining_bits();
        int get_bw();
        Flow get_flow();
        void start();
        void abort();
        void attach_monitor(DataTransferMonitor dtm);
}
```

### B.8.4 Message

The simulation of the transport of messages, such as a DV update message, is further simplified by the usage of `Message` objects. A `Message` object does define the size of the message and the action to perform when the message is delivered to it's destination. A `Message` object is used by a `Packet` object to define the contents of the packet.

```
public interface Message
{
        int size();
        void arrived(Node to);
}
```

### B.8.5 Packet

A packet simulates a single packet using a flow. The packet is simulated by keeping the flow running for the amount of time needed to transport all the data. The data contained in the packet is represented by a Message object, which determines the it's size and what should be done when the packet arrives on a node.

```
public interface Packet
{
        Message get_msg();
        int get_remaining_bits();
        void start();
}
```

## B.9 LinkManager

A `LinkManager` does simulate the Admission Control/Queueing for a Link. It does decide which flows are blocked and which are accepted. It's main function is setting the current-bandwidth in function of the desired-bandwidths of the flows currently forwarded on the link (see `Flow`).

LinkManagers are used to model network interfaces on a node.

Some LinkManagers do provide a distinction between *best-effort* (`be`) and *non-best-effort* or *reserved* (`notbe`) flows. The amount of reservable bandwidth and the total bandwidth currently reserved by non-best-effort flows can be asked in such a case.

```
public interface LinkManager
{
        Iterator flows();
        Node get_peer();
        Link get_link();
        int get_rbw_notbe();
        int get_reservable_bw();
        long add(Flow flow);
        void bw_changed(Flow flow);
        void remove(Flow flow);
        void attach_monitor(LinkManagerMonitor lmm);
        void detach_monitor(LinkManagerMonitor lmm);
}
```

## B.10 Router

The routing is done in `flowsim` in three phases:

1. The flow `F` is started or arrives on node `N`. The router which is responsible for routing the flow is found with `N.get_service(F.get_router_port())`.
2. The router `R` is instructed to route flow `F` by calling `R.route(F)`
3. The router `R` chooses the appropriate interface `I` (represented by a `LinkManager`) through which the flow should go and calls `F.forward(I)`.

```
public interface Router extends Service
{
        void route(Flow flow);
}
```

### B.10.1 RoutedFlow

`RoutedFlows` are flows which can be routed by Routers. The `router_port` property is used to get the right router on the nodes (via `node.get_service(router_port)`). The `forward` is called by the router when it has made the decision on which link the flow should go (see `Router`).

```
public interface RoutedFlow extends Flow
{
        int  get_router_port();
        void set_router_port(int port);
        void forward(LinkManager lm);
}
```

# Appendix C

# Distance-Vector Routing in Flowsim

## C.1 Introduction

In the `flowsim` core package only the interface which should be implemented by routers is provided: no router implementations. The interface is as follows:

```
public interface Router extends Service
{
        void route(Flow flow);
}
```

The only method which should be implemented is `route`. The `route` method of of a router will be called whenever a flows requests to be routed by that router. See appendix B for further explanations.

It was chosen to implement DV-routing in a separate package (`flowsim.dv`), because it is a specific algorithm and because `flowsim` can work also without it. It was also an advantage to do so because the extendibility of `flowsim` for new routing algorithms was thus proved.

This implementation was also thought to show how to extend `flowsim` without bloating the core package.

## C.2 DV Update Messages

The DV Update Message is so implemented (only the interface is shown):

```
public final class DVUpdateMessage implements Message
{
        public int size();
        public void arrived(Node to);
        public DVUpdateMessage(Node from, DVRouter router,
                DVTable table, int count);
}
```

First thing to note is that the `Message` is implemented, so that the update messages can be very comfortably be transported by `Packet` objects (see `flowsim.core` description).

To implement the `Message` interface, `size` and `arrived` are implemented.

Because it is interesting and because it is short, the `arrived` method is shown:

```
public void arrived(Node to) {
        router.receive_update(from,table);
}
```

The `receive_update` method of the DV-router where the message arrives is called. This is very simple: no complicated parsing and demultiplexing of raw packets is done!

Note that `router` was initialised in the message constructor, along with the source node and the DV-update message contents.

## C.3 DV Table

A DV Table is simply a database of `(destination,cost)` pairs. `cost` is normally the *hop-count* to that destination. This is the interface of `DVTable` objects:

```
public interface DVTable extends Cloneable
{
        void put(Node n, int cost);
        int get(Node n);
        Enumeration nodes();
        int size();
        Object clone();
}
```

The methods should be self-explanatory ...

## C.4 Router Implementation

The DV Router interface is very simple:

```
public interface DVRouter extends Router
{
        void receive_update(Node from, DVTable t);
        void debug_table();
}
```

`receive_update` is called, as previously explained, by DV update messages which should be processed by this DV router. `debug_table` does print the DV Table in a readable way so as to debug the implementation.

Upon receipt of update messages, the DV router will update the table for that neighbour and re-run the DV algorithm.

## C.5 Class-Based DV Routing

Class-Based Distance Vector Routing is implemented in the `flowsim.cbdv` package by subclassing the `DVRouter` implementation.

In `DVRouter`, the cost metric was implemented as a method call (`link_cost`) which did always return 1, thus implementing a *hop-count* DV Routing. In `CBDVRouter`, the `link_cost` method was re-implemented.

How this single-metric is constructed is explained in the experiments chapter 3.

# Bibliography


[BBC+99] Steven Blake, David Black, Mark Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, 1999.

[BCS94] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. RFC 1633, 1994.

[CF98] David D. Clark and Wenjia Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4), August 1998.

[CN98a] Shigang Chen and Klara Nahrstedt. Distributed QoS routing. 1998.

[CN98b] Shigang Chen and Klara Nahrstedt. An overview of quality of service routing for next-generation high-speed networks: Problems and solutions. *IEEE Network*, November/December 1998.

[GOW96] R. Guerin, A. Orda, and D. Williams. Qos routing mechanisms and ospf extensions, November 1996.

[Hui95] Christian Huitema. *Routing in the Internet*. Prentice Hall, Englewood Cliffs, 1995.

[Jac90] Van Jacobson. Compressing tcp/ip headers for low-speed serial links. RFC1144, 1990.

[KS98] Seok-Kyu Kweon and Kang G. Shin. Distributed QoS routing using bounded flooding. 1998.

[LR89] K. Lougheed and Y. Rekhter. A border gateway protocol (bgp). RFC 1105, 1989.

[MSC] http://sdl-forum.org.

[NJZ97] K. Nichols, V. Jacobson, and L. Zhang. A two-bit differentiated services architecture for the internet. http://www-nrg.ee.lbl.gov/papers/2bitarch.pdf, November 1997.

[ns2] http://www-mash.cs.berkeley.edu/ns.

[Pos81] Jon Postel. Internet protocol. RFC 791, 1981.

[RZB+97] R.Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. Resource ReSerVation Protocol (RSVP), functional specification. RFC 2205, 1997.

[SL97] Q. Sun and H. Langendorfer. A new distributed routing algorithm with end-to-end delay guarantee, 1997.

[Sti96] Burkhard Stiller. *Quality-of-Service, Dienstgüte in Hochleistungsnetzen*. International Thomson Publishing GmbH, Bonn, first edition, 1996.

[SW] S. Shenker and J. Wroclawski. Analysis and simulation of a fair queueing algorithm. *Internetworking: Research and Experience*, 1(1).

[Tan96] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, Inc., New Jersey, third edition, 1996.

[Var96] Hal R. Varian. *Intermediate Microeconomics, a modern approach*. Norton, 1996.

[WC96] Zheng Wang and Jon Crowcroft. Routing algorithms for supporting resource reservation. 1996.