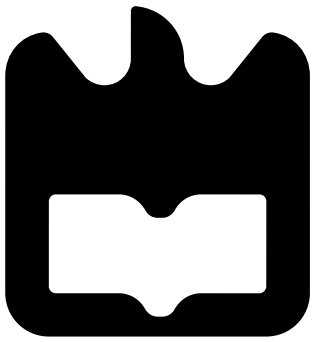
Universidade de Aveiro
Departamento de
Electrónica, Telecomunicações e Informática.
2010

Carlos Miguel Martins de Campos

Sobre-Reserva em Redes com Controlo Centralizado e Distribuído



Carlos Miguel Martins de Campos

Sobre-Reserva em Redes com Controlo Centralizado e Distribuído

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica, realizada sob a orientação científica da Professora Doutora Susana Isabel Barreto de Miranda Sargento, Professora Auxiliar do Departamento de Electrónica, Telecomunicações e de Informática (DETI) da Universidade de Aveiro, e co-orientação do Professor Doutor Augusto José Venâncio Neto, Professor adjunto do Instituto de Informática da Universidade Federal de Goiás, Brasil, e colaborador no Instituto de Telecomunicações de Aveiro.

o júri / the jury

presidente / president

Professor Doutor Aníbal Manuel Oliveira Duarte
Professor Catedrático do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

vogais / examiners committee

Professora Doutora Susana Isabel Barreto de Miranda Sargento
Professora Auxiliar do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro

Professor Doutor Rui Pedro de Magalhães Claro Prior
Professor Auxiliar do Departamento de Ciências e Computadores da Faculdade de Ciências da Universidade do Porto

agradecimentos / acknowledgements

Quero agradecer de forma especial aos meus pais, que sempre me apoiaram e ajudaram. É devido ao esforço deles que hoje acabo esta etapa da minha vida. Agradeço também à minha avó, pelo carinho e apoio que deu sempre, ao longo da vida. Aos meus irmãos, que estiveram sempre presentes, muito obrigado.

Agradeço carinhosamente à minha namorada pela sua enorme paciência, compreensão e encorajamento. Por me apoiar nos momentos difícies e por me fazer ver os meus erros, nunca deixando de estar sempre do meu lado.

Quero agradecer à minha orientadora, Prof. Dra. Susana Sargento, pela sua permanente disponibilidade e por me mostrar sempre o melhor caminho. Agradeço ao Tiago Condeixa pela importante ajuda dada com o simulador usado neste trabalho. Sem ele, teria tido muitas dificuldades adicionais. Agradeço em especial ao Evariste Logota pela grande ajuda e motivação que me deu. O bom planeamento da parte teórica que ele fez, contribuiu para melhorar a plataforma final desta dissertação. Aprendi muito com ele, tanto a nivél técnico como pessoal. Foi muito importante para a realização deste trabalho.

Agradeço a todos os meus amigos, não só pela ajuda e preocupação extra neste último ano, mas por tudo ao longo deste “nosso” percurso académico. Foram e continuarão a ser uma parte importante da minha vida.

palavras-chave

SOMEN, COR, A-COR, QoS, CoS, controlo distribuído, sobre-reserva

resumo

As redes de futura geração são esperadas vir a suportar novas funcionalidades (escalonamento, qualidade de serviço, *muticast*) sobre tecnologias de transporte heterogéneas, com suporte para muitos utilizadores simultaneamente e a um custo razoável. Neste sentido, esforços de investigação recentes afirmam que o uso de mecanismos de sobre-reserva em redes com classes de serviço é promissor, pois permite a optimização do desempenho global da rede, uma vez que possibilita tratar agregados de fluxos simultaneamente. No entanto, as propostas existentes enfrentam vários problemas de eficiência, onde existe desperdício de largura de banda e, portanto, o aumento desnecessário da probabilidade de bloqueio da sessão.

Por outro lado, a crescente utilização da *Internet*, faz com que mecanismos centralizados fiquem sobrecarregados, ocorrendo cada vez mais falhas. Embora os mecanismos centralizados possam ser mais fáceis de gerir, deixam muito a desejar em comparação com sistemas distribuídos. A descentralização permite a execução simultânea de operações em entidades diferentes através de uma rede, conseguindo um melhor desempenho. No entanto, exige a sincronização de informações de controlo para evitar decisões erradas e incompatíveis.

Esta dissertação implementa e testa uma plataforma que usa mecanismos de sobre-reserva de recursos dinamicamente (*COR*, *A-COR*), que permitem gerir agregados de fluxos assegurando requisitos de qualidade de serviço, tanto num cenário de controlo centralizado como distribuído, e suporte para *multicast* e balanceamento de carga. Particularmente, estes mecanismos levam em consideração a partilha de interfaces pelos diversos caminhos da rede, melhorando a distribuição de recursos disponíveis pelas diversas classes. Mais importante, tudo é feito de forma dinâmica e apenas quando necessário para cada interface na rede, melhorando a prestação global do serviço. *Self-Organization of Multiple Edge Nodes* (*SOMEN*) opera de uma forma distribuída permitindo que pontos chave na rede (*CDP*) operem em conjunto para controlar os recursos da rede mantendo um nível baixo de mensagens e processamento.

Os resultados obtidos por simulação demonstram que sobre-reserva de recursos pode prevenir a violação dos requisitos de qualidade de serviço enquanto minimiza significativamente a probabilidade de bloqueio da sessão. Demonstram também que a distribuição do controlo da rede é escalável para redes futuras, mantendo baixos níveis de mensagens de sincronização.

keywords SOMEN, COR, A-COR, QoS, CoS, distributed control, Over-provisioning

Abstract Future Networks are expected to support new features and capabilities (scalability, Quality of Service (QoS), multicast) in order to provide support for value added sessions (e.g. multimedia, personalized, haptics, etc.) over heterogeneous transport technologies to many users simultaneously, and at reasonable cost. Hence, recent research efforts claim that class-based network resource over-provisioning is promising, since it allows minimizing undesired QoS control states, processing and signaling overheads to improve system overall performance. However, existing proposals mostly waste bandwidth and therefore increase session blocking probability unnecessarily while incurring QoS violation. Besides, the increasing dependence on IT technologies is making central controllers more and more bottlenecked while the network infrastructures are subject to frequent failures. As an alternative, decentralization allows simultaneous operations at distributed entities throughout a network, achieving better performance. Nevertheless, it must be correctly designed to assure a cost-effective synchronization of the distributed decision points to prevent wrong and incompatible decisions.

Keeping this in mind, this dissertation implements and provides a platform to assess new mechanisms for dynamic aggregate QoS over-reservations in both centralized and decentralized architectures with support for multicasting and load balance. In particular, the mechanisms improve the control of aggregated resources by taking communication paths correlation patterns and sessions requests into account and efficiently distribute the shared surplus of reservations among existing Classes of Service (CoSs). More importantly, this is performed in a dynamic manner for every outgoing interface inside a network upon need, in a way that effectively allows system overall performance optimization. Moreover, a decentralization control approach, assuring an efficient Self-Organization of Multiple Edge Nodes (SOMEN) has also been evaluated. Hence, multiple distributed network Control Decision Points (CDPs) are enabled to cooperate and self-manage keeping low control signaling, states and processing load.

The simulation results prove that efficient bandwidth over-reservation can effectively prevent QoS violation while significantly minimizing the waste of bandwidth and the unnecessary increase of session blocking probability. Further, they show a cost-effective decentralization of the control in current and future networks with low synchronization signaling loads.

Contents

List of Tables

List of Figures

Acronyms

IP	Internet Protocol
IPTV	Internet Protocol Television
CoS	Class of Service
QoS	Quality of Service
NS	Network Simulator
OSMAR	Overlay for Sourcespecific Multicast in Asymmetric Routing environments
STDOUT	Standard Output
MRIB	Multicast Routing Information Base
RIB	Routing Information Base
NetCIB	Network Context Information dataBase
CDP	Control Decision Point
VOPR	Virtual Over PRovisioning
SOMEN	Self Organizing Multiple Edge Nodes
SOMEN-E	SOMEN Edge
SOMEN-C	SOMEN Core
COR	Class-based bandwidth Over-Reservation
A-COR	Advanced Class-based bandwidth Over-Reservation
MARA	Multi-user Aggregated Resource Allocation
BW	bandwidth
DFF	Dynamic Filtering Functions
DTSF	Dynamic Threshold Synchronization Functions
SRF	System Resilience Functions

IGMP	Internet Group Management Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
BGRP	Border Gateway Reservation Protocol
RTP	Real-Time Transport Protocol
SRM	Scalable Reliable Multicast
PIM	Protocol Independent Multicast
ISP	Internet Service Provider
CPU	Central Processing Unit
CBR	Constant Bit Rate
SSM	Source Specific Multicast
Cen.	Centralized Scenario

Chapter 1

Introduction

1.1 Motivation

Nowadays, the need for various and attractive services over the Internet is increasing rapidly. These services include, but not limited to, the information sharing, video streaming, IPTV, Mobile Gaming, video conferencing and personalized services. YouTube is a good reference to illustrate this reality.

It is also expected that the Internet integration should be exploited to enhance connectivity for mission-critical information exchange among users in domains where life may be at risk such as emergency and crisis intervention. In the research community, it is argued that multicasting and context-awareness together will exploit situations in which users share the same interests and request similar services to improve the effectiveness of session and network control for many users simultaneously with lesser amount of the network resource.

On one hand, the context-awareness is considered to allow the collection and delivery of information on mobile terminals, network and environment, so that dynamic events can trigger session and network reactions, such as service and network reconfiguration, multiparty session content delivery and re-negotiation, and seamless context-aware mobility. On the other, multicasting consists of sending the same copy of packet to many users simultaneously and therefore allows optimizing the network resource utilization.

However, the context-aware networks pose serious scalability problems since any change to context, such as, location, mobility, velocity, preferences, presence, etc., can change the overall services and network environments, requiring completely restructuring of the network and multicast sessions in a very dynamic way. Attempt to overcome these issues becomes even more demanding due to the fact that communication networks are subject to frequent failures, mostly being unintentional such as natural disasters, human errors while many failures are now often the result of intentional interruptions (e.g. malicious attacks). Moreover, applications and services must be provided over emerging heterogeneous technologies in a way that assures acceptable Quality of Service (QoS) while the current Internet is mainly best-effort based. Hence, as the IP Multicast works on the Internet, much research has been done in recent years for the support of QoS in multicast networks, both with fixed and mobile users.

Even today, existing mechanisms to provide multicast services are still very static, being without taking into account the user's mobility and their preferences (context). Further, the use of a central control element that takes decisions in such dynamics networks for the immense variety of context

information and service demands is not scalable.

Considering the heterogeneity and the dynamics of next generation networks, there are many challenges that still deserve special attention. A primary challenge is to develop dynamically configurable context-aware and QoS-enabled networks with support for multicasting to be highly available to allow access to information anywhere and anytime.

Hence, this Dissertation aims to evaluate new mechanisms for the control of the network and propose improvements in accordance with the results. In particular, two novel Class-based bandwidth Over-Reservation algorithms, Class-based bandwidth Over-Reservation (COR) [14] and Advanced COR [12], are evaluated and compared with a state-of-the-art solution, the Multi-user Aggregated Resource Allocation (MARA) [19]. The evaluations are carried out in both centralized and decentralized architectures which introduce each, new control functionalities to effectively allow a cost-effective control of the network, keeping low control signaling, states and processing overheads.

1.2 Objectives

The main objective of this dissertation is to evaluate, in the Network Simulator (NS), new mechanisms for the control of the network resources, COR and Advanced COR, in both a centralized network architecture and in a distributed network architecture, using Self Organizing Multiple Edge Nodes (SOMEN). Several reasons justify the use of this simulator over other exiting softwares. Most of the reasons come from the fact that this simulator is open source and widely used in the scientific community. Moreover, some related works (e.g. MARA[19]) have been implemented in this simulator.

The SOMEN mechanism focuses on distributed network control, but the changes needed to make it operate in a centralized way were justifiable to implement a centralized approach as well. Also, it contributes to a more promising work once we can compare results between both approaches. The evaluation of SOMEN is them divided into two main architectures: one with centralized control and another with distributed control.

Both SOMEN architectures and all its composing mechanisms had to be fully programmed in C++ for the referred simulator.

By means of simulation we compare the efficiency and performance of both COR and A-COR with a state of the art proposal, the Multi-user Aggregated Resource Allocation (MARA) [19].

The metrics for evaluation are mainly based on how effectively each over-provisioning algorithm uses the underlying network resources, the amount of wasted bandwidth (BW), the total singling load and the number of blocked sessions per algorithm.

1.3 Document Outline

In chapter 2 an overview about the fundamental aspects of the work is given, with emphasis on QoS, multicast, and distributed network control. Some of the main difficulties are explored and solutions are discussed.

The following chapter, chapter 3, will detail the solutions mentioned in chapter 2.

The centralized scenario is presented and their inner workings are explored. Next, the distributed approach is explained with emphasis on its main advantages and drawbacks. As an important part of this work, the different reservation algorithms are discussed later in the same chapter.

Chapter 4 is the description of the work done. The context in which it was developed, as well as the trade-offs made are discussed and explained. An introduction to the simulator is given, some implementation specific details are described, and a full explanation about the work is performed.

In Chapter 5 we discuss the simulation results obtained. These results shall validate the initial goals and reveal the potential of the algorithms discussed.

At last but not least, chapter 6 presents the conclusions on the implemented algorithms, mechanisms and performance, based on the attained results. Finally, several suggestions are made on how improvements could be done in the future.

Chapter 2

State of The Art

2.1 Introduction

The new trend of social, economic and technological merging, strongly requires QoS-aware scalable and robust network architectures.

Today, though it is widely accepted that redundancy and multi-homing in decentralized networks would better support scalability, reliability, availability and fast responsiveness, there is a serious lack of standard and solutions in this field.

The organization of this chapter is as follows.

Section 2.2 introduces the multicast basic functionality mechanism, as well as the involvement in this work.

In section 2.3 we deeply discuss the operations and trade-offs of over-provisioning schemes as well as some of the arguments that make this type of approach so promising.

Section 2.4 aims at emphasizing the benefices of distributed network control over the centralized one.

2.2 Multicast

Multicast can be useful if there is information that should be transmitted to various hosts. One common situation in which it is used is when distributing real time audio and video to the set of hosts which have joined a certain multicast group.

Multicast is much like Radio or Television in the sense that only those who have tuned their receivers, receive the information. Instead of frequencies, we have the Internet Protocol (IP) as a base, precisely IP addresses. The IP addressing scheme already reserves some addresses for this purpose. [27]

The usual practice in multicast is to use User Datagram Protocol (UDP), as Transmission Control Protocol (TCP) requires the feedback mechanisms support. However, research is taking place in order to develop some new multicast transport protocols. Several of these protocols have been implemented and are being tested. Giving descriptions of those multicast protocols is out of the scope of this section, but to give an example: Real-Time Transport Protocol (RTP) is concerned with multi-partite multimedia conferences, Scalable Reliable Multicast (SRM) is used by the web. Several multicast protocols can be found.

# Class	Address Range	Supports
A	1.0.0.1 to 126.255.255.254	Supports 16 million hosts on each of 127 networks
B	128.1.0.1 to 191.255.255.254	Supports 65,000 hosts on each of 16,000 networks
C	192.0.1.1 to 223.255.254.254	Supports 254 hosts on each of 2 million network
D	224.0.0.0 to 239.255.255.255	Reserved for **multicast groups**
E	240.0.0.0 to 254.255.255.254	Reserved for future use

Table 2.1: IP Address Range

The standard work of a multicast protocol is as follows. An IP multicast group address is used by sources and receivers to send and receive multicast messages. Sources use the group address as the IP destination address in their data packets. Receivers use this group address to inform the network that they are interested in receiving packets sent to that group.

There is usually a command join to start receiving packets in a certain group, as well as a leave message to stop. The protocol typically used by receivers to join a group is based on the Internet Group Management Protocol (IGMP).

To deliver the packets to several hosts multicast trees are built, adding information to the router's routing tables, Multicast Routing Information Base (MRIB). The protocol most widely used for this is Protocol Independent Multicast (PIM). There are variations of PIM implementations: Sparse Mode (SM), Dense Mode (DM), Source Specific Mode (SSM). In this work the later, SSM, is of relevance because it builds trees that are rooted in just one source, offering a more secure and scalable model. As so, it will be presented an overview of this technology.

When a source desires to start a transmission, IGMP protocol is used to inform its neighbor router that it will send data to a certain group. Then, when a user is interested in receiving this group data, it uses the same protocol to send a message to its neighbor router informing that he wishes to join the group.

After the router receives this message, it will send a PIM join message to all the other PIM routers, notifying them about the join of this user. The propagation of this PIM join message will enable the creation or the extension of the multicast delivery tree of the desired group from the source until the user. Routers maintain information about the interfaces they should forward packets belonging to this group.

Later, if the user aims to stop receiving the data flow, it leaves the group sending again a IGMP message to the neighbor router. If there are no more listeners in this sub-network, the router will send a PIM prune message to its above routers informing them to stop forwarding these packets in his direction. The same way, if at any time the source wishes to stop the transmission, it will also send an IGMP message to its neighbor router to communicate its decision, terminating the multicast communication to all the hosts.

The evolution to IGMPv3 became possible to users to specify the source, or group of sources, from which they want to receive data. To make this possible, new multicast protocols were developed to communicate between routers, as the PIM-SSM. This version has as main advantages the fact of being easily implemented, avoiding the necessity of configuring one router as Rendez-Vous Point, which in the former versions of PIM act as a confirmation point for all the requests, making this approach even more scalable than the prior versions.

As referred, each multicast group is represented by an IP address from a well-defined range (Table 2.1).

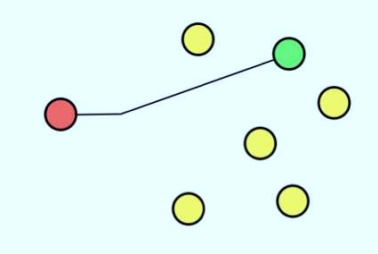


Figure 2.1: Unicast Diagram

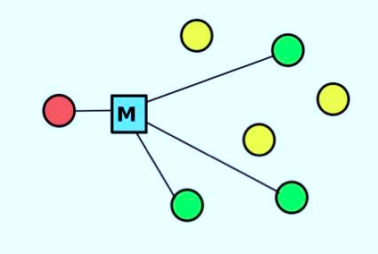


Figure 2.2: Multicast Diagram

Nevertheless, the integrated deployment of class-based QoS and IP multicast is not trivial, once the first permits an implementation of QoS in a scalable form and the second economizes bandwidth in the network. Moreover, QoS achieves scalability by pushing the complexity to edge routes, IP multicast operates on a per flow basis throughout the network. Simple PIM-SSM is not enough. The solution presented in the paper [22], is envisioned to overcome the lack of adaptation of the existing multicast protocols to asymmetric routing. Thus the Overlay for Sourcespecific Multicast in Asymmetric Routing environments (OSMAR) approach was designed to be used as an overlay for source-specific multicast protocols as PIM-SSM, empowering them to provide content distribution considering QoS and handle network asymmetries.

OSMAR aims to be used as an overlay for source-specific multicast protocols, like PIM-SSM, enabling them to deal with network asymmetries and to provide QoS-aware content distribution, while maintaining their specifications and state machine.

To accomplish this, OSMAR changes the values of the Multicast Routing Information Base (MRIB) tables, used by the multicast routing protocol to build the multicast tree, and considers the path from source to receivers. The updated MRIBs will then be used to build optimal trees based on the path that data really uses. In opposition, the normal operation of multicast protocols creates trees following the data path from receivers to the source. Therefore, OSMAR enabled multicast branches are created taking into consideration the connectivity (e.g. unidirectional links) and QoS characteristics configured for each source-receiver data path.

Later on, we'll see that this approach was not enough for this work mainly due to its limitation of only one path may exist between any two source – destination pair of nodes. To overcome this limitation, additional code had to be written and modified for the OSMAR NS 2.31 existing implementation support multiple paths.

2.3 QoS Model and Bandwidth Over-Provisioning

The future networks are expected to support new features to provide value added sessions (e.g. multimedia, personalized, etc.) over heterogeneous transport technologies with acceptable service quality. However, the current packet-based technologies cannot allow data transport beyond best-effort, narrowing session flows to experience undesired delay, jitter, and even packets losses.

Hence, there is the need to allocate bandwidth and deploy signaling to install, maintain, or remove resource reservation for sessions, with schedulers on nodes to ensure that each session receives the amount of bandwidth allocated to it, the bandwidth reservation mechanisms.

Although QoS control approaches are known as indispensable to maximize the value of future

networks, per-flow reservation approach such as IntServ[4], introduces excessive states, processing and signaling overheads and therefore raise serious scalability problems. Hence, QoS models based on Class of Services (CoSs) appeared suitable to prevent the performance degradations of per-flow approaches. In class-based networks (e.g. DiffServ [1]), flows are classified into a set of services CoSs at network borders (e.g. ingress routers) or central stations, based on predefined policies regarding QoS, protocols, etc.

Hence, in per-class QoS control, the reservation states are maintained per CoS and not per-flow, allowing to reduce the control overheads. However, the per-class reservation mechanisms driven by per-flow signaling approaches [17] still confront scalability issues since the QoS control operations are triggered with the increasing number of session demands.

In other words, the excessive control messages placed by per-flow signaling strategies, not only put heavy processing load on core router's Central Processing Unit (CPU), but also consume more bandwidth, memory and energy, while they affect the session setup time.

Alternatively to per-flow QoS control signaling approaches, aggregated over-provisioning techniques envision reserving to each CoS more bandwidth than currently required. This way, multiple flows can be accepted without signaling the network nodes, as long as resources are still available in order to optimize the network overall performance.

However, previous solutions mostly waste resources, increase session blocking probabilities unnecessarily while they incur QoS violation. Perhaps more importantly, they even fail to properly operate in completely decentralized networks with multiple distributed control decision points (e.g. ingress routers).

In the literature, the over-reservation technique used in Border Gateway Reservation Protocol (BGRP)[21] for aggregate flows destined to a certain domain – a Sink-Tree-Based Aggregation Protocol, is too static and therefore fails to efficiently utilize the network resources. Besides, the Dynamic Aggregation of Reservations for Internet Services (DARIS) [2] over-reserves bandwidth for aggregate flows over several domains.

However, DARIS focuses on reservation aggregations and signaling performances to improve system scalability. The Simple Inter-Domain QoS Signaling Protocol (SIDSP)[23] system over-provisions the so-called virtual trunks of aggregate flows. However, bandwidth over-reservation based on predictive algorithm (e.g. based on past history) without any mechanism to dynamically control the shared resource between existing trunks is not efficient, as it can lead to waste of bandwidth.

The recently patented Multi-user Aggregated Resource Allocation (MARA)[19] distinguished itself from the previous solutions by dynamically configuring and reconfiguring bandwidth over-reservation parameters for CoSs. Moreover, MARA deals with wasting bandwidth by attempting to grant a congested CoS with a portion of residual bandwidth over-reservation of remaining classes, taking into account current resource demand and session requirements.

However, MARA confronts serious efficiency problems in its over-reservation control mechanism, by wasting bandwidth especially when the network is near to congestion, while the signaling load is not too minimized. Moreover, MARA does not provide any information on how this approach could work in dynamic scenarios with unpredictable cross-traffic such as in decentralized networks.

As described in [14], communication paths happen to correlate by sharing link(s), meaning that a CoS on a shared link is used by all paths on which it lies, while traffic loads in different paths are unpredictable.

Hence, the dynamics of bandwidth utilization in various CoSs on shared links make over-reservation approach very challenging. In other words, an efficient dynamic bandwidth over-reservation mecha-

nism strongly requires appropriate functions to:

(a) take communication paths correlation patterns and cross traffic into account;

(b) compute appropriate bandwidth to over-reserve for each Class of Service (CoS);

(c) deal with residual bandwidth (reserved but unused) to reduce the waste of bandwidth and prevent CoS starvation.

The COR scheme introduced novel techniques to over-allocate bandwidth to CoSs, in a way that avoids QoS violation while significantly minimizing the waste of bandwidth and the session blocking probability.

In this Dissertation, the capabilities of COR and A-COR algorithms are evaluated in both centralized and decentralized architectures through the Network Simulator, version 2, or NS-2[20]. The evaluation is performed in comparison with MARA, which represents the stat of the art.

2.4 Network Decentralization Control

The increasing dependence on IT technologies is making central controllers bottlenecked while the networks are subject to frequent failures. The performance aspects of centralized mechanisms are shortcoming, mainly in terms of system scalability, robustness and availability.

Today, most of the major architecture designs are centralized-based such as TISPAN, 3GPP and even the European Context Casting C-CAST project despite their well-known scalability and single point of failure problems.

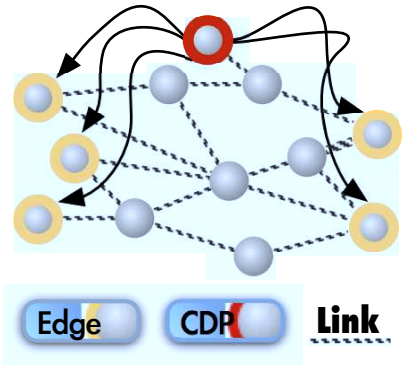


Figure 2.3: Centralized Network Diagram

Usually there is one central node (Control Decision Point (CDP)) that is responsible for taking all the decisions, and communicating the results to the edge nodes, for enforcement. The increase in the network complexity, being more information and larger databases to be processed leads to scalability and processing load issues.

As an alternative, decentralized approaches fit better in large-scale environments than centralized solutions. Decentralization allows the support of simultaneous operations at entities throughout a network, achieving better performance. However, it requires the synchronization of control information of multiple distributed decision points to avoid wrong and incompatible decisions.

Today, the inexistence of standards for decentralization forces each distributed system to deploy its own control strategies. As a result, Internet's complexity increases even more by the addition of new protocols and mechanisms. More importantly, the synchronization signaling overheads pose serious problems in existing decentralization designs such as in peer-to-peer networks [11], in Ad hoc networks [6], etc.

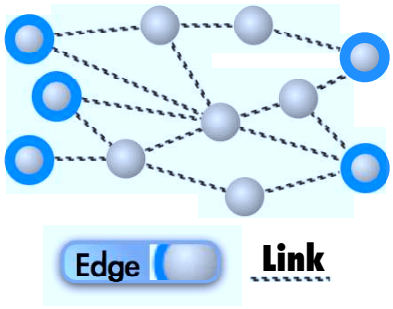


Figure 2.4: Distributed Network Diagram

This is due to the fact that exceeding control signaling consumes too much resource in terms of bandwidth, memory, CPU and energy. Therefore, decentralization must be correctly designed to prevent network performance degradations.

In the literature today, it is widely researched that knowledge of network topology and links state information, mainly the available resources on bottleneck outgoing interfaces and their location on paths, are of relevant importance for researchers and Internet Service Providers (ISPs) to improve system overall performance.

Indeed, current designs mostly enable a single control decision point in a network to maintain knowledge of underlying network topology in terms of nodes, paths bottleneck interfaces capabilities, list of on-path outgoing interfaces, and so on.

In this sense, the Self Organizing Multiple Edge Nodes (SOMEN) [13] work, introduced a new decentralization control approach which effectively allows keeping low synchronization control load while assuring multiple distributed Control Decision Points (CDPs) to cooperate and self-manage to optimize the network overall performance. Perhaps more importantly, the solution provides effective support for aggregate resource over-reservation in dynamic and distributed network environments.

Hence, this Dissertation implements three major aggregate over-provisioning schemes such as MARA [14], COR [14] and Advanced Class-based bandwidth Over-Reservation (A-COR) in the SOMEN architecture and evaluate the control performance mainly in terms of signaling load, waste of resources and session blocking issues in decentralized scenarios.

Chapter 3

Architecture

3.1 Introduction

This chapter introduces the different control mechanisms evaluated in this dissertation. The work provides a mechanism for the support of decentralized network control operations for the current and future Internet, the Self Organizing Multiple Edge Nodes (SOMEN). In this chapter, the SOMEN protocol will be presented and explained, and the control mechanisms will be described, applied in both centralized and distributed SOMEN architecture.

SOMEN was designed to operate in a distributed fashion, but a modified version of the protocol was later adapted to allow it to operate in a centralized manner. Both approaches are described, with an overview in Sections 3.3 and 3.2 and a more detailed operation can be found in Sections 3.6 and 3.7.

The main functionalities used by SOMEN are described in Section 3.4. Moreover, we present and discuss the two reservation algorithms COR and A-COR. The MARA is not a new proposal, and hence it is not explained in detail.

There are however, some important key words and definitions that the reader should be familiar with. For instance, all network scenarios take for granted some configurations. We consider that the network is composed of a certain number of nodes that fall within three distinct categories:

Ingress Nodes: Network border nodes where packets can be injected into the network.

Egress Nodes: Network border nodes where packets can leave the current network, e.g. to another Autonomous System.

Core Nodes: All nodes except the two types above. They forward packets, enforce QoS parameters and make possible the existence of multiple paths.

The network implements a Class of Service (CoS) based system. The number of classes may vary from three to seven for different network scenarios, but must be constant within the same network. An example network is given using 4 CoSs. Packet flow is generated by the means of well formulated requests that can come at any given time, for any given *Ingress* node only.

The parameters a request must have are the following:

Ingress The point of entry in the network.

Egress The destination

BW Amount of bandwidth (BW) to use

CoS The required Class of Service (CoS)

In this work, a multicast channel is configured per path in order to force every packet (merged into a path) to follow the same path associated.

Section 3.8 presents a summary of the chapter.

3.2 Distributed Architecture Overview

Self Organizing Multiple Edge Nodes (SOMEN) operation consists of enabling multiple distributed network Control Decision Points (CDPs) that dynamically exploit network appropriate context information to infer accurate cross traffic loads in each Class of Service (CoS) on every outgoing interface within a network without probing the related communication paths. This way, every CDP is able to obtain detailed information about each outgoing interface within a network with low signaling overheads. The network context information of interest includes, but not limited to, paths correlation patterns, network topological information (e.g. list of on-path outgoing interfaces, etc.) and traffic load dynamics in various CoSs on the existing paths within the network.

For this purpose, SOMEN introduces several functions that enable CDPs (e.g. network borders nodes), operating in a decentralized manner, to cooperate, exchange appropriate control information, and quickly adapt to changes in network and the related resource states, keeping low signaling load. These functions include:

(a) **Dynamic Filtering Functions DFFs:** which allow each CDP to minimize the decentralization control signaling load by filtering not only the information to advertise to other cooperating CDPs, but also by advertising only the CDPs which are really interested in the information to be advertised (the correlated CDPs);

(b) **Dynamic Threshold-based Synchronization Functions DTSFs!s:** based on which, each CDP significantly minimizes the decentralization control signaling load by means of dynamically controllable threshold (VOPR) which allows avoiding per-flow or per-change advertisement (a.k.a. synchronization) messages exchanges between the CDPs.

(c) **System Resilience Functions SRFs:** with the purpose of assuring system availability and reliability in presence of network dynamics such as nodes or links failures. These functionalities are not implemented in this Dissertation and therefore will not be described later.

The SOMEN operations are implemented by two control agents: SOMEN Edge (SOMEN-E) and SOMEN Core (SOMEN-C) agents. SOMEN-E is a state-full agent deployed by CDPs and implements the main SOMEN functions such as the DFF, the DTSF and the SRF functions. SOMEN-C is a lightweight-state agent deployed by control decision enforcement points (e.g. core routers), and simply implements appropriate functions to properly interpret and process control signaling messages

(e.g. resource control, message forwarding, etc.). Thus, SOMEN pushes control complexity to CDPs and leaves other nodes simple for scalability purposes.

In Figure 3.1 nodes *0, 1,* and *2* would be configured as SOMEN-E agents, while interior routers configured as SOMEN-C agents. Note that this figure will serve as an example network for the description of both the distributed and the centralized SOMEN mechanisms. The link connecting node 8 to the network is a dashed line because in the distributed scenario, node 8 doesn't exist, the control is not centralized. In the centralized scenario this node is the responsible for all key decisions, as the next section will introduce.

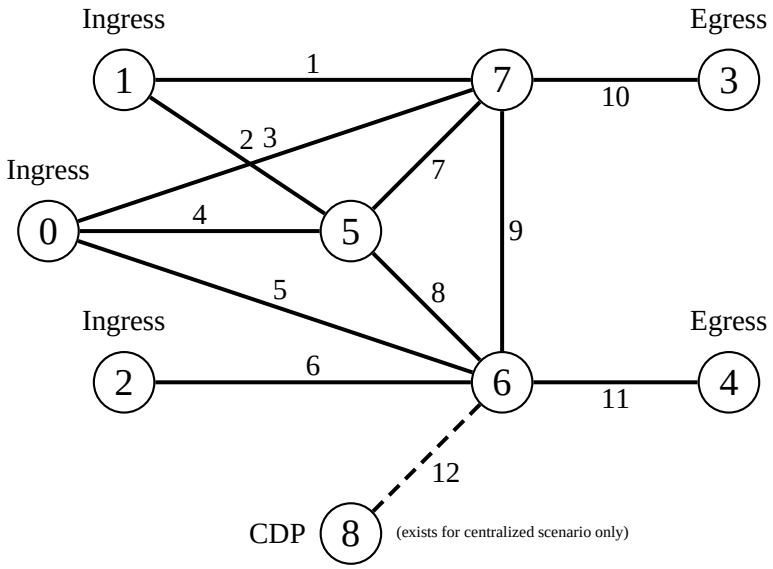


Figure 3.1: Example 1 Network Topology

3.3 Centralized Architecture Overview

In a centralized scenario, there is only one node that contains the complete information about the whole network. This node is considered to be the Control Decision Point (CDP). In Figure 3.1 we have a possible network topology were the CDP is running the centralized version of SOMEN.

For this purpose, the CDP implements (partially) the Dynamic Filtering Functions (DFF) mentioned in the previous section. There is no need for this node to have the Dynamic Threshold Synchronization Functions (DTSF) since these functions have been designed especially to support distributed network control. In a centralized scenario, the CDP has on a real time basis the complete information about every link, and CoS in the network and can admit or deny requests directly based on available BW.

We say that it implements the filtering functions partially, because it does not need all of them. Some of these functionalities include techniques to lower message exchange when the *Ingress* nodes need to exchange context information. This exchange of information does not exist in a centralized scenario.

3.4 Functionalities

In this section it is explained the functionalities of the Network Context Information dataBase (NetCIB) and how the tables are constructed. The NetCIB is the database used by the SOMEN protocol. It is composed of six main tables:

- **Global Paths:** all the known Paths to all *Egresses*
- **PATHS:** selected Paths to admit requests
- **CORRELATIONS:** Paths that share links with one another
- **TOPOLOGY**: all links from all Paths along with respective QoS requirements (used, available, reservation BWs etc.)
- **VOPRS:** respective Virtual Over PRovisioning (VOPR) values for all CoSs, all links, for each Path
- **LISTS:** CDP's addresses that should be advertised about changes for each Path
- **SESSIONS:** flows or requests already admitted and the relevant information stored (path selected, BW, mapped CoS, etc.)

3.4.1 Dynamic Filtering Functions (DFF)

The SOMEN DFF functions are responsible for enabling each of the distributed CDPs to maintain a good knowledge of the network paths correlations patterns, the network topology and the related resource status.

In order to obtain a good knowledge of the underlying network topology, at system initialization when no session is running yet, each CDP deploys appropriate signaling protocol, creates many edge-to-edge paths, and maintains the best ones for service delivery.

The best paths selection can base on, but not limited to, the delay, the number of hops and the available bandwidth on the created paths, which can be defined according to specific design purposes. Afterwards, the CDPs exchange appropriate context information about the selected paths (e.g. path IDs, list of outgoing interfaces on the paths). Thus, each distributed CDP is allowed to build its own control database called Network Context Information dataBase (NetCIB) to store and maintain the network topology, the resource status in each CoS on each outgoing interface, the paths correlations patterns, and so on.

A flooding-based technique is used for the paths creation, during system initialization phase. Each Ingress node composes control signaling messages, well flagged for the network initialization, and sends them throughout all its own interfaces. Every SOMEN-C agent will save its address in the header and forward these messages out all its interfaces, only if it's address is not already there, meaning that a loop has occurred and the packet is discarded. When one of these packets reaches an *Egress* node, the later will build and send to the respective *Ingress*, a response message advertising a new path discovered.

This way, after a few seconds, every CDP is able to obtain all the possible paths from itself to any other CDP within the network. All the paths that have been discovered are saved into a table in each CDP local database. This table is called the PATHS Table.

As these initialization messages flow, the QoS requirements are installed on every link for all CoSs defined. A share of bandwidth (the initial reservations) is reserved for each CoS accordingly to the reservation algorithm defined. These algorithms are discussed later in Section 3.5.

The QoS parameters taken into account are:

- χ Reservation threshold
- B Reserved Bandwidth
- U Used bandwidth
- Av Available bandwidth ($B - U$)

After the system initialization is over, all CDPs have the information about all the possible paths to all Egress nodes, and all the initial over-reservation parameters are implemented in the network links.

Notice however that this information is still local, for instance, node 0 knows nothing about paths node 1 has discovered. This information needs to be exchanged. In order to do so, each CDP advertises all others with information about all of its own selected paths. The advertisement messages includes the selected paths' IDs, the list of the IDs of the outgoing interfaces that compose each path and the capacity of each outgoing interface. Thus, upon receiving the advertisement messages from others, every CDP is able to build it's own local SOMEN Network Context Information dataBase (NetCIB).

The NetCIB of a CDP is composed of appropriate tables such as the *CORRELATIONS*, the *TOPOLOGY*, the *PATHS*, the *VOPRS*, the *LISTS* and the *SESSIONS* tables, introduced by SOMEN functions to store network key control information in a way to effectively improve system performance with low control load.

To facilitate the understanding of the DFF functions for one CDP, the Ingress node 1 in Figure 3.1 is used. Consider that the network has 4 CoSs implemented, being one of them saved for protocol specific control signaling messages leaving 3 CoSs available for data traffic.

After initialization phase, the respective *PATHS* Table for node 1 would be Table 3.1. Each path is described here as being a list of nodes that compose it (last column). For each class there is also the initial QoS parameters defined. The *U,A,V* refer respectively to the used, available and VOPR BWs values, on the bottleneck link of the path. Each path is given an unique path ID, that is used by the protocol to identify one specific path. This way, the control messages can carry only this number which minimizes overhead. The first column identifies the destination of the path, the node it goes to.

By saving the used BW every CDP is able to record the accurate amount of bandwidth being used by its active sessions (sessions running through itself) in each CoS on each of its selected paths. Besides, the available bandwidth parameter in *PATHS* table, represents the total amount of bandwidth reserved but unused in CoS_i on the bottleneck outgoing interface of the path P_n i.e. the minimum available bandwidth in the path. Further, the Virtual Over PRovisioning (VOPR) parameter in the PATHS table is a dynamically controllable threshold of a CoS_i on the bottleneck outgoing interface of a path P_n. More details on how the available bandwidth and the VOPR parameter are maintained in the *PATHS* table of a CDP are provided later in the document.

Besides the *PATHS* table, node 1 builds its *TOPOLOGY* table to store the links and respective detailed QoS parameters for each CoS. This table is very important once the *PATHS* table can only support information about the bottleneck link, *TOPOLOGY* has information for all the links.

The *TOPOLOGY* table for node CDP_1 is as Table 3.2 represents. For each known link, we store the capacity and for each CoS store all the parameters to ensure QoS. These values come from the

respective reservation algorithm being used and correspond to the initial reservation values. *B,U,A,X* mean respectively reserved, used, available, threshold BW values. Notice that *X* does not exist if the reservation algorithm is A-COR.

The *CORRELATIONS* table can be built based on the previous ones, the *TOPOLOGY* and *PATHS* Tables. The algorithm will check for all the paths that use common links, and save that information, building the table. It is generated by running the algorithm 3.4.1.

The *CORRELATIONS* table is used to store the paths correlation patterns for each of the interfaces that compose its own selected paths. In the rest of this document, the interfaces composing a CDP's own selected paths are called own interfaces.

In this sense, the *CORRELATIONS* table of node 1 shown in Table 3.3, stores the following control information: Link IDs, sharing factor (W) and a list of paths. For each link in the *TOPOLOGY* table, we have a list of paths that use that link. The total number of paths that use it is called the sharing factor (W). This away each CDP has a way to identify which information needs to be updated if a requests is mapped in a given path.

Algorithm 3.4.1: BUILD CORRELATIONS TABLE($void$)

---

comment: For all Paths

for $j \leftarrow 1$ **to** *number of paths*

do
- get $Path_j$ from *PATHS* Table
- **comment:** For all On Path Links
- **for** $i \leftarrow 1$ **to** *number of links*
  - **do**
    - get $Link_i$ from *TOPOLOGY* Table
    - **if** link L_i belongs to $Path_j$
      - **then** record P_j ID and the corresponding CDP address

---

Furthermore, the *LISTS* Table shown in Table 3.4 is built based on the *PATHS* and *CORRELATIONS* Tables. The *LISTS* table stores for each own selected path P_j, a list of the CDPs whose paths happen to correlate with the path P_j. Hence, the *LISTS* table, allows each CDP to easily retrieve the correlated CDPs to advertise about changes in path upon need. This way, the SOMEN DFF functions prevent broadcasting advertisement messages to all CDPs unnecessarily.

Algorithm 3.4.2: BUILD LISTS TABLE($void$)

---

comment: For all Paths

for $j \leftarrow 1$ **to** *number of paths*

do
- get list of links from $Paths_j$
- **comment:** For all Links in the list
- **for** $i \leftarrow 1$ **to** *number of links*
  - **do**
    - get all *Paths* that use $link_i$ from *CORRELATIONS* table
    - **for** $k \leftarrow 1$ **to** *number of Paths*
      - **do** {store this *Path Ingress* ID to a list – *Ig_list*
- Remove duplications from the *Ig_list*
- Associate this *Ig_list* to the current $Path_j$ as a new entry in the *LISTS* table

---

#	Ig-Eg	Path ID	CoS 1 (U:A:V)	CoS 2 (U:A:V)	CoS 3 (U:A:V)	Path Links
0	1-3	191	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 7 \rightarrow 3$
1	1-4	207	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 5 \rightarrow 6 \rightarrow 4$
2	1-3	227	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 3$
3	1-3	199	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 5 \rightarrow 7 \rightarrow 3$
4	1-4	203	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 7 \rightarrow 6 \rightarrow 4$
5	1-4	216	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 5 \rightarrow 7 \rightarrow 6 \rightarrow 4$
6	1-4	221	0:1.65:0.165	0:1.65:0.165	0:1.65:0.165	$1 \rightarrow 7 \rightarrow 5 \rightarrow 6 \rightarrow 4$

Table 3.1: PATHS Table CDP_1

#	Link ID	Capacity	CoS 1 (B:U:A:X)	CoS 2 (B:U:A:X)	CoS 3 (B:U:A:X)
0	0-5	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
1	0-6	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
2	0-7	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
3	1-5	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
4	1-7	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
5	2-6	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
6	5-6	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
7	5-7	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
8	6-4	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
9	6-5	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
10	6-7	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
11	7-3	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
12	7-5	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30
13	7-6	10	1.65:0:1.65:3.30	1.65:0:1.65:3.30	1.65:0:1.65:3.30

Table 3.2: TOPOLOGY Table CDP_1

#	Link ID	W	List of Correlated Paths (Ingress,Path ID)
0	0-5	4	(0,156) ; (0,123) ; (0,97) ; (0,132)
1	0-6	3	(0,69) ; (0,92) ; (0,101)
2	0-7	3	(0,88) ; (0,117) ; (0,73)
3	1-5	4	(1,199) ; (1,227) ; (1,207) ; (1,216)
4	1-7	3	(1,191) ; (1,203) ; (1,221)
5	2-6	3	(2,270) ; (2,279) ; (2,283)
6	5-6	6	(0,105) ; (0,117) ; (0,132) ; (1,227) ; (1,207) ; (1,221)
7	5-7	6	(0,123) ; (0,97) ; (0,101) ; (1,199) ; (1,216) ; (2,283)
8	6-4	10	(0,69) ; (0,88) ; (0,105) ; (0,117) ; (1,123) ; (1,207) ; (1,203) ; (1,216) ; (1,221) ; (2,270)
9	6-5	2	(0,101) ; (2,283)
10	6-7	4	(0,92) ; (0,132) ; (1,227) ; (2,279)
11	7-3	10	(0,73) ; (0,92) ; (0,97) ; (0,101) ; (0,132) ; (1,191) ; (1,199) ; (1,227) ; (2,279) ; (2,283)
12	7-5	2	(0,117) ; (1,221)
13	7-6	4	(0,88) ; (0,123) ; (1,203) ; (1,216)

Table 3.3: CORRELATIONS Table CDP_1

#	Path ID	List of CDPs to advertise
0	191	0,2
1	207	0,2
2	227	0,2
3	199	0,2
4	203	0,2
5	216	0,2
6	221	0,2

Table 3.4: LISTS Table CDP_1

In this example CDP_1 has to advertise both CDP_0 and CDP_2 for all paths, but this may not be the case for other network topologies. This happens because there are few core nodes which make for a strong paths correlation. If we consider a larger network with more core nodes the *LISTS* table would have different lists of nodes for different paths, but the number of paths would grow very fast, and this example would become complicated to present due to the large size of the *TOPOLOGY* and *PATHS* tables.

In order to further minimize the distributed network control overhead, SOMEN introduces the Dynamic Threshold Synchronization Functions (DTSF) to dynamically minimize the advertisement frequency in a way to assure a proper synchronization of the CDPs with low control messages overheads. The DTSF functions are detailed in the subsequent sub-section.

It is important to recall that the initial SOMEN NetCIB database, is created at the system initialization phase when no session is running yet. Therefore the related processing load is not seen as a burden in the network. More importantly, the NetCIB is designed in order to allow keeping low control processing and maintenance load during the network operation phase.

3.4.2 Dynamic Threshold Synchronization Functions (DTSF)

The SOMEN DTSF introduce a dynamically controllable function called Virtual Over PRovisioning (VOPR), aiming to prevent per-flow or per-change advertisement messages between the distributed CDPs. By deploying the DTSF functions, the CDPs are able to dynamically minimize the synchronization signaling load in a network.

The basic idea of VOPR is to enable every CDP within a network to dynamically control a certain amount of resources, the virtual resources, for each CoS on the bottleneck outgoing interface of each of its own selected path. This is done in a way that allows every CDP to admit new sessions/requests, to readjust the QoS requirements or to terminate an active session in a CoS on any of its own selected path without being required to advertise others as long as the associated VOPR threshold is not exhausted.

The VOPR value is computed accordingly to equation 3.1 for each link, for each CoS within the network. All values for VOPR are stored in the *VOPRS* Table and similar to other tables, each CDP has it's own. This table can be built by running algorithm 3.4.3. An example of the initial table for node 1 of network in Figure 3.1 can be found in Table 3.5. The *VOPR* Table is built by computing the VOPR values for each CoS for all links on $Path_i$ for all possible Paths.

$$vopr = U + \frac{Av}{W} \tag{3.1}$$

Where U and Av are respectively, the current Used (at system initialization is zero) and Available BWs for the given link with *sharing factor* W – number of Paths that use the link. These variables

can be collected by accessing the local *TOPOLOGY* and *CORRELATIONS* tables.

Algorithm 3.4.3: BUILD *VOPR* TABLE($void$)

comment: For all Paths in *PATHS* Table

for $j \leftarrow 1$ **to** *number of paths*
do
- get list of links from $Paths_j$
- **comment:** For all Links in the list
- **for** $i \leftarrow 1$ **to** *number of links*
- **do**
  - get $link_i$
  - **for** $k \leftarrow 1$ **to** *number of CoSs*
  - **do**
    - compute and store new VOPR value for $link_i$, CoS_k in *VOPR* table
    - **comment:** VOPR computed accordingly to equation (3.1)

#	Path ID	On-path Links and Related VOPRs (CoS 1 to 3)
0	191	1-7(0.550 : 0.550 : 0.550) → 7-3(0.165 : 0.165 : 0.165)
1	199	1-5(0.412 : 0.412 : 0.412) → 5-7(0.275 : 0.275 : 0.275) → 7-3(0.165 : 0.165 : 0.165)
2	227	1-5(0.412 : 0.412 : 0.412) → 5-6(0.275 : 0.275 : 0.275) → 6-7(0.412 : 0.412 : 0.412) → 7-3(0.165 : 0.165 : 0.165)
3	207	1-5(0.412 : 0.412 : 0.412) → 5-6(0.275 : 0.275 : 0.275) → 6-4(0.165 : 0.165 : 0.165)
4	203	1-7(0.550 : 0.550 : 0.550) → 7-6(0.412 : 0.412 : 0.412) → 6-4(0.165 : 0.165 : 0.165)
5	216	1-5(0.412 : 0.412 : 0.412) → 5-7(0.275 : 0.275 : 0.275) → 7-6(0.412 : 0.412 : 0.412) → 6-4(0.165 : 0.165 : 0.165)
6	221	1-7(0.550 : 0.550 : 0.550) → 7-5(0.825 : 0.825 : 0.825) → 5-6(0.275 : 0.275 : 0.275) → 6-4(0.165 : 0.165 : 0.165)

Table 3.5: VOPRS Table CDP_1

This table has the Path ID and the on path links with the respective VOPR values for each CoS separated by colons. Thus, by using the *VOPRS* table, a CDP obtains the VOPR of every CoS on the bottleneck interface of each own selected paths, being the minimum VOPRs of the CoS among the interfaces that compose the path, and stores it in its *PATHS* table.

Whenever a CDP_j receives an authorized session request to a CoS_i on a path P_j, it locally checks the resource availability in the CoS_i on the path P_j using its *PATHS* Table. In particular, it uses the VOPR of the CoS_i on the bottleneck interface of P_j and the respective used bandwidth. If the sum of the used BW and the request BW is not above the value of VOPR, it is said that VOPR is available, otherwise is is said that VOPR is exhausted.

Then, when the CDP realizes that the VOPR of the CoS_i on the bottleneck interface of the path P_j is available, it simply processes the request and updates its local NetCIB accordingly, without issuing any advertisement message to the correlated CDPs. Only when VOPR exhausts, there is an advertisement event.

This means that the CDP would admit a new session, readjust the QoS requirements, or terminate a session in a path P_j without triggering any advertisement messages to the correlated CDPs of the path provided that the related VOPR is available. In other words, the DTSF functions provide means

to effectively hide the dynamics of sessions and network resource states changes from the CDPs. Thus, SOMEN allows avoiding per-flow or per-change synchronization signaling messages while assuring the accuracy of the databases. Therefore, the advertisement signaling load can be significantly reduced, depending on the values of the VOPRs, the used bandwidth and the demands of incoming sessions or requests. This approach is promising for a better control of the future Internet which envisions a ever growing network capacity.

3.4.3 Flow Re-Routing

The SOMEN mechanism envisions a way to balance traffic load among the several paths. This mechanism is called flow re-routing and is explained in this section. In both centralized or distributed scenarios the algorithm is the same, note only that in SOMEN decentralized networks, the available bandwidth of a CoS in a path should be seen as the VOPR of the CoS in the path.

When all paths are congested, meaning that none of them meets the requirements of the request r, then the flow re-routing is triggered. The basic idea of the flow re-routing here is to efficiently re-route the running flows between the paths such that requests can be admitted as long as possible, in a way that maximizes system throughput as well as resource utilization efficiency.

To help on understanding this re-routing algorithm, a path from which certain flows are re-routed is called *mainPath* and those into which flows are mapped are called *assistingPaths*. Moreover, let u_i be the used bandwidth of a flow i and av_j be the available bandwidth in a given path j. A flow chart in Figure 3.2 represents the steps that compose the algorithm, and the enumeration below explains in detail flow re-routing.

1. Sort all the paths between A and B in a decreasing order of their available bandwidth, and index the paths as P_1, P_2, ..., P_n with n the total number of the paths. This set of paths is stored in a table called *SPT* – Selected Paths Table. Then, go to **(2)**

2. Select the path P_i from *SPT* which has the highest available bandwidth (e.g. P_1 at first iteration) as *mainPath*. All other paths of *SPT* are *assistingPaths* and are copied into a second table called *APT* – Assisting Paths Table. Then, go to **(3)**

3. Update a variable called *mainList* so that is contains a list of all the active flows in the *mainPath* (flows that belong to the requested CoS_k). These information can be found using the *SESSIONS* table. Then, filter the *mainList* by removing the flows whose used bandwidth u_i is superior or equal to the available bandwidth of the *assistingPath* which has the highest available bandwidth among the *assistingPaths* (e.g. P_2 at first iteration). These are the flows that cannot be re-routed because there is not enough resources in the *assistingPaths* to accommodate them. So, if the *mainList* is empty, go to **(5)**, else, go to **(4)**

4. Select the flow which has the highest used bandwidth u_i in the *mainList* as a Candidate flow to be re-routed from *mainPath* into the *assistingPath* which currently has the highest available bandwidth. Then, bind this flow to that *assistingPath* and delete the former from the *mainList*. Compute the sum S of all the selected Candidate flows for re-routing from *mainPath* to the *assistingPaths*. If the sum of S and the available bandwidth av of the *mainPath* is higher or equal to the request ($S + av \geq r$), re-route the selected Candidate flows into their respective *assistingPaths* and admit the request r into the *mainPath*. Then, update the relevant context information in local database for all the correlated paths accordingly and end the admission process. The algorithm has been successful (box 6 in the flow chart 3.2).

Else, if ($S + a < r$), update the available bandwidth of the current *assistingPaths* taking into account the used bandwidth u_i of all the selected Candidate flows for re-routing (box 7). Temporary variables are used during the process to prevent intermediate changes from affecting the system (e.g. updating available bandwidth of *assistingPaths*). This means that the changes are applied only after the final decision. Then, go to **(3)**.

5. If $i \neq n$ with P_i the index of the current *mainPath*, then select the path with the next index ($P_i + 1$) as *mainPath* and go to **(3)**. Else, if $i = n$, meaning that there are no more thats to explore, return failure. The re-routing algorithm cannot find any possible re-routing that would be useful. The request should be processed according to local policies, that is, the request might be denied or mapped to a CoS with lower QoS requirements.

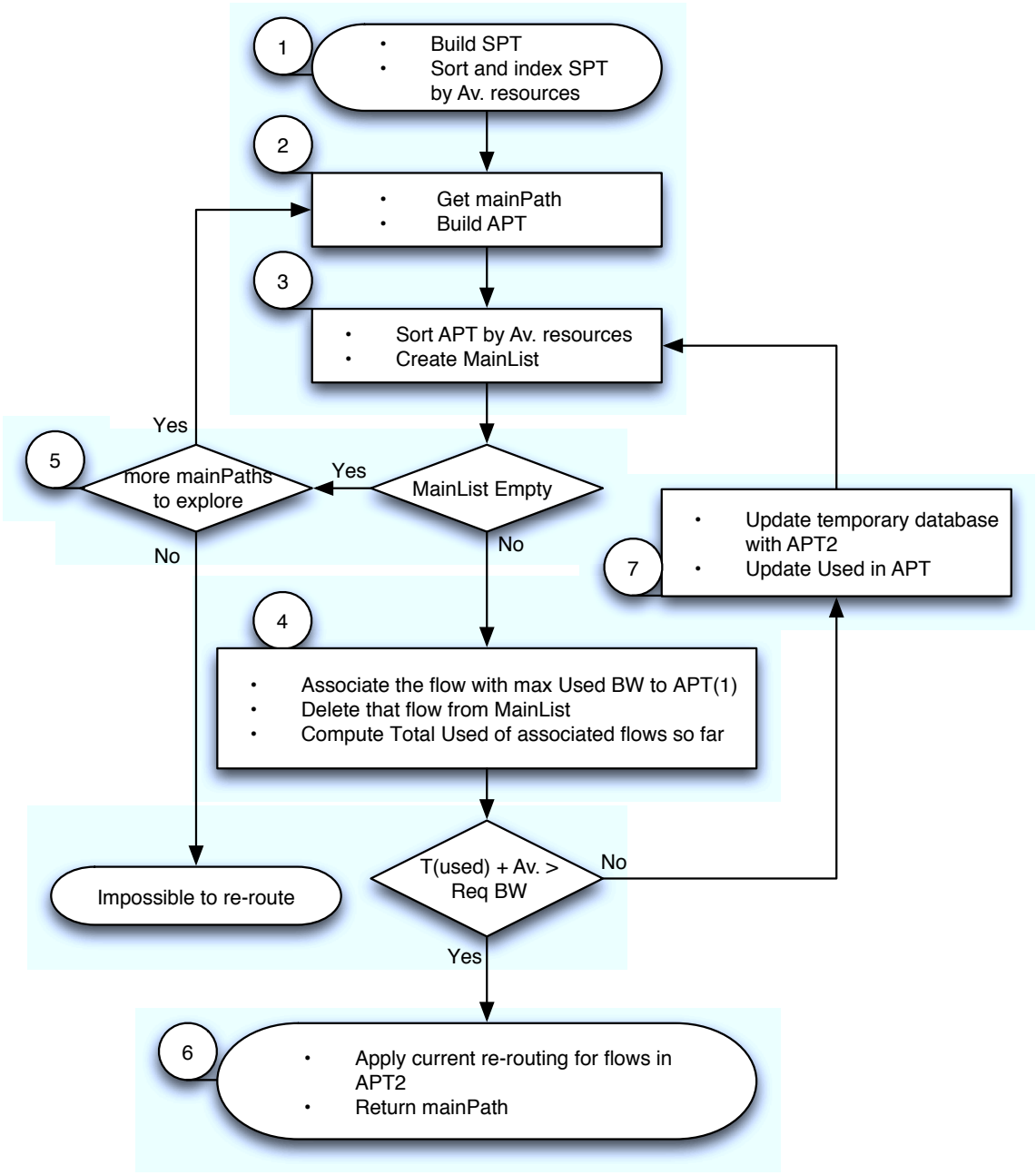


Figure 3.2: Flow Re-Routing flow chart

3.5 Bandwidth Over-Reservation Algorithms

The main objective of this section is to present the resource over-reservation algorithms that were implemented in the architectures (centralized/distributed) described earlier in this chapter. This reservation scheme includes three different Reservation Algorithms, namely, Class-based bandwidth Over-Reservation (COR), Advanced Class-based bandwidth Over-Reservation (A-COR) and Multi-user Aggregated Resource Allocation (MARA). The purpose, is to better evaluate the impact of different algorithms on a given network.

The first two, COR and A-COR, are the new proposals being evaluated in the simulation process in comparison with MARA, which represents the state of the art. Therefore, only COR and A-COR are clearly described here, as details about MARA can be found in the literature [19].

As discussed in section 2.3, over-provisioning techniques envision reserving more bandwidth to each CoS than currently required, in order to minimize signaling messages and control load. This way, multiple flows can be accepted without signaling the network nodes, as long as resources are still available.

In order to facilitate the understanding of the description below, consider that the network has *k* CoSs configured on each link and a fraction of link capacity is assigned to each CoS. Such fraction of a CoS_i is denoted the weight α_i, with $1 \leq i \leq k$ such that:

$$\sum_{i=1}^{k} \alpha_i = 1 \tag{3.2}$$

One common hypothesis is to set all CoSs to the same weight, following:

$$\alpha_i = \frac{1}{k} \quad [1] \tag{3.3}$$

For simplicity reasons, the weight of each CoS_i is defined as equation 3.3 dictates.

3.5.1 COR

A flow chart following the operation of the algorithm is present in Figure 3.3 at the end of this section. To start explaining the algorithm, let's us go over the QoS parameters defined by COR for each CoS:

- χ Reservation threshold
- B Reserved Bandwidth
- U Used bandwidth
- Av Available bandwidth ($B - U$)

At system initialization phase, COR initializes the reservation B_i of each CoS_i with a minimum share of bandwidth, $1/2$ of the maximum reservation threshold χ_i of the CoS_i. Considering that a capacity C is destined to the k CoSs, COR initializes the maximum reservation threshold χ_i of each CoS_i according to its weight α_i as:

[1] Done during system bootstrap

$$\chi_i = \alpha_i \times C \tag{3.4}$$

So, the system administrator defines the α_i values, that will give the χ_i and the respective B_i values, for all CoSs.

During system running phase, whenever a QoS decision point (e.g. QoS Broker, ingress router, etc.) realizes that the reservation of a CoS_j ($1 \leq j \leq k$) is insufficient to admit a new request Req_j ($B_j < U_j + Req_j$) , where U_j is the used bandwidth in CoS_j, COR computes new bandwidth allocation patterns to be readjusted for all CoSs along the related path.

If the maximum reservation threshold of CoS_j is not exhausted ($\chi_j \geq U_j + Req_j$) , the new over-reservation γ_j is computed as in MARA by:

$$\gamma_j = \frac{U_j}{\chi_j}(\chi_j - U_j - Req_j) \tag{3.5}$$

If $(\widehat{B_j} + Req_j + \gamma_j < \chi_j)$, the new reservation parameter B_j of CoS_j is updated as deprecated in equation (3.6), otherwise accordingly to equation (3.7). Where $\widehat{B_j}$ is the old reservation to be updated for CoS_j.

$$B_j = \widehat{B_j} + Req_j + \gamma_j \tag{3.6}$$

$$B_j = \chi_j \tag{3.7}$$

However, when the maximum reservation threshold of CoS_j is exhausted $(\chi_j < U_j + Req_j)$, COR exploits the weights of each CoS and redistributes the total unused bandwidth on the link to all CoS_i, with $1 \leq i \leq k$, in a balanced way to better control congestion probability within the CoSs.

$$\Delta_T = \sum_{i=1}^{k}(\chi_i - U_i) \tag{3.8}$$

$$\eta_i = \alpha_i \times \Delta_T \tag{3.9}$$

Hence, the total unused bandwidth Δ_T on the link is computed by in equation (3.8) and the amount of unused bandwidth η_i to allocate to each CoS_i is obtained by equation (3.9).

If the bandwidth η_j to allocate to the CoS_j is sufficient to admit the new request ($Req_j \leq \eta_j$), the maximum reservation thresholds χ_i of all CoS_i for $1 \leq i \leq k$ are readjusted as:

$$\chi_i = U_i + \alpha_i \times \Delta_T \tag{3.10}$$

Consequently, the reservation B_j of CoS_j is updated as in equations (3.6) and (3.7), and the reservations B_i of other CoS_i with $i \neq j$ are readjusted accordingly to equation (3.11) if $\widehat{B_i} + Req_j + \gamma_i < \chi_i$, or else accordingly to equation (3.12). Where $\widehat{B_j}$ is the old reservation to be updated for CoS_j.

$$B_i = \widehat{B_i} + Req_j + \lambda_i \tag{3.11}$$

$$B_i = \chi_i \tag{3.12}$$

However, if the bandwidth η_j to allocate to CoS_j is insufficient to admit the request Req_j ($Req_j > \eta_j$), COR first compares the request Req_j to the total unused resource Δ_T of equation (3.8).

Hence, when $Req_j \leq \Delta_T$, Δ_T is allocated to CoS_j, and the session can be admitted to avoid wasting bandwidth and therefore avoid increasing session blocking probability.

The reservation and the maximum reservation threshold of CoS_j are updated as:

$$\chi_j = U_j + \Delta_T \qquad B_j = U_j + \Delta_T \tag{3.13}$$

$$\chi_i = U_i \qquad B_i = U_i \qquad (i \neq j) \tag{3.14}$$

The request is denied only if the total unused bandwidth on the link is not enough: $Req_j > \Delta_T$ There cannot be waste of resources.

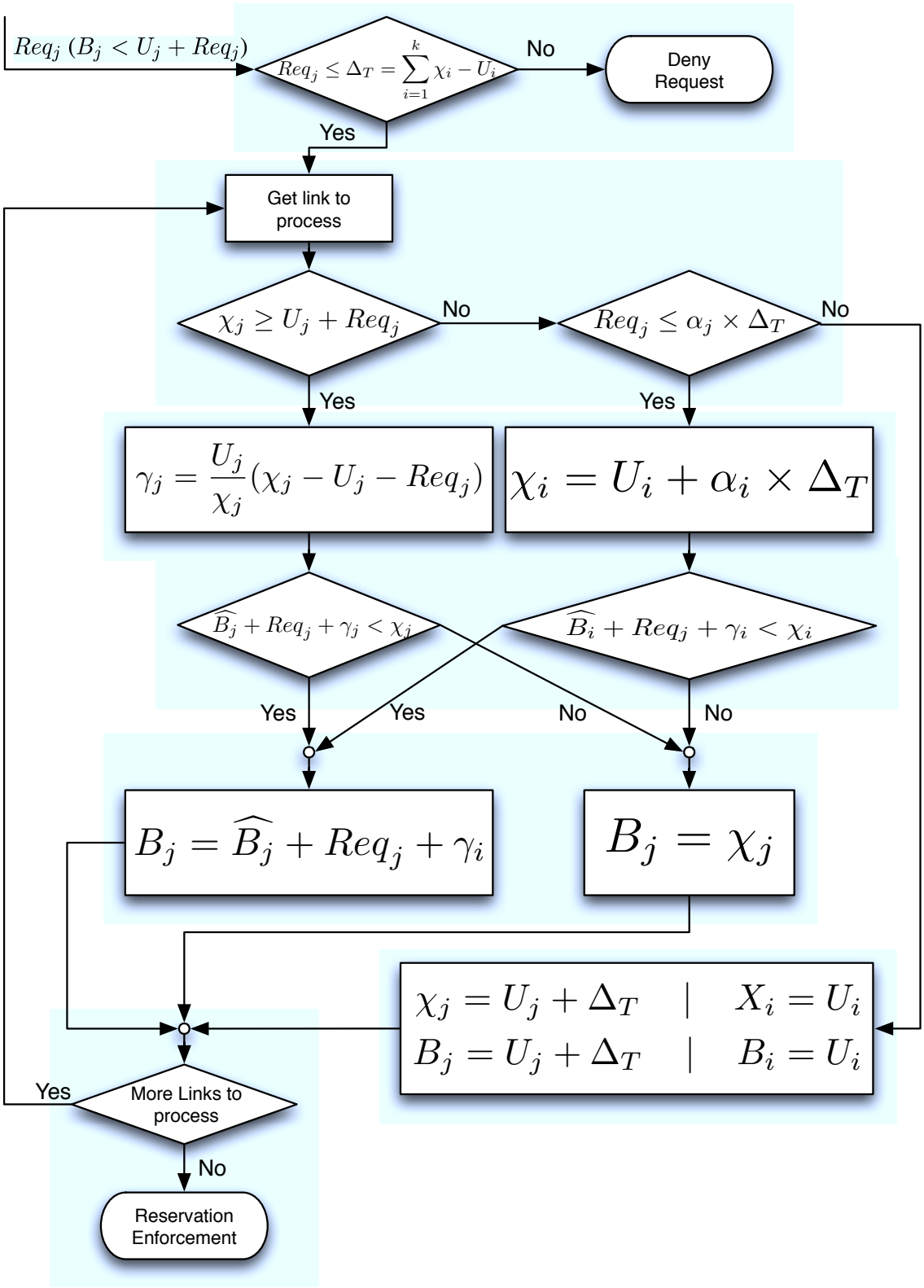


Figure 3.3: COR flow chart

3.5.2 A-COR

Advanced Class-based bandwidth Over-Reservation (A-COR) is an improvement of COR. As can be seen from the flow chart in Figure 3.4 it is considerable simpler as well. The purposes and the goals of both algorithms are the same, but A-COR proves to be better than COR. One of the main differences is that the χ (reservation threshold) is eliminated being all computations done based on B, the reserved bandwidth. So the the QoS parameters defined by A-COR for each CoS are the following:

- B Reserved Bandwidth
- U Used bandwidth
- Av Available bandwidth ($B - U$)

At system bootstrap, the startup values for B for all CoS_i are set as:

$$B_i = \alpha_i \times C \tag{3.15}$$

Where C is the link capacity, and α_i the weights set by the network administrator for each CoS. Once again the α values must sum to one, as equation (3.2) describes.

In A-COR the QoS requirements of Reserved Bandwidth (B_i), Used Bandwidth (U_i), Available bandwidth ($Av_i = B_i - U_i$) are maintained for each CoS_i ($1 \leq i \leq k$) on each network link, that belongs to a Selected Path.

In this sense, the total unused bandwidth for a given link (Δ_T) can be computed by equation 3.16 (similar to equation (3.8)):

$$\Delta_T = \sum_{i=1}^{k} (B_i - U_i) \tag{3.16}$$

Upon receiving a request for a given CoS_j, that cannot be accommodated immediately [2] the A-COR mechanism re-computes new Reservation values in an attempt to make it possible to admit the request.

It only makes sense to compute new Reservation values if the bottleneck link on Path P_n has enough unused BW (Δ_T) such that, $Req_j \leq \Delta_T$, otherwise it is physically impossible to admit the request.

In this case [3] the reservation values must change for all links (L_p) of the Path. To a given link L_p if the $Req_j \leq \alpha_i \times \Delta_T$ we compute new values with equation (3.17), otherwise with equation (3.19).

$$B_i = U_i + \alpha_i \times \Delta_T \tag{3.17}$$

$$B_j = U_j + Req_j + \alpha_i \times (\Delta_T - Req_j) \tag{3.18}$$

$$B_i = U_i + \alpha_i \times (\Delta_T - Req_j) \qquad 1 \leq i \leq k, (i \neq j) \tag{3.19}$$

[2] BW requested (Req_j) is higher than the minimum amount of Available BW min($Av_j; P_n$) for CoS_j, in all possible Paths - P_n

[3] $Req_j \leq \Delta_T$ for all links on P_n

The process is repeated for all L_p links on Path P_n. The new values are stored, and passed to be enforced on the network. The request can now be accommodated without signaling.

The request is denied only if the total unused bandwidth on the link is not enough: $Req_j > \Delta_T$ There cannot be waste of resources.

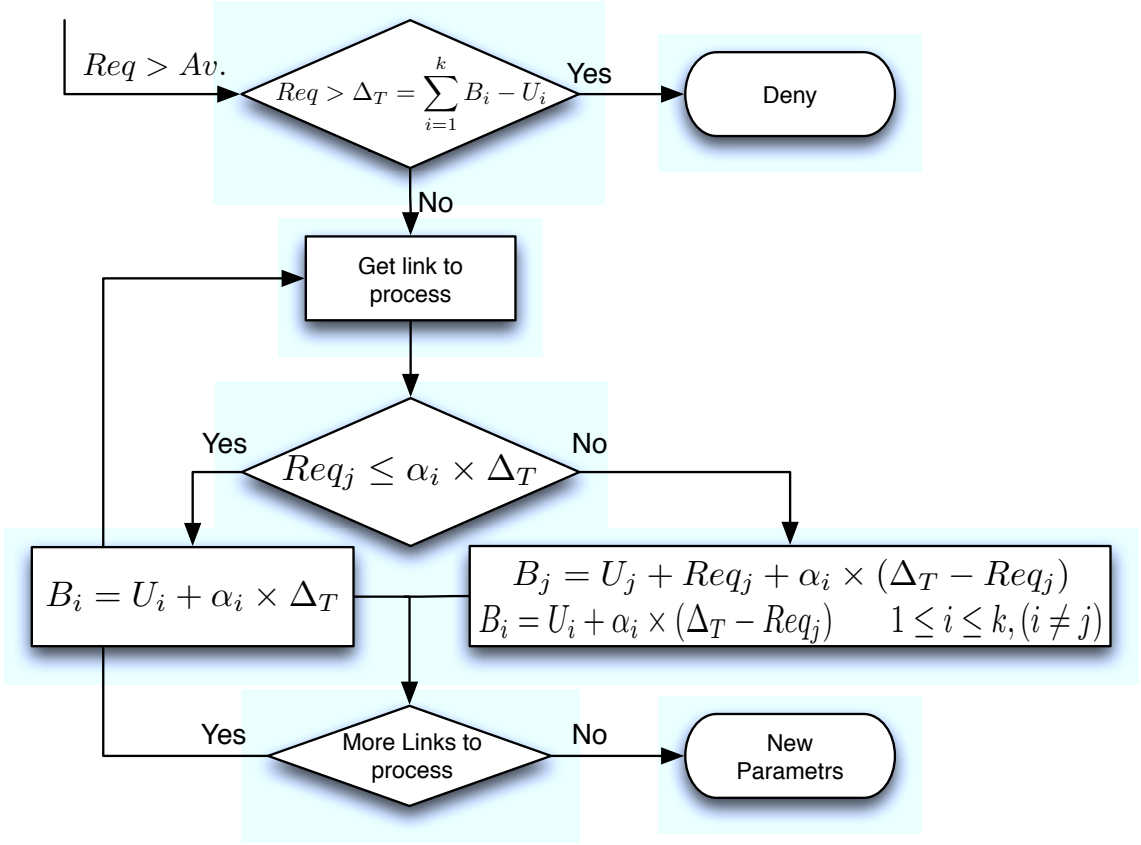


Figure 3.4: A-COR flow chart

3.6 Centralized Scenario

In this scenario there is only one CDP that is responsible for taking all the relevant decisions about what traffic comes into the network and what is denied, as well as to defined the respective QoS parameters and enforce them.

At system initialization, the network administrator has a strong influence on the booting up of the network. He must configure all nodes to implement the correct agents and define the CoS weights (α_i values for al CoS. At system bootstrap, when no session is running yet, the unique CDP triggers all the Ingress nodes (Ig) under its control to create edge-to-edge paths to any Egress nodes (Eg). This triggers the flooding mechanism briefly described in Section 3.4.1.

The objective is to find all possible paths between each *Ingress* and each *Egress* on the network as well as to discover all relevant topology information, for instance, every link's delay and capacity. The flooding mechanism was originally used in the MARA implementation, but it was adapted for this work and so the final C++ code is different from the one in the original version. It works as in the following.

Each Ingress node, sends out of each one of it's own interfaces a well flagged message that will be forwarded by all nodes. At the same time paths are being discovered, initial resources are being reserved for each CoS at each link in the network. This message is responsible for triggering the initial reservation of resources for all CoSs (accordingly to the reservation algorithm defined by the network administrator) for all links it goes trough. Each node ID is stored in a vector (*Path vector*), that will later become the new path discovered.

Upon receiving such a message each type of node takes different actions. In the case the current node is an *Ingress* node, the received packet is discarded, meaning that a loop on the network has occurred. In the case that this node is a *Core* node it looks into the *Path vector* contained in the message. If it doesn't find its own address there, then adds it, and forwards the message out all of it's own interfaces, except the interface at which the message was received. It also fulfills the respective data fields for links bandwidth and delay with the correct values (not done in the initial MARA implementation). This way the information about the network topology is gathered. If it finds its own address there, this means a loop has occurred and the message is discarded, avoiding devastating loops. The last possibility is that the receiving node is an *Egress* node. In this case, a response message is built and all the information gathered copied, and sent back to the *Ingress* node that generated it. The response follows the exact path done in the reverse order, to correctly build the multicast tree, using source routing.

When a response message reaches the *Ingress*, it is forwarded to the CDP node that will store the information. After a given time, all the possible responses have been received, meaning that all links and CoSs initial reservations are done, as well as all multicast groups created.

The CDP receives all of these messages and builds its database, composed of the *TOPOLOGY*, *CORRELATIONS* and *PATHS* tables, as described earlier in the document. Locally it deduces the initial reservation parameters for each CoS on each interface, once it knows all links capacity. This way,the signaling messages does not need to carry the initial reservations parameters and therefore avoid increasing signaling message size unnecessarily.

Notice that in the centralized scenario there is no need for the *VOPR* and *LISTS*, tables because there is only one CDP and it has all the information it needs. VOPR is only useful to avoid signaling between distributed control points, and the *LISTS* table useful to know which nodes should be advertised for changes in a given link or set of links.

At this point, initialization phase is over, and requests can be parsed. The local database is built and initialized. All requests will be received by the CDP where it will process and map requests into a certain Path that meets the QoS requirements. The process that regulates this decision is called Admission control. To help understand these mechanism a flow chart is presented in Figure 3.5.

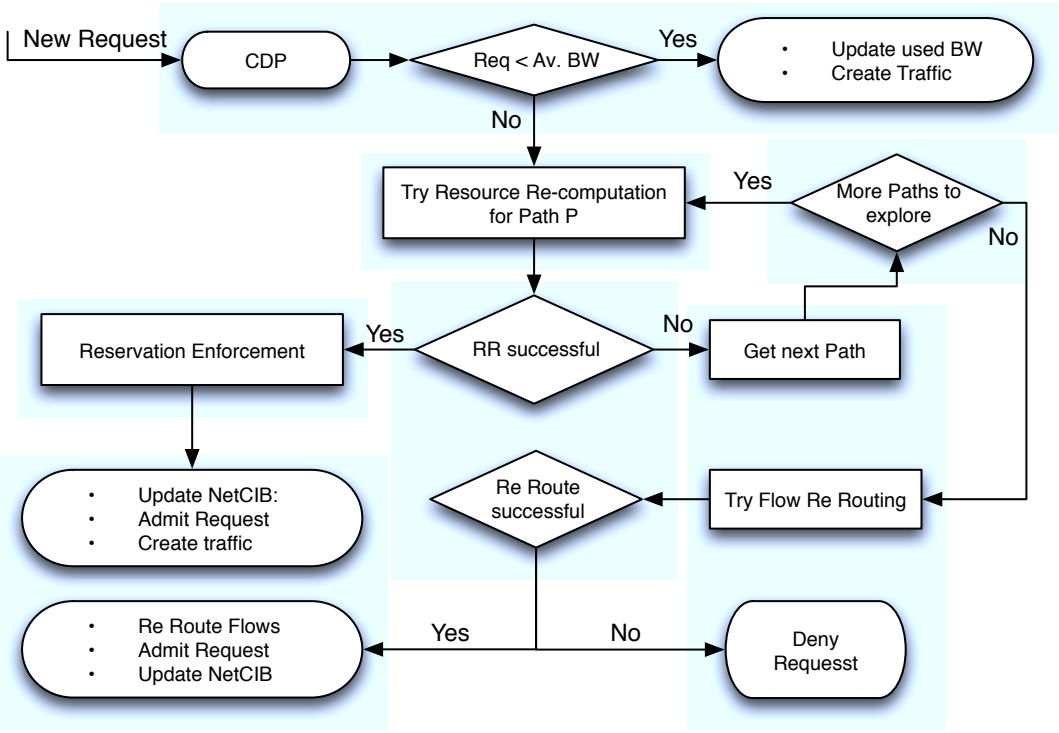


Figure 3.5: Centralized Scenario Normal Operation

The first step to parse the request is to make a selection of all the possible paths that connect the $Ig - Eg$ pair, the candidate Paths. Then, compare the request BW to the amount of available bandwidth in the Path with highest available BW. This amount is heavily influenced accordingly the the reservation algorithm used (and dependent on network usage, of course).

If the request BW is lower than the highest available bandwidth, the request is admitted into that Path. The CDP's database is updated, and packets start flowing trough the pre-configured multicast group, as we can see in the flow chart mentioned above.

In the centralized approach, requests are admitted based on available resources, because the CDP has all the information at any given point in time. If the request cannot be immediately admitted then, new values for the CoS reservation need to be computed. This is done accordingly to the reservation algorithm defined, and is explained in section 3.5. The computations are performed for all the candidate paths until they are successful in one. In this case the request can be mapped to it.

Before the packets can start flowing, we need to make sure that the calculations made by the CDP are enforced in the correct path. To do this, the CDP builds a message – Reservation Enforcement Message (RE) – containing these new computed parameters. The message gets the on-Path nodes addresses and travels trough the network using source routing.

At each node, the QoS requirements are updated, and the packet forwarded. Eventually, the

message reaches the end of the path in an *Egress* node. This node sends an acknowledgement back to the CDP which can admit the request into the network. Again, the CDP's database is updated, and packets start flowing trough the pre-configured multicast group.

In the case that no reservation re-computation is successful for any path, a mechanism called flow Re-routing is called. It aims at re-mapping existing flows from one path into another in order to free enough resources to accommodate the new request. This mechanism is explained further in section 3.4.3 once it is far from trivial.

If it is not possible to re-route any traffic, the request is denied. This mechanism repeats itself for every request and ensures that no QoS violation occur and that the network resources are capable of being fully used (no waste of resources).

3.7 Distributed Scenario

Self Organizing Multiple Edge Nodes (SOMEN) provides a mechanism to minimize the impact in network performance of decentralized network control in current and future Internet scenarios. It consists of enabling multiple network CDPs (e.g., network borders), operating in a decentralized manner, to cooperate, and quickly adapt to changes in the network and the related resource states, keeping low control, processing and signaling load.

We consider that requests (with a given BW) can come at any time to any CDP node. The CDP must decide on its own if the request can be accepted into the network or not, without degrading the QoS requirements of any active flow.

In the distributed scenario there are several CDPs that can accommodate requests. To synchronize the information on the several CDPs, information exchange is going to be mandatory. The challenge is how to exchange the least amount of information, the least number of times, but still allow for the best network performance possible.

The straight forward way is to make each CDP send a message to all other CDPs each time a new request is admitted. The so called *Per-Flow signaling*. This is far from meeting the requirements of low control, processing, and signaling load. In this sense, a dynamically controllable variable called Virtual Over PRovisioning (VOPR), is introduced.

The main idea of VOPR is to enable CDPs to dynamically define and exploit virtual resource for each Path based on links that lie on it. The VOPR of a CoS in a Path is represented by the VOPR on the bottleneck link of the Path. Every CDP can utilize the virtual resources defined for own Paths without issuing advertisement messages, and more importantly, without damaging system performance (while significantly reducing advertisement overhead). Moreover, functions to determine which CDPs should be advertised about specific information and the ones who don't, are taken into account assuring the least amount of messages are sent (Based on the *LISTS* Table).

At system initialization phase, each CDP finds all the possible paths between itself, and all others border routers (exit points – *Egresses*). This information is stored in a table, *Global Paths Table*, in each CDP's own NetCIB. This uses the same flooding mechanism describe in the previous section.

In large networks, the number of paths can grow very fast, which may affect computation time. To take this into account, a Path selection mechanism is implemented to copy some Selected Paths from the *Global Paths Table*, to a *PATHS Table* (the table being used for all computations). The mechanism that defines which Paths should be Selected can be based in several parameters: the number of hops, the available bandwidth, the cost, the correlation patterns, delay, etc. For simplicity reasons, the selection implemented is based on delay alone, being one Path selected if it's delay is below 500ms

(default value, can be modified). Hence, after selecting all its best paths according to the local policies defined by the operator, a CDP advertises all the other CDPs within the network about its selected paths and the related context information.

The information to be exchanged includes the Paths IDs, the list of the IDs of the outgoing interfaces that compose each Path and the capacity of each outgoing interface. Thus, upon receiving this information from others CDPs, every CDP is able to build the rest of it's own NetCIB.

The NetCIB database of a CDP is composed of several control information tables such as the *CORRELATIONS*, *TOPOLOGY*, *PATHS*, *VOPR*, *LISTS* tables. How these tables are build is explained section 3.4.

System initialization phase is similar to the centralized scenario, using the same mechanism for initial Path creation (flooding). The main difference is based on the type of database being built, and that the information gathered is exchanged between acsSOMEN-E agent instead of being sent to a central CDP.

After initialization phase all CDPs have their database built, and are able to accept requests. To easy the understanding of the normal operation, consider the flow chart presented in Figure 3.6. Any CDP_i can accept requests, i.e., any SOMEN-E agent, so these terms are interchangeable. In the flow chart SOMEN-E refers to the same node as CDP_i in the text.

When a request comes at CDP_i, the agent immediately checks if the request BW plus the used BW is lower than the highest available VOPR in any possible Path. If true, it is said that VOPR is available and the request is admitted into the network without signaling. The local NetCIB can then be updated by adding the amount of requested BW to the correct CoS in the mapped path in the *PATHS* table, after that the CDP_i returns to normal operation state. Therefore, the per-flow synchronization signaling is avoided without negatively affecting the accuracy of NetCIB and the system performance can be optimized, as long as VOPR is said to be available.

In the case there is not enough VOPR to accept the request (VOPR exhausted), CDP_i builds an advertisement message with the ID's of all possible Paths and advertises other correlated CDP's. These nodes to advertise are obtained from the *LISTS* Table. The message is called 1^{st} *ADVERTISEMENT* VOPR exhausted parameter 03. This advertisement takes into account the information in the *LISTS* Table, to ensure that only the CDPs that have relevant information receive the control message.

Upon receiving such advertisement, each other CDP retrieves their own links and own paths, which correlate with the advertised paths using the *CORRELATIONS* Table, and reply to the advertiser CDP_i with the IDs of own correlated paths and the used bandwidth in each CoS of each correlated path – the *Aggregated Used* BW information. The message is called *Response to* 1^{st} *ADVERTISEMENT* message, parameter 05. CDP_i waits until it receives all these messages and updates its NetCIB accordingly.

At this point CDP_i has on its local NetCIB the real description of the network. It can take all the decisions by himself, and send the results back to others in the end.

After the database is updated, new reservation parameters are computed for all advertised paths, accordingly the algorithm being run[4]. The request is admitted in the first Path that has enough resources accordingly to the new reservation parameters. In this case, the database is updated with the requested BW value, added an entry to the *SESSIONS* table, and the new parameters computed enforced in the respective path. The enforcement is done the same way as in the centralized scenario. New values for VOPR are computed for all Paths, so that the number of requests admitted without signaling has higher probability of occur.

[4]COR, A-COR or MARA

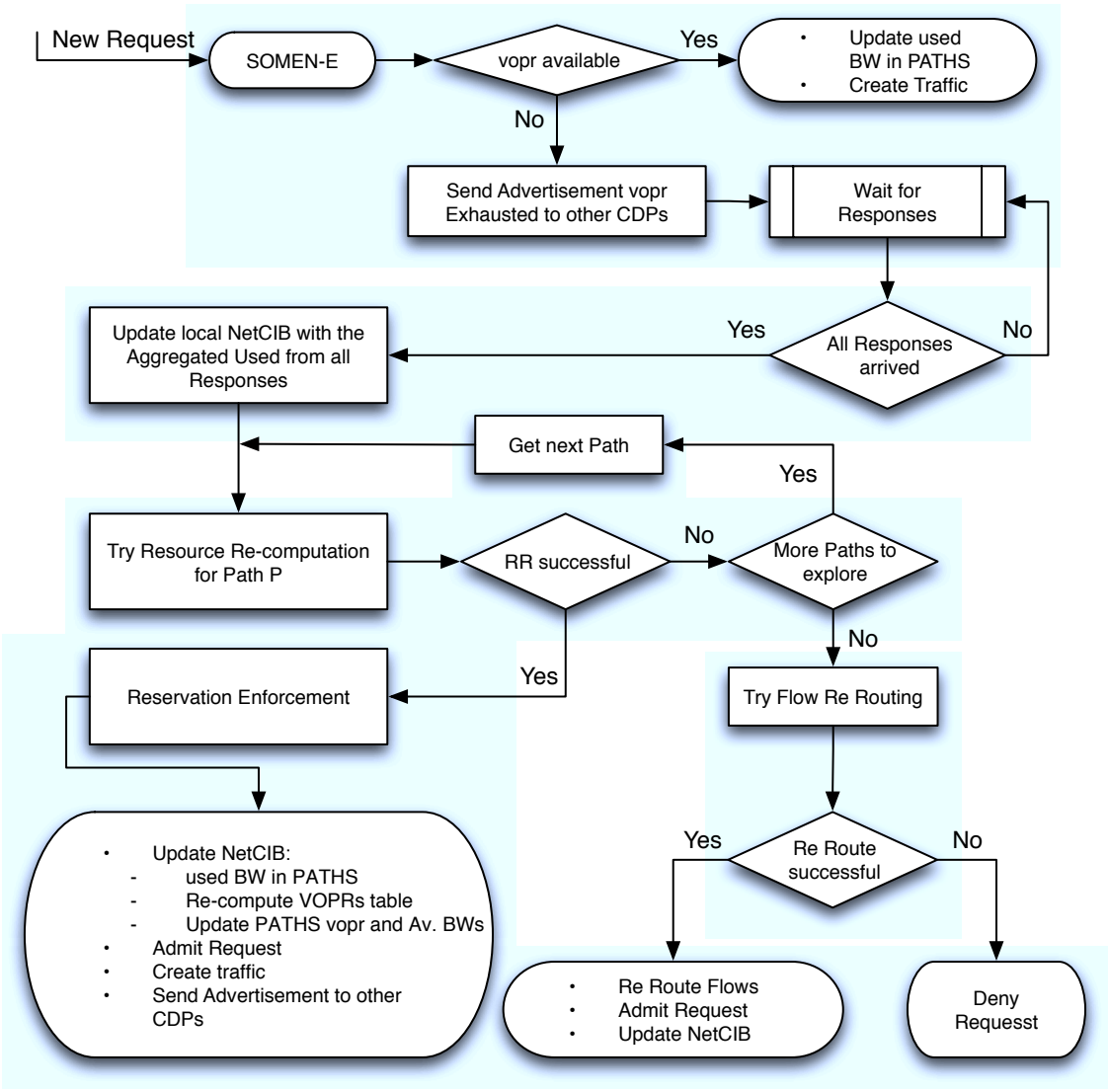


Figure 3.6: Global SOMEN Operation

Finally a well flagged message is built to carry these new reservation parameters and sent to the same list of CDPs nodes so that they can update their local NetCIBs as well. This message is called 2^{nd} *ADVERTISMENT* message, parameter 06. System returns to normal operation state.

If it is not possible to accept the request at any given path, all reservation computations fail, a mechanism called flow re-routing is called. This mechanism attempts to re-map existing flows from congested paths into other paths in such a way that makes it possible to free enough resources in some path so that the request can be admitted. The operation of this mechanism is complex and explained further, in section 3.4.3.

If flow Re-routing is successful the database is updated with the request used BW value, added an entry to the *SESSIONS* table, new values for VOPR are computed for all Paths.

Once flow Re-routing only considers active flows accepted through CDP_i there is no need to build a message to advertise other CDPs about all the changes in the network. Only an acknowledgement is sent so that other CDPs may return to normal operation state. This is due to the fact that new reservation parameters were not successfully computed for any path, so they stay with their old values. Values that are identical to he ones on all CDPs NetCIBs. The information that is different is the amount of used bandwidth for some paths, but that does not violate the respective VOPR values for any CDP. So, if some CDP VOPR exhausts, the whole process is triggered and the respective information updated.

Only when flow Re-routing fails the is request denied.

3.8 Conclusion

This chapter described the main features and functionalities of the proposals in order to facilitate the understanding of the implementations, as well as the results obtained. Therefore, the algorithms and mechanisms were explained from a theoretical point of view.

The centralized and distributed scenarios were presented, with a small discussion about the general operation of each one.

A special effort was made to explain the operations of the reservation algorithms used.

This dissertation continues with chapter 4 that will dwell in the implementation trade-offs and restrictions, as well as giving an insight into the simulation environment used.

Chapter 4

Implementation

4.1 Introduction

The implementation of the discussed proposals in chapter 3 is done using the well known Network Simulator (NS) version 2.31, that will be described in section 4.2.

All the code was written in C++[5], and in TCL [25] for the running scripts. This chapter will deeply present and explain the developed work, describing the mechanisms implemented to support the referred proposals. [8][5][25]

In section 4.2 we analyze the simulator used, and give a little insight about its design and capabilities.

In section 4.3 it's presented the work related to the centralized scenario.

Section 4.4 is the core of this work, the distributed architecture running SOMEN.

In section 4.8 are discussed the modifications done to the current multicast implementation in order to support multiple paths between the same A, B pair or nodes. It is fundamental for this work.

Finally a summary of the work can be found in Section 4.10.

4.2 NS2 Simulator

The Network Simulator (NS) (also commonly called NS-2, in reference to its version) is a discrete event network simulator. NS is popularly used in the simulation of routing and multicast protocols, among others, and is heavily used in ad-hoc networking research. It supports most of the popular network protocols, offering simulation results for wired and wireless networks alike.

Due to its open source model, it is specially popular in academia. However, in general, modeling is a very complex task, given the need to learn scripting, programming, modeling etc.

The basic design of the simulator consists in a split objects model. NS was built in C++ and provides a simulation interface through OTcl, an object-oriented dialect of Tcl. The user describes a network topology by writing OTcl scripts, and then the main NS program simulates that topology with specified parameters. Objects created in OTcl have a corresponding object in C++, as shown in Figure 4.1.

This automatically raises one question: *Why two languages?*

The answer lies in the kind of things it needs to do. On one hand, detailed simulation of protocols requires a system programming language which can efficiently manipulate bytes, packet headers, and

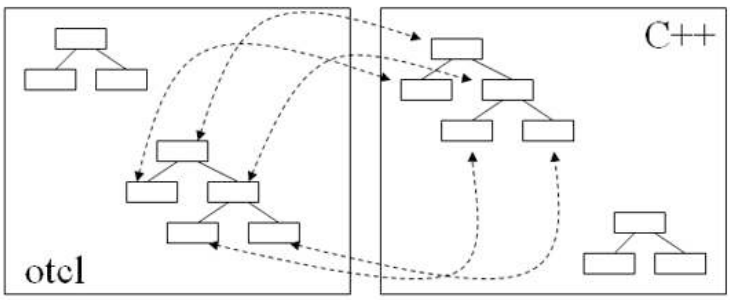


Figure 4.1: NS2 C++ Otcl structure

implement algorithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important.

This is handled with C++ code. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation.

On the other hand, a large part of network research involves varying parameters and configurations, or quickly exploring several different scenarios. In these cases, iteration time (change the model and re-run) is important. Since configuration runs once (at the beginning of the simulation), run-time is not very important.

This part is done with OTcl scripts. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration.

However, the user must master two completely different languages, that involve different principles, ideas, and syntaxes.

4.3 Centralized Scenario

The already existing implementation of the whole MARA platform was initially taken as a starting point. The first step was to run a few scripts and get used to the MARA platform. However, the platform was very low commented, very confusing, difficult to understand the meaning of the variables, and TCL configuration was very static making the program crash often due to small details.

The initial idea was to adapt the existing implementation by modifying only what is needed to implement SOMEN. After several months and several modifications and many lines of code written/rewritten, it was getting barely impossible to move forward. We decided to leave the existing implementation and build a new one from a blank page, using the existing one as a reference for specific details.

Basically not much was reused from the initial code apart from the initial paths discovery mechanism, based on flooding. Therefor, the SOMEN platform is mostly developed independently.

4.3.1 Network Initialization Phase

Initialization Phase has several purposes. It is responsible for paths discovery, initial DataBase construction, topology information gathering, among other functionalities, that will allow the system to work.

At the very beginning of the simulation, instances of all classes are created (nodes, links, queues, etc.) accordingly to the TCL script file. This is responsible for creating all Agents and nodes, variables initialization, links, queues, etc.

One C++ Agent Class called *LightAgent* must be attached to each node in the simulation. This class implements the basic functions of the *SOMEN-C* described in the chapter 3, like message forwarding, and information gathering, among others. Another class, *cdp_Agent* class is attached only at the CDP node. This implements all the functionalities of the CDP node.

After this simulation is running. At this point, we define the beginning of the system Initialization Phase.

In the TCL script, the command *"setupInterfaces"* must be called in the first place for all *LightAgents* in the network. After that, the command *"corInitialization"* must be called at all *LightAgents* that are also *Ingresses*. The later, will start the flooding mechanism for path discovery, and information gathering (every link's delay and bandwidth).

The flooding mechanism follows MARA implementation, although once the purposes of this work and MARAs are different, additional code had to be written and/or modified.

The objective is to find all possible paths between each *Ingress* and each *Egress* on the network. So, *"corInitialization"* builds and sends a message out all its own interfaces (or links). This message goes with the header *hdr_mrrp* struct fields *flag_i* and *msg_type* set to 1 (true/active) and to *RESERVE*, respectively. This message is received at each *LightAgent* function *recv()*. By looking at the previous header flags, through an if structure it calls *treatInitialization()*.

This function checks its own agent type: *Ingress*, *Core* or *Egress* node. In case this agent is an *Ingress* the received packet is discarded, meaning that a loop on the network has occurred. In case this agent is an *Core* node it then looks into the path vector on the *hdr_mrrp* field *reserve_path*. If it doesn't find its own address there, then it adds it, and forwards the message out to all its own interfaces, except the interface at which the message was received in the first place. Additionally, also fills the respective data fields for links bandwidth and delay with the correct values, obtained dynamically trough TCL interaction (not done in the initial MARA implementation).

If it finds its own address there, this means a loop has occurred and the message received is discarded, current function is terminated and this node returns to normal operation state. This avoids all possibilities that some packets would indefinitely run through the network in loops, devastating performance.

The last option, this agent is an *Egress* node. In this case a new path has been found. A response message is built, and sent to the *Ingress* node, by following the path in reverse direction. The response message carries a vector with all the nodes it has gone trough, as well as all the links bandwidths and delay values. This information is used to build the system DataBase.

Another functionality of this flooding mechanism, is that it initializes all CoS information for all links on the network. This has to be done for every link, for every CoS, accordingly to the Reservation Algorithm used. The Reservation Algorithm got from TCL is used by the *Core* nodes to correctly compute these values. To avoid reserving resources twice, or overwrite existing data, this computation is assured to be done only once. Initially all values are set to zero by the Agent class constructor. Upon receiving a flooding message, if the agent verifies that the CoS information is zeroed out, fulfills it, by computing the correct values according to the Reservation Algorithm. On the other hand, if those fields already have some value, it means that computation has been done and the agents skips this step.

The computations discussed in the previous paragraph are performed by the method

CoS_startUP_Reservation(interface) that takes one interface as input parameter. The computed values are enforced in the simulator environment upon successful computation. The command *addQueueWeights* is used. This command receives one CoS and a percentage value (*p*) as input parameters, then makes the given CoS have the weight *p*.

At some point, the flooding messages being propagated by the *Core* nodes get to an *Egress* node. As referred before, this means that a new path has been found and the Egress must build a *RESPONSE* message. This *RESPONSE* message follows the reverse path through each *Core* node until the *Ingress* node that generated it. As it goes trough each node, the function *treatResponseMessage()* creates and fills the MRIB routing tables and performs the required actions in the simulator environment to enforce the correct multicast tree creation. One multicast tree is created by each different path. The address of each multicast tree is defined as a variable value in the NS simulator as a string. Once there is no need to define and store a real IP address (similar to a node's IP being integer) the Path ID value is used as the multicast tree address. This value corresponds to the unique number that identifies each message in the simulator. It's used because in this way we uniquely map the address of a multicast tree to the respective Path ID, so when creating traffic sources the destination can be set to the correct multicast address, that will undoubtedly make packets follow the desired path. This way also simplifies a little the DataBase Paths table once there is no need to store the respective multicast address for each path (it is the path ID value).

The multicast tree creation is done in the reserve path to avoid sending another message in each path to build the MRIB tables. It cannot be done in the forwarding flooding because at that time the paths are not known and message loops may occur. By doing it in the return message we solve this problem.

The *RESPONSE* message is eventually received at the *Ingress* that automatically sends the information to the CDP node. A response message is built, and sent to the CDP node of which the address is know at first hand given by the TCL script file. The *RESPONSE* message carries a vector with all the nodes it has gone through (the Path), as well as all the links bandwidths and delay values.

Upon reception on the CDP, the *recv()* function calls *storeInitializationInformation()* which parses the information on the received packets and adds the new paths and links information to the respective database's tables. The flow chart 4.2 describes this process.

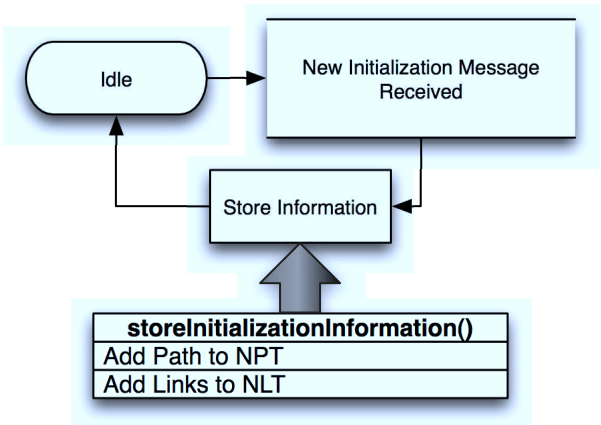


Figure 4.2: Store Initialization Information

4.3.2 Normal Operation State

The CDP normal operation state is defined as being the state after initialization phase where the CDP is idle, waiting for requests to come. It begins right after all the flooding messages are over, and all paths have been found. The CDP has no way to know when this happens, so the initial TCL script comes in to play another role. After calling *"corInitialization"* at all *Ingresses* the user must give a time out (usually 1 second is more than enough, but it depends on the size of the network and on the defined delay values for each link) and then call *"Build_Database"* at the CDP node in the referred TCL script. This will trigger three functions at the *cdpAgent*. One to build the Selected Paths Table[1], another to build the Correlated Paths Table[2] and a last one to fill CoS information into the Network Links Table[3].

This gets the database built. The follow flow chart of Figure 4.3 also represents this mechanism.

After this point requests can be parsed. One remark to be done is the detail in building the Selected Paths Table. This table gets all the Paths from the Network Paths Table, that has all possibilities. However, some of those possibilities may not be of interest in large networks (several hundreds of paths), because they may increase unnecessarily the computation time of the algorithm, among other avoidable problems. Some filtering needs to be done. This can be done in several ways, but it is not the focus of this work. We have done filtering based on total Path delay alone. By copying from the Network Paths Table to the Selected Paths Table only the Paths whose total delay is below a certain threshold (*500ms* by default).

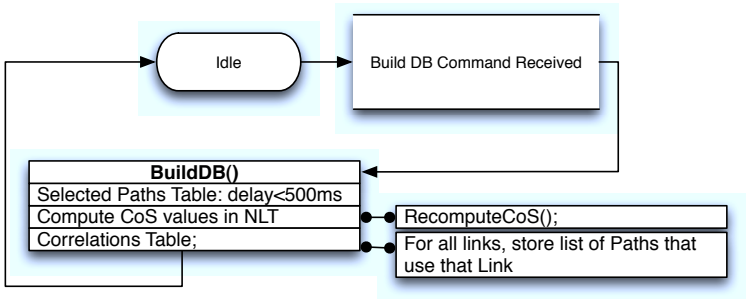


Figure 4.3: Build dataBase command

Now lets go over the interesting part of this work – parsing requests. All the above steps are done and the CDP is in normal operation state ready to accept a new request. From a code point of view, this is just another TCL script function being called with several parameters, as Table 4.1 describes.

Ingress	Egress	CoS	BW Kbps	Sid	Fid	Traffic	Duration
source	destination	class	bandwidth	Session ID	Flow ID	Exponential CBR Pareto	seconds

Table 4.1: TCL Request Parameters

[1] *Build_Selected_Paths_Table()*
[2] *Build_Correlated_Paths_Table()*
[3] *Recompute_CoS()*

The entry and exit points of the requested traffic (*Ingress* and *Egress* nodes) are given to the CDP, as well as the bandwidth requested and the CoS. A session ID and flow ID are passed and checked to be unique in the C++ code to prevent overlapping values or repetitions. The duration of the request in seconds is given. The CDP agent uses this value to automatically schedule in the NS the call of the method *"ReleasingRequest()"* (Section 4.9.1) so that this specific traffic ends after the given time. This method will stop the traffic in the simulation environment and update the database by removing the request bandwidth from the mapped path, and correlated paths. This variable can also take the value -1 meaning that the specific request will last until the end of the simulation.

The traffic type may be one of following: Exponential distribution, Constant Bit Rate (CBR), Pareto distribution. The traffic type parameter in used in the method described in Section 4.9.2 that generates the traffic in the simulation environment. This leads to a more realistic scenario of the network utilization by mixing these types of packet generation together. All three of them are natively supported by the simulator.

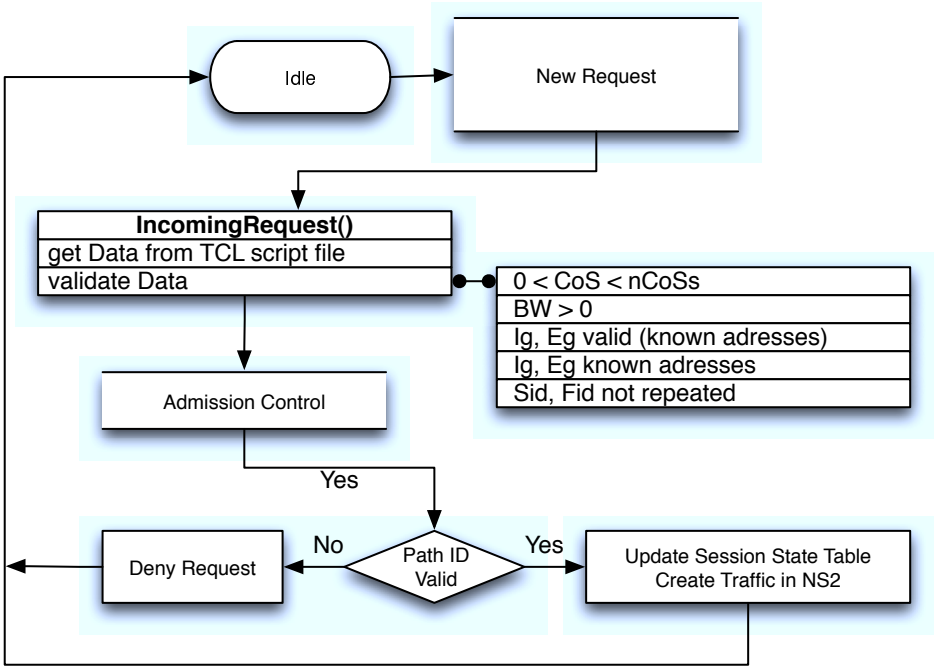


Figure 4.4: CDP Normal Operation (simple)

Once the TCL request command runs, *IncomingRequest()* is called by the *cdp_Agent* class, as shown in the Figure 4.4. This function will get all the data from the TCL and bind it to C++ local variables, then validate that data and call *AdmissionControl()* mechanism.

The validation of the data assures that a valid *Ingress–Egress* pair is given by checking the CDP database, that no negative values for Bandwidth are passed, that the CoS given is within the correct bounds and that the pair *Sid–Fid* is not already stored in the Session State Table. If any of the conditions above fail, the request is considered invalid and ignored, printing a warning to *stdout* (the screen, or the log file).

If the request is considered to be valid, Admission Control performs as will be described in 4.3.3 and returns a *Path* variable. If that variable has a negative value (*-1*) means that Admission Control

has found that the request cannot be accepted (not enough resources). The request is then denied, another warning is printed to *stdout* and the CDP returns to normal operation state.

In the best case scenario Admission Control returns a valid Path and the request is added to Session State Table. The database is updated and the traffic with the given parameters is generated in the simulator with the function described in 4.9.2.

CDP returns to normal operation state, ready to parse another request. The following flow chart describes the discussed mechanism.

4.3.3 Admission Control

Can the given request be admitted?

Admission control does only one thing: answer the question above. It's everything but an easy task. In the C++ files the admission control is implemented as a member of *cdp_Agent*, with the same name, *AdmissionControl( )*. It takes some input parameters of the pending request, like CoS and BW.

The steps performed can be followed by looking at the flow chart presented in Figure 4.5. First, the candidate paths between the requested *Ingress* and *Egress* are collected into a temporary table. This table is called *Temporary Selected Paths Table* – or for short *SPT* – and sorted by Available bandwidth in a decreasing order.

The temporary table is needed to compute the required operations without jeopardizing the original information in the database. The process may fail, and the original information should be preserved.

The Path with highest Available bandwidth (*index 0* after sorting) is tested to see if the request can be mapped into it. In the best case scenario, the request can be admitted (bandwidth below Available value) and admission control returns the current Path, to update local database and start traffic in the simulator. This situation is called *Admission Without Signaling*.

In some situations, the request cannot be admitted so easily. In the case the bandwidth requested is higher than the highest Available value, re-computation of the CoSs limits are done for all paths, one by one, until the first re-computed values are enough to accommodate the request. In that case, a message is built with the new computed information for Path_i and sent through all the links of Path_i using multicast.

The resource re-computation can be done in three distinct ways:

- Class-based bandwidth Over-Reservation (COR) – section 3.5.1
- Advanced Class-based bandwidth Over-Reservation (A-COR) – section 3.5.2
- Multi-user Aggregated Resource Allocation (MARA) – [19]

The resource reservation mechanism must be defined in the initial TCL script by setting the variable *ReservationAlgorithm* of both the *cdpAgent* and all *LightAgent*'s to 1, 2 or 3 respectively for COR, A-COR or MARA. Omitting this option from the TCL script will result in running the default value, COR algorithm, and a warning message being print to the Standard Output (STDOUT).

Still, the CoS re-computation may fail for all possible paths. This tends to happen more when network is near full utilization. In the event of such situation, the *Admission Control* mechanism calls *Flow Re-routing*, to search for the possibility of re-routing some flows from one Path to another in

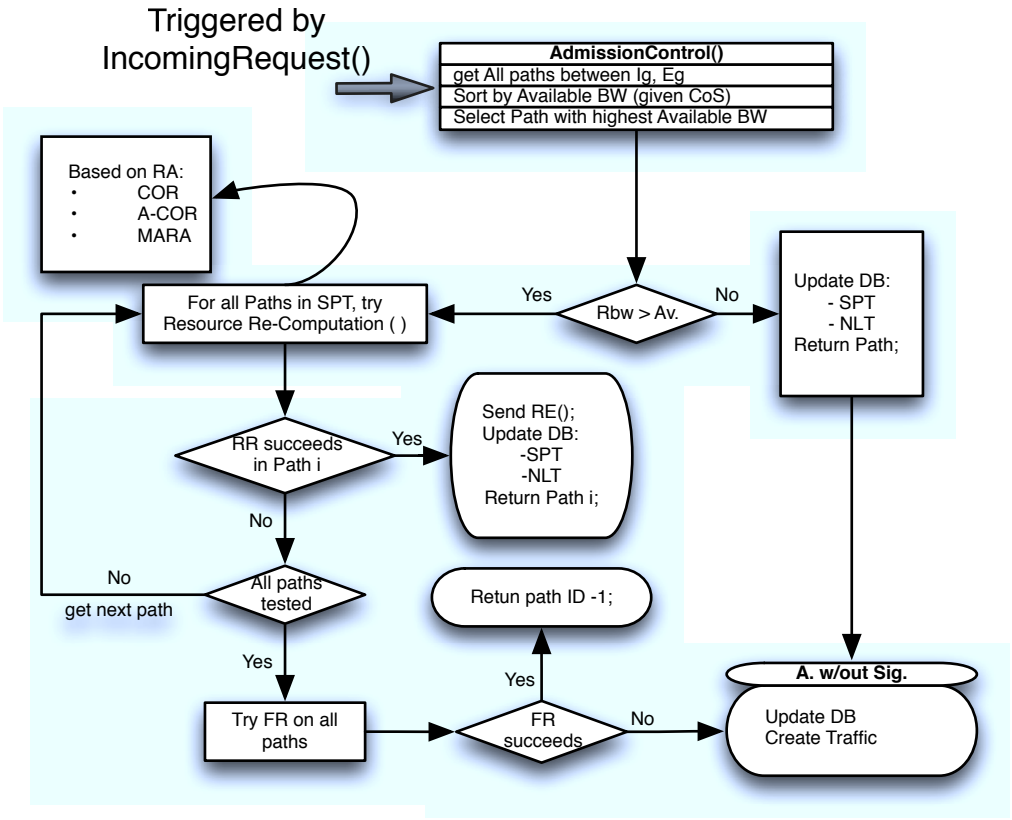


Figure 4.5: Admission control

order to get enough free space in some Path that will allow the system to accommodate the pending request. The complex behavior of it is described in the section 3.4.3 (general algorithm) and in Section 4.7 (the implementation).

From a black box point of view, the *Flow Re-routing* algorithm will return a valid Path if it succeeds. This path is returned by *Admission Control* and the request is admitted into it and the database updated. Similar to an *Admit without Signaling* event. Also *Flow Re-routing* guaranties that all the re-routing of the flows and the corresponding database updates are done before returning.

If *Flow Re-routing* returns an invalid path, it means it has failed and the request should be denied. CDP returns to normal operation state, ready to parse another request.

4.4 Distributed Scenario

4.4.1 Network Initialization Phase

The initialization process is almost the same as described in section 4.3.1. In this scenario there is no central CDP node, so the final information is not sent anywhere, it is stored in each *Ingress* database, the NetCIB.

The agents attached to each node in the TCL script file are different from the ones in the centralized scenario. We attach a *LightAgent* to all nodes and the *SOMENAgent* class to the *Ingresses*. The later, implements the functionalities of the SOMEN-E described in chapter 3.

As in the centralized scenario, the TCL script file must call the command *"setupInterfaces"* for all *LightAgents* and the command *"corInitialization"* for the nodes running the *SOMENAgent* (*Ingresses*). They perform the same functions as in the centralized scenario, the former gets information about Link Bandwidth, delay, among other variables, dynamically from the simulator, and the latter starts the flooding mechanism for path discovery.

The method *storeInitializationInformation()* receives the initialization packets sent by the *Egresses* and adds the information to the respective tables (new links to the Topology Table, new paths to the Global Paths Table). The mechanisms for path discovery, MRIB creation and information gathering are the same as in the centralized scenario.

The only remark to be made in this part is that, in order for all CDPs to be aware of all the Paths, the *Igresses* send their own Paths to all other *Ingresses*.

The *Ingresses* have no way to know if any information about a path will come in the future, so they cannot finish initialization phase and build the rest of the NetCIB on their own. The same situation was solved in the centralized scenario by issuing a command from the TCL script file.

In the distributed scenario, the command *"Build_NetCIB"* is invoked a timeout after both the commands *"setupInterfaces"* and *"corInitialization"* are called. Note that once each *Ingress* has it's own local database (NetCIB), *"Build_NetCIB"* must be issued for all *Ingress* nodes.

The command *"Build_NetCIB"* runs several methods from the *SOMENAgent* class, namely to build *PATHS*, *CORRELATIONS*, *LISTS* and *VOPR* tables. Finally, it computes and fills the *PATHS* table with current Available and VOPR bandwidth values.

After the NetCIB is built, Initialization Phase is over, and the systems enters normal operation state.

4.4.2 Normal Operation State

The normal operation state is defined as being the state after initialization phase where all CDPs are idle, waiting for requests to come. All *Ingresses* are considered CDPs because all run the *SOMEN-Agent*.

Requests come as TCL commands, basically with the same structure as in the centralized scenario (see Table 4.1). The difference is that the command can be issued to any of the SOMEN aware *Ingress* nodes, instead of being issued to the only centralized CDP.

The behavior that allows the request to be accepted or not, is very distinct from the centralized scenario. The flow chart of Figure 4.6 represents the actions taken upon receiving a request.

Following the flow chart referred, the general operation will be further explained. For the remaining of this section, let CDP_0 refer to the SOMEN Agent that received the request, and CDP_i all the SOMEN Agents that will be advertised.

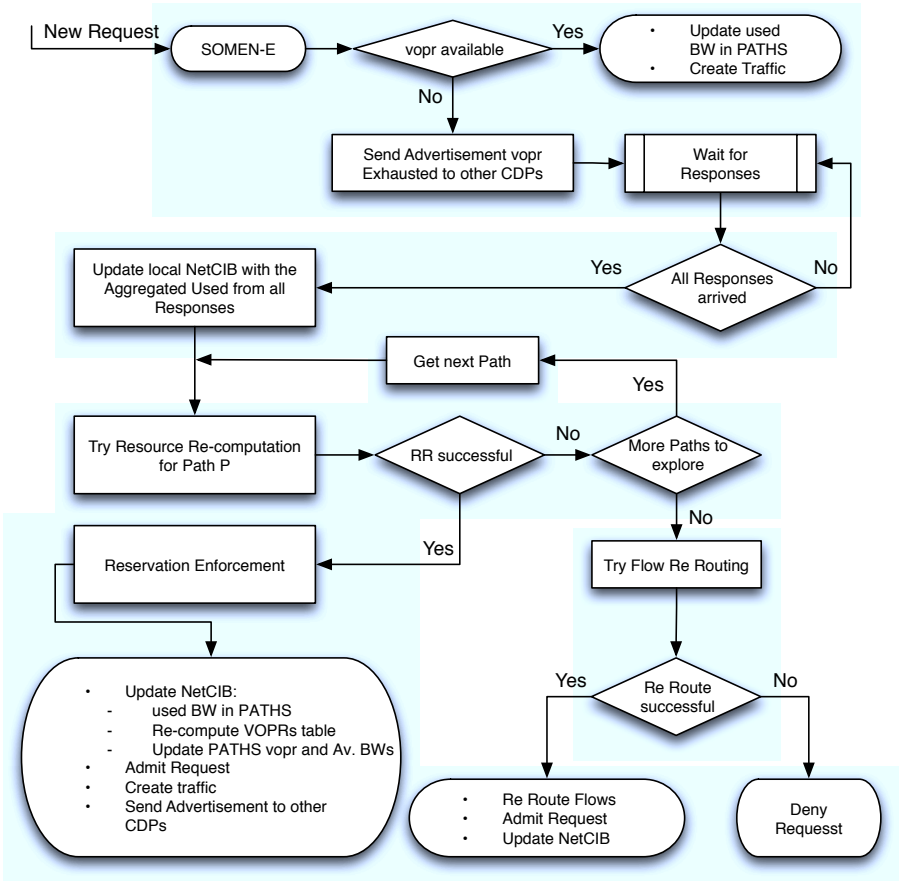


Figure 4.6: Global SOMEN Operation

The CDP_0 is idle and a new request comes. The method, *IncomingRequest()* is called. It parses the variables passed through the TCL command and validates them (non-negative CoS and BW values, existing *Egress* node, *Sid–Fid* not repeated). Then, the Admission control mechanism is called.

For all possible paths to the given *Egress* node, admission control compares the sum of the requested BW and the used BW in $Path_i$ to the respective VOPR value of $Path_i$ (for the requested

CoS), as follows:

$$request + used > vopr\ in\ Path_i$$

If the sum is above the VOPR value, admission control returns a negative number, meaning VOPR exhausted, otherwise returns $Path_i$.

This Path is selected to accommodate the request and the NetCIB is updated (request BW summed to used BW for $Path_i$). Traffic is generated in the simulation by calling the method *CreateTraffic()* 4.9.2, identical to the one in the centralized scenario. The system returns to idle state, and is able to parse another request. This is called *Admit Without Signaling*.

In the case Admission Control returns a negative value (-1), then other nodes must be advertised for the local NetCIB to be updated. This is called a *VOPR exhausted event*.

The CDP_0 searches it's local *LISTS* table and gets all the nodes it needs to advertise (all CDP_i), i.e., all the nodes that have at least one link that correlates with any of the possible paths. Then, for each node that must be advertised, a packet *VOPR_EXHAUSTED* is built that carries a vector with all Path ID's between CDP_0 and the *Egress* node in the request (destination). By using the *LISTS* table it's assured that only the nodes that need to be advertised are so, hence minimizing signaling load.

Upon receiving such a message, each CDP_i computes their Aggregated Used and builds a *RESPONSE* message, carrying the result of that computation. It sends this information to the CDP that originated the *VOPR_EXHAUSTED* message.

The aggregate used is the sum of all used BW from all previously admitted requests for all paths advertised, in each CoS. The involved CDP_i replies to CDP_0 with its used bandwidth in each CoS on each of its own paths which correlates with the advertised paths. This data will be used to correctly update the database of CDP_0 to the current network utilization state. We can see this in the *Wait for Responses* box in the flow chart of Figure 4.6.

In the C++ the method responsible for this computation is *ComputeAggregatedUsed(PathID_s)* that belong to the *SOMENAgent* class. It receives a vector of path ID's and returns a vector of Aggregated Used, built as being all the on-path links and respective used BW per CoS. This Aggregated Used is an instance of a class called *tuple_Link_Agg* that saves a link ID and a vector of used BWs, one per CoS.

The CDP_0 receives all of the above messages, as described in the flow chart. Then, it will update its NetCIB with the information received from all messages.

This mechanism is implemented in the method *VOPRexhausted()*. The temporary Aggregate Used from each CDP_i is saved in a static vector – *All_AggUsed_s*. This needs to be done because each time a node receives a packet, the NS triggers this node's *recv()* method. The SOMEN Agent *recv()* then calls *VOPRexhausted()* each time a packet comes. The static variable ensures that information is kept between calls of the method. Another static variable is used to save which nodes have already replied and which ones are pending. When *VOPRexhausted()* has received all responses it starts the NetCIB update using the information from the aggregate used saved in the static vector *All_AggUsed_s*. The method *UpdateTopolyTable(All_AggUsed_s)* does this update for the *TOPOLOGY* Table.

After the update, the CDP_0 database describes the current state of the full network utilization concerning the used bandwidth for all paths. Therefore it can make all the decisions alone, and send the result later to others CDP_i.

The next step is to try resource re-computation for all paths advertised. The algorithm can be COR, A-COR or MARA, as defined in the TCL script. For each path, the active algorithm runs, by calling the method *ReservationComputation(path,Rbw,CoS)* (Section 4.5). If any of the paths CoS

parameters can be successfully re-computed to accommodate the pending request, this mechanism is over, and we proceed to the database update and traffic generation in the simulator. No other path is tested for resource re-computation.

One important step to be done is the reservation enforcement i.e., create a message with the new computed CoS parameters and enforce them along the $Path_i$. This will re-configure the queues along all links on $Path_i$ in the simulation environment, done by method explained further in Section 4.6, *ReservationEnforcement(PATH,Links_Parameter_s)*.

The Update of the database takes into account the new computed QoS parameters (χ,B,etc) and is performed by method *UpdateNetCIB(LinkParameter_s)*. *LinkParameter_s* is a vector of the links that compose the path in which resource re-computation was successful, it is returned by method *ReservationComputation(path,CoS)*.

After this, system returns to idle state and can parse another request.

In the event that all Paths have been explored and resource re-computation is not possible, the algorithm tries *flow re-routing*. This mechanism is described in the flow chart of Figure 4.7 and in Section 4.7.

Flow re-routing will check if it is possible to re-route some flows from a given $Path_i$ to any another Path, in order to free enough resources to accommodate the pending request. It is a complex mechanism, but if it succeeds, the traffic is re-routed in the simulation and the method returns a Path variable, meaning that the request should be accepted in that Path.

If re-mapping is not possible, the method returns a negative value (-1) meaning failure, and the request is denied. If the simulation runs COR or A-COR this means that it is physically impossible to accept that request, i.e, request BW is higher than the available resources. There is no wasted bandwidth.

After this, system returns to idle state and can parse another request.

4.5 Bandwidth Over-Reservation Algorithms

The mechanisms of the reservation algorithms: COR, A-COR and MARA are all implemented in the same method. Whenever reservation re-computation is needed the following method is called:

ReservationComputation(path,Rbw,CoS)

There is an implementation of the method in the *SOMENAgent* class as well as in the *cdp_Agent* class. The method receives three parameters: one path ID, a value of BW and a CoS.

The method will try the active reservation algorithm (COR, A-COR or MARA) for the given path, in the given CoS, to see if the amount of BW given is possible to accommodate.

The active reservation algorithm is set by the TCL scrip by defining the TCL variable *"ReservationAlgorithm"* to a value of 1, 2 or 3 corresponding respectively to COR, A-COR or MARA. That TCL variable is bound to the C++ global variable *"ReservationAlgorithm"* present in both *SOMENAgent* and *cdp_Agent* classes.

By checking the value of it with an *IF* statement the method *ReservationComputation()* applies the correct calculations, as defined in Section 3.5. Only in this method does the reservation algorithm needs to be checked. The code was written in such a way that all the decisions and computations are independent of the reservation algorithm defined.

In case the reservation algorithm is set to three (MARA), another method by the name of *MrrpReservationComputation(path,request_bw,CoS)* is called. It was done this way once the MARA algorithm was re-used from an existing implementation, and all of it's code written inside this method. Changes were performed so that there was no variables in conflict and that the correct data was passed to the algorithm, retrieving it from the NetCIB. The method receives the same data passed to *ReservationComputation()* and returns the same data structure as well, a vector of links – *Links_Parameter_s*.

If the reservation algorithm is not MARA, then the computations are done as described in Section 3.5.1 for the COR algorithm, or as described in Section 3.5.2 for A-COR algorithm.

The details of the variables and data structures used are not needed to be fully explained, once one may read the implementation code and understand them with the help from comments and the respective Dissertation sections. Explaining them here would be repetitive once they follow very closely what has been defined in sections 3.5.1 and 3.5.2.

To be noticed however, is once A-COR does not need the threshold variable (χ), this variable is set to be -1 in the code. This way, if it's value is accessed while it shouldn't, it will lead to a negative value of resources, immediately identifying the problem.

The returned value of the described method (a vector of links as referred above) is composed by all the on-path links of the given path, with the QoS variables (χ,B,Av,etc) re-calculated and updated.

4.6 Reservation Enforcement

Both the *cdp_Agent* and the *SOMENAgent* have the same implementation of:

ReservationEnforcement(path,Links_Parameter_s)

This method is responsible to apply the QoS requirements computed along a certain path.

The parameters it receives are the path to which the changes should be applied, and a vector of on-path links with all the information per link attached: threshold (χ), reservation (B), available (Av), used (U) BWs. In other other words, the value returned by method *ReservationComputation(path,Rbw,CoS)*, described in Section 4.5.

Reservation Enforcement starts by parsing the information stored in the *Links_Parameters_s* vector into an appropriate format for the header of the *RESERVE* message. The appropriate format is a vector called *Reservation_Parameter_s* that is composed of instances of class *tuple_Rsv_Prm* (tuple Reservation Parameters). This class stores one link (node A, node B variables) and two vectors, one for threshold's (χ), another for reservation's (B) values. Both have size equal to the number of CoSs in the current simulation.

After the message is filled with these informations, we use multicast to propagate it along a path (once the multicast groups are created per path). The information of node *A*, node *B* is used for this purpose as well, so that no separate vector of nodes/links is needed. This message passes through all of the nodes running the *LightAgent* in the respective Path. Each one of them will read the information inside the header and update the simulation environment accordingly.

The method responsible is the *LightAgent::treatCoreReserve(packet)*. It starts by checking its own agent type which can be *Ingress*, *Core*, or *Egress* node. If agent type is *Ingress* or *Core*, information should be updated. If agent type is *Egress*, it has no information to update, but has to send a response back to the CDP saying that Update has been complete, send *Acknowledgment* message.

In case it is an *Ingress* or *Core* nodes the information is updated by calling the command:

$Q ***addQueueWeights*** *$CoS $percent*

This command will update the needed variables and information in the simulation environment by setting the CoS (*$CoS*) of the link *$Q* with the correct percentage (*$percent*) of the total link bandwidth. This percentage is computed has being the Reservation (B) divided by the total link capacity times one hundred ($percent = \frac{B}{T_{BW}} \times 100$).

4.7 Flow Re-Routing

The main objective of this algorithm is to try to re-route some flows from one path (*mainPath*) into other paths (*assinting paths*), in order to free enough resources in the *mainPath* so that the pending request can be admitted.

The SOMEN *flow re-routing* implementation (both centralized and distributed scenarios) explained in Section 3.4.3 is described here. In both scenarios the algorithm is the same, and implemented by method *FlowReRoute()*. Note only that in SOMEN decentralized networks, the available bandwidth of a CoS in a path should be seen as the VOPR of the CoS in the path. Remember that there is no re-routing taking flows from different CoS into account. Only flows that belong to the requested CoS are considered for re-routing.

The method receives some parameters including the tuple of *Ingress – Egress* nodes, the CoS and the request BW:

FlowReRoute(Ig–Eg, CoS, Rbw)

Some important variables to take into account are described in Table 4.2 for an easier reading. These variables will be addressed upon need by it's short name. The short name corresponds to the C++ code names used.

Short Name	**Full name**	**Description**
SPT	Selected Paths Table	Store the Paths that link *Ig – Eg*
SPT2	Selected Paths Table 2	Copy of SPT that stores temporary computation values
APT	Assisting Paths Table	A copy of SPT without the current *mainPath*
APT2		Structure is totally different from APT. This one stores a tuple of *pathID – list of flows*. It is used to compute the re-mapping of flows
MainList	MainList	List composed by the Flows that belong to the requested CoS in the *mainPath*. Changes with each iteration
mainPath	mainPath	current *mainPath* being processed

Table 4.2: Relevant variables from method *FlowReRoute()*

At first an *SPT* variable (Selected Paths Table) is built by copying from the main database the Paths that have *origin – destination* the *Ingress – Egress* addresses received. If this table as only one possible path, the algorithm terminates, as it is impossible to re-route anything because there are no other possible paths.

The table *SPT* is sorted by means of VOPR (or available BW in the centralized scenario) and indexed. This way, the first path has the highest available resources and is set to be the current *mainPath*. The *STP* table remains constant throughout all computations.

In order to make the C++ code jump back to the correct places accordingly to the flow chart in Figure 4.7 a somewhat complex loop structure was need. It consists of a *FOR* loop, and inside it a *Do While* loop. The *FOR* loop runs all paths of *SPT*. The *Do While* loop tries to re-map as many flows as needed, one per iteration, until there are enough free resources in *mainPath* or until all possibilities have been tested. After the above steps are taken, the *FOR* loop starts.

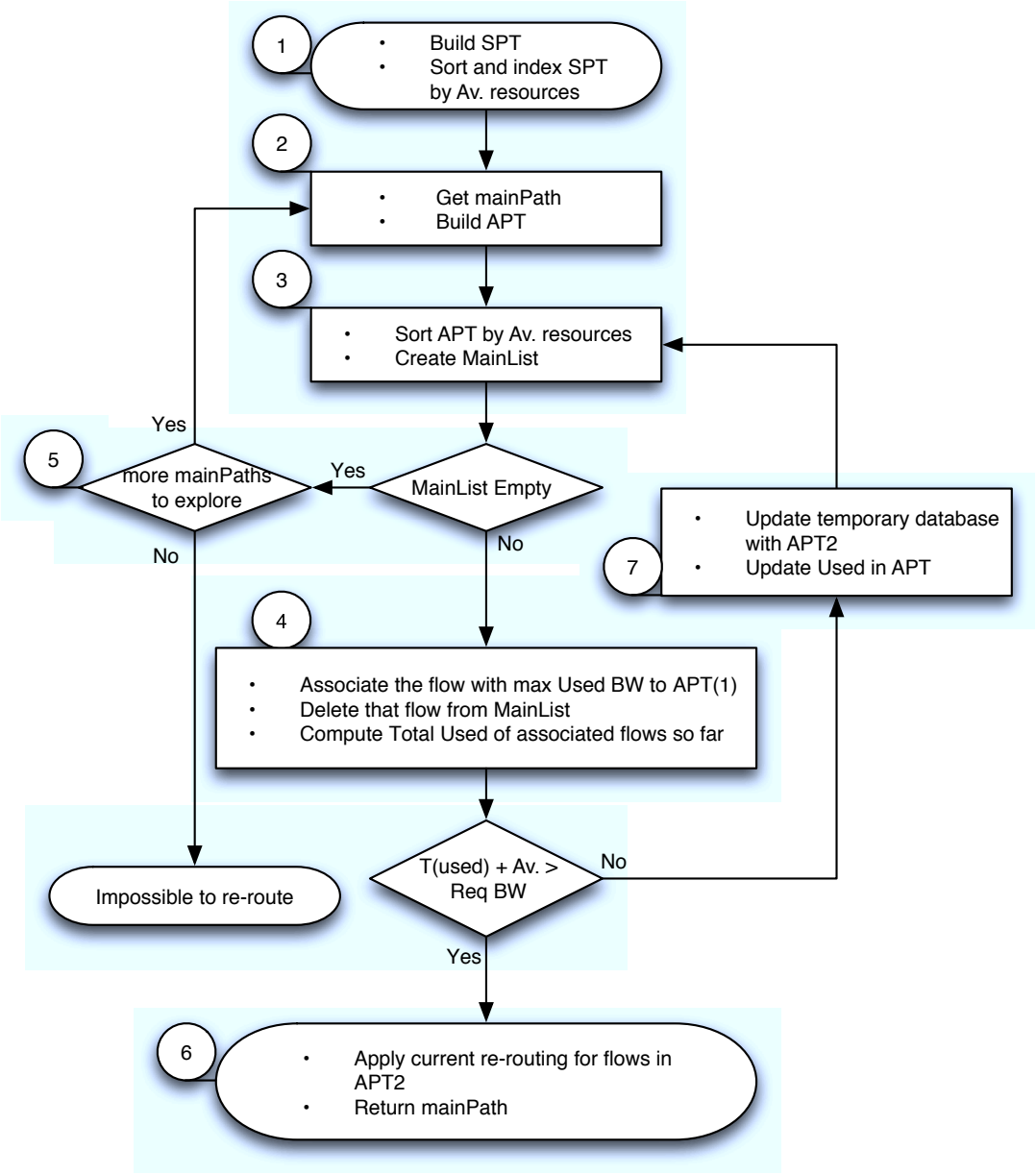


Figure 4.7: Flow Re-Routing

All paths except the current *mainPath* in *STP* are copied to another table called *APT – Assisting Paths Table*. These are the paths into which the algorithm will try to re-route the flows from the *mainPath*. APT is sorted by available resources.

The variable *MainList* is created, composed by all the active flows of the *mainPath*. Then, we delete the flows whose used BW is higher than the available resources in the index 1 of *APT – APT(1)* – because these flows can never be re-routed. Note that *APT(1)* is the assisting Path with highest available resources, if any flow cannot be mapped into this path it cannot be mapped into any other.

When removing flows from the *MainList* it may be empty, meaning that no candidate flows to be re-routed exist, so we check this with an *IF* statement. If the *MainList* is empty, we should get the next path in *SPT* to be the next *mainPath*, and resume computation. So *IF MainList* is empty, *continue*, which will break the current *FOR* iteration and start over in the next iteration. All variables are local to the *FOR* loop and therefore, are deleted. No need to worry about clearing the tables and so on.

The process repeats until there is at least one flow in *MainList* that can be re-routed. In this case the *Do While* loop starts. *SPT2* is created as being a copy of *SPT*. It is sorted by available resources. *APT2* table is created as well. These variables are local to the while loop so that there is no need to clear information between any two iterations, as done for the *FOR* loop.

To create the *APT2* table we copy the first assisting path (*APT(1)*) and to that path we associate the flow with highest used BW from the *MainList*. That flow is deleted from the MainList.

Then we assume that all the flows of *APT2* where actually re-routed. So we update the information of the *SPT2* table as if the re-routing as taken place, used BW changes for the paths in question and available resources are re-calculated. Remember that *SPT2* is temporary so if re-routing fails, the real database is still un-changed. Updated tables are sorted again for available resources.

Based on this we try to accommodate the request into the *mainPath*. This is done by computing the total used bandwidth of all the flows taken out of the *mainPath* plus the available resources that already existed, and compare it to the requested BW. For example, *mainPath* has 3 flows: *F1(100Kbps)*, *F2(200Kbps)* and *F3(300Kbps)*. Suppose that the incoming request is *450Kbps* and the remaining available BW in mainPath is *50Kbps* (meaning that the total capacity is *650Kbps*, in this example). So if we re-route *F1* and *F3* we free *100Kbps plus 300Kbps* – the total used is *400Kbps*. By comparing the total used plus the available resources (*450Kbps*) we see that there are now enough resources in the *mainPath* to accommodate the request. Temporary changes should be implemented and the new request admitted.

Now, if the incoming request would be of *500kbps*, for instance, this would not be enough and the while loop would run one more time and try to re-map *F2* as well. This would result in a total used of *600kbps* plus the already available makes *650Kbps* that is enough to accommodate the *500Kbps*.

This is what the flow chart decision box $T_{used} + Av. \geq Req_{BW}$ aims to do. In the case this condition comes to true the flows in the *APT2* are re-routed, and the real database is updated with the information of the *SPT2* Table. The *mainPath* is returned (it is guaranteed that there are enough resources to accommodate the pending request) and used by *Admission Control* to accept the request. The request is not accepted inside the *FlowReRouting()* method.

In the case that the condition proves to be false we need to get more flows out of the *mainPath* so the *Do While* goes to next iteration and the process repeats. Remember that if *Mainlist* becomes empty at any point a *continue* command is issued ending the while loop and running the next *FOR* iteration. Also, if all paths have been tested, the *FOR* loop will quit, meaning failure, and the method will return an invalid Path, with value -1. This will let Admission Control know that the request must be denied.

In order to update the real database, the mechanism is different for centralized and distributed scenarios. In the distributed scenario it's a little simpler once we just need to update the used BW for all paths in the *PATHS* table. This is done by copying the correct value of used BW from *APT2*, for all paths, all CoS.

Consider the Table 4.3 as being *APT2* table for the example presented above, in the case the request is *500Kbps*. Flows 1 and 3 where re-mapped to path 197 and flow 2 to path 207.

Assisting Path	**List of Flows**
Path 197	*F1(100Kbps)* • *F3(300Kbps)*
Path 207	*F2(200Kbps)*

Table 4.3: APT2 Table Example

So the update of the real database must add *400kbps* to used BW on path 197 and *200kbps* to used BW on path 207, all for the CoS in question (there is no re-routing between different CoSs). As well, *600kbps* must be decreased in the used BW of the *mainPath* once flows were taken from it and re-mapped. No other table needs to be updated once SOMEN uses VOPR and if it is exhausted, an advertisement event will be triggered, other nodes will be advertised, the database (several tables) will be updated and new VOPR values computed. In the distributed scenario the *TOPOLOGY* table is only updated after the next advertisement event is triggered.

In the centralized scenario it's more complicated, once the *TOPOLOGY* and *PATHS* tables must be updated. So, for all paths in *APT2*, for all on-path links, we sum the total used (*400kbps*) to the used BW value of each link. For the links in the *mainPath*, we decrease *600Kbps* in used BW.

Then, the easiest way to update the *PATHS* table is to re-use the methods that update all the correlated paths. Again, for all paths P_i in *APT2*, we get a list of paths that correlate with P_i (share at least one link). For all of those paths we re-compute the used BW values and respective available BW, taking the information of the recently updated *TOPOLOGY* table into account. This way the *PATHS* table gets it's values of used and available bandwidth (BW) updated.

The only loose end left to explain is how the re-routing is really done in the simulator environment. For this it may be useful reading Section 4.9.2 that explains how traffic is created.

Again, for all paths in *APT2* the records of each flow in the *SESSIONS* table are modified as some flows are now present in other paths. Then, for each flow that is updated in *SESSIONS* table, the command *"flow_name stop"* is called to stop the traffic in the simulator and the method *CreateTraffic()* is called with the required parameters taken from the stopped flow. This will effectively stop the on-going packets and start them up again in another path, hence re-mapping the flow.

4.8 Multicast Trees

The multicast trees referred in section 4.3.1 did not exist natively in the network simulator and were modified to work as needed. The multicast implementation used is also a non standard one, Overlay for Sourcespecific Multicast in Asymmetric Routing environments (OSMAR) ([22]).

One limitation of the OSMAR implementation is that only one MRIB table could exist in the simulation. In our scenario we needed a MRIB table per path. To better explain this limitation, let us go through the MRIB table.

Consider *NN* to be the number of nodes on a given simulation (number of *Ingress*, *Egress* and

Core nodes summed). So, accordingly to network Figure 4.8, $NN = 6$. One MRIB table is created as being a square matrix of *NN* lines by *NN* columns. In each position of the matrix is the next hop information.

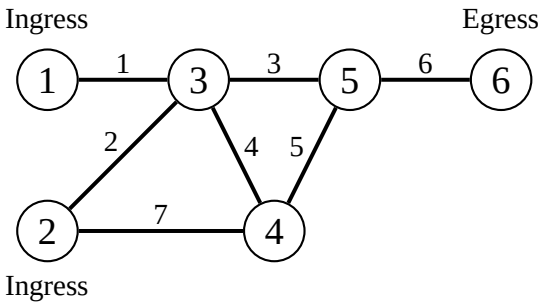


Figure 4.8: Network Example

For instance, if the packet is at node 1, and its destination is node 6, at position (1,6) there is, let's say, 3. That means the next hop between 1 and 6 is the node 3. At node 3, the next hop should be 5, for instance. So the field (3,6) is 5, and so on until the message get's to node 6. In the following tables, 4.4 corresponds to the path: 1,3,5,6 and table 4.5 to path 1,3,4,5,6.

	1	2	3	4	5	6
1	-	0	0	0	0	3
2	0	-	0	0	0	0
3	0	0	-	0	0	**5**
4	0	0	0	-	0	0
5	0	0	0	0	-	6
6	0	0	0	0	0	-

Table 4.4: MRIB for path 1–3–5–6

	1	2	3	4	5	6
1	-	0	0	0	0	3
2	0	-	0	0	0	0
3	0	0	-	0	0	**4**
4	0	0	0	-	0	5
5	0	0	0	0	-	6
6	0	0	0	0	0	-

Table 4.5: MRIB for path 1–3–5–4–5–6

As we can see, the position (3,6) has two different values for two different paths between the same source–destination pair of nodes. This limitation comes from the fact that MARA only takes into account one possible path for each pair, and this system was based on the original MARA implementation code. These path restrictions are no longer the case. Having only one table in the network is not enough.

The solution implemented was building one table per path. Although not being the most efficient way, its the way less changes into the code are needed, and once there is already one working platform –OSMAR – we are able to use it.

To make the simulation run with several tables, the functions to build them were modified to receive one input parameter, the table name. This name is the Path ID value, explained in section 4.3.1. To access one table one must specify its name as well. After making the changes for these methods, and to call them with the correct modified versions, the system works.

One more remark to be done is that these example tables are filled with zeros in all non-relevant positions to explain the limitations of the initial implementation. In a real simulation this zeros get the values from the standard routing information got from NS with command *"get-route-logic"*. The construction of these tables is done in the method *mrib_update()* of the *LightAgent* class.

4.9 Other Functions

4.9.1 Releasing Request

A request has begun, and there must be a way so it can end.

The running traffic must have a name, there cannot be two equal names in the simulation at the same time. The problem is how to give traffic variables a unique name and later be able to find that name to call the stop command at any given time. To solve this problem one should look into the format of the *Request* command and its input parameters 4.1.

We quickly notice that the value of the *Ingress* and the pair *Sid–Fid* uniquely identify one request, because no two requests coming from the same node can have the same *Sid–Fid* values. To uniquely name the traffic and be able to identify which name belongs to which request, the names of the traffic variables contain these three input parameters: *"trf_IngressSidFid‘"*; examples: *trf_014*, *trf_010* or *trf_317*.

This way we can call back the name of the traffic for any particular request that's running. So, the command *Release* takes the three input parameters discussed above: *Ingress* and pair *Sid–Fid*.

To stop the packet flow the function confirms within the *SESSIONS* Table the existence of the specified request. In the event the request is not found, the release command is terminated, effectively doing nothing but printing a warning message to *stdout*.

Upon success finding the request in the database the simulator is instructed to stop the ongoing traffic and the active flow is deleted from the *SESSIONS* Table. The database is updated by decreasing the used bandwidth in the TOPOLOGY and PATHS Tables accordingly.

4.9.2 Create Traffic

There are three types of traffic that can be generated. All are supported natively by the simulator, namely:

- CBR – *constant bit rate*
- EXP – *exponential distribution*
- PTO – *pareto distibution*

The CBR traffic creates packets at a constant rate that is given in Kilo bytes per second, between the specified *source* and *destination*. The EXP traffic creates a CBR traffic type during a set *ON time* and pauses during a set *OFF time*[4]. The Pareto traffic also takes into account the value set in the *shape* parameter to modify the packet flow rate.

There are more parameters that can be tuned like packet size for instance, but those are out of this scope and one can consult the NS documentation [9] to know more about the supported traffic types.

The *CreateTraffic* function receives all the information from the request command and executes the needed commands in the simulator environment to start the traffic.

The destination field of any traffic is set as the multicast address for the selected path. This way it is assured that all packets follow the desired path.

[4]There parameters are set to vary randomly between 10 and 50 ms

4.9.3 Compute Remaining BW

At the end of the simulation, it is important to collect the remaining BW for all paths. It is an important parameter to evaluate the performance of the algorithms. One that is able to use more of the total resources available is far better than one that uses less. For this, one could simple go trough all the lines of the PATHS table in each Ingress, get the Available BW for each path.

This is not correct. In SOMEN the CDPs will not have the database synchronized and as so, the Available BW values will not be correct. This lack of knowledge of the current network resources is not a problem during normal operation because we use VOPR. Due to this we must force a synchronization and update at least one NetCIB of a CDPs before trying to compute the remaining BW.

The solution implemented for this problem is as follows. At the call of *SaveStatistics()* (section 4.9.4) each CDP saves their own statistics as described, plus a temporary file containing the local used BW for each path, for each CoS in a tabular form, Table 4.6.

Address	Path ID	Used BW per CoS (Mbps)		
0	521	354.24	528.65	0.00
0	826	370.24	540.65	10.20
1	901	0.00	59.65	203.12
2	204	68.87	700.65	34.10
2	413	890.24	34.65	0.00

Table 4.6: Used BW Information in Temporary File

All CDPs append information to this temporary file. As the *SaveStatistics()* must be called for all ingresses, all paths used BW will be saved in that file. After this, a method will be called to parse the file and build a new database that has the latest information.

From the user point of view, all that needs to be done is calling the command *"Compute Remaining BW"* from the TCL script file, after calling *"Save Statistics"* for all ingresses. This command will load the file, and based on the *CORRELATIONS* table re-builds a Global Paths Table (all network paths) and with the later re-build a *TOPOLOGY* table with the correct information from the underling network. After the new *TOPOLOGY* Table has been built we can now compute the remaining BW,

accordingly to algorithm 4.9.1.

Algorithm 4.9.1: COMPUTE REMAINING BW($void$)

comment: For all Paths in *Global Paths* Table

for $j \leftarrow 1$ **to** *number of paths*
do
- get list of links from $Path_j$
- **comment:** For all Links in the list
- **for** $i \leftarrow 1$ **to** *number of links*
  do
  - get $link_i$
  - **for** $k \leftarrow 1$ **to** *number of CoSs*
    do { sum Used BW value for $link_i$, CoS_k
  - $freeBW \leftarrow linkCapacity - sumUsed$
  - **comment:** Store the minimum amount of freeBW in Path
  - **if** $freeBW \leq remaining$
    then $remaining \leftarrow freeBW$
- • Save the Remaining along with respective path ID into a file

4.9.4 Save Statistics

This function is called at last in the TCL script file to save all the CDP specific data of the simulation into log files. Parameters like the number of requests parsed, or how many times Flow Re-routing has run successfully, etc, are saved. This data is used to plot the results.

4.9.5 Auxiliary Functions

For non-critical functions like printing a vector of a certain data type, or save some string to a text file, a new file to accommodate all of this code was written – *AuxiliaryFunctions.h*. All of those functions are declared inside it and are very useful to hide some lines of code from the main implementation, leaving it less confusing and more readable.

4.10 Conclusion

This chapter described the main aspects of the implementation. The key methods and relevant flow charts were introduced for a better understanding of the work mechanism.

In Section 4.2 we introduced the simulation software used. Also, the simulation environment was detailed deeper and the requirements for running a simulation were presented.

Sections 4.3 and 4.4 described the implementation for the centralized and distributed approaches. The main key points and methods were explained. We introduced in particular the implementation of the reservation algorithms in Section 4.5 and the way reservation enforcement is done in the simulator, Section 4.6.

The implementation of the Flow Re-routing algorithm is also discussed in Section 4.7. In Section 4.8 the multicast implementation is detailed, once it was a difficult part of this work and not so

trivial to achieve. Nevertheless, it is of uttermost importance once it assures that packets flow in the desired path.

Some minor, but useful methods, were described in Section 4.9, *Other Functions*.

The next chapter will present and discuss the results obtained from simulation.

Chapter 5

Results

5.1 Introduction

This chapter exposes the performance evaluation of the different over-provisioning schemes analyzed in this work, in both the centralized and distributed approaches.

Section 5.2 explains the simulation setup and the parameters involved, focusing on the TCL configuration file.

Section 5.3 relates to the centralized network results and Section 5.4 to the distributed scenario running SOMEN.

The following sections provide the results and conclusions of different evaluations for several simulation parameters.

5.2 Simulation Scenario Details

For the simulation setup we used three different topologies, Topology 1, Topology 2 and Topology 3. These topologies are shown by figures: Figure 5.1, Figure 5.2 and Figure 5.3. Topology 2 and 3 have 1Gbps links and Topology 1 has 10Mbps links. Topology 1 is used to get the data related to QoS specifications, such as packets drops and mean delay. This topology has the same layout as Topology 3, but has 10Mbps links instead of 1Gbps links.

The reason for this is that when running the simulations, if link capacity is high packet numbers are high, and as the simulator treats packets individually, processing time and log file size become enormous. So, to make the file size and simulation times reasonable, we have decreased the links capacity. Nevertheless, for results related to the performance (e.g. number of events, load, etc.), we do not need traffic flowing. We can make all the computations assuming traffic is created, but then do not issue the command to actually create the packets. So, without data traffic on the network (just the protocol messages) simulation time and logs size is again small and can therefore be done.

The advantage is that we evaluate the work in a more realist network with 1Gbps links than in a network with 10Mbps links. When presenting the plots, each one will have the abbreviated topology run in the title, T1, T2 or T3.

All simulations run from two possible TCL script files, one file for the centralized scenario, another for the distributed. These files are responsible for configuring all the simulation parameters in NS, from the number of nodes and topology details, to the agents each node implements and the

different events throughout the simulation. The reason why two different files are needed is that several changes are done in order to correctly simulate each one of the scenarios. This is easier done in separate files than all in one.

One can also specify the reservation algorithm to use or the number of requests. If those values are set, simulation uses the values given, otherwise runs with standard defined values, being the reservation mechanism set to COR and the number of requests to a random number between 50 and 100.

Two types of seeds are used for the random generation. One of them affects mainly the network creation, i.e., BW requested, arrival time, etc. This seed can be named TCL *seed* once it influences the pseudo-random functions in the TCL environment. Another seed, the NS *seed*, affects the pseudo-random functions inside the simulator meaning that packet arrival times, delays (and so on) change if the seed is changed.

Although the TCL script written can create a pseudo-random network topology given the number of *Ingress*, *Egress* and *Core* nodes, we end up not using this functionality once the main objective is to compare behavior. For that we need a well structured topology, that remains the same through out several runs.

In this evaluation, we use two fixed topologies for the distributed scenario (T2 and T3). The two may seem similar, although they share the same number of nodes, they differ in the number of links. Topology 2 represents a network with low path correlation (few links) and topology 3 a high path correlation (more links). This will influence the way nodes advertise one another, hence the impact of load, particularly this will show the impact of the filtering mechanism implemented by the use of the LISTS table. We run the centralized scenario using topology 3 so that we can compare with the distributed architecture.

The TCL script starts by creating the number of nodes required and building a connection matrix. This matrix (m) has $N \times N$ size being N the total number of nodes. In each position $m(i, j)$ there is a 0 or a 1. If a 1 is present then node i connects to node j, if a zero is present there is no connection. The generation of the matrix can be pseudo-random, based on the TCL *seed*. If so, it also takes some restrains into account like all *Ingress* and *Egress* nodes must have at least one connection to a *Core* node, and a maximum of 3 different connections. The *Core* nodes must connect between them at least 2 times (avoid router on a stick situation) and up to a maximum of 4 links. *Ingresses* are not allowed to directly connect to *Egresses*, neither a node is allowed to connect to itself.

As said before, we are not using this random matrix but instead running fixed topologies, so by changing each position of the matrix $m(i, j)$ to the desired value (1 or 0), we enforce the creation of the desired network.

Based on this matrix, the links are created dynamically. The capacity of each link is set to be the value in the variable *LinkBW* so that all links have equal capacity. This value is *1Gbps* by default, but can be modified, for instance, this becomes *10Mbps* for the topology 1. All the Links delay need to be set in the simulator, and are random generated between 1 and 5 milliseconds.

The number of CoS can be defined (by default is 3) as well as the reserved bandwidth for the signaling class, set by default to *100Kbps*. The *SOMEN* or *cdp_Agent* classes are attached to the correct nodes, and parameters needed by those agents are configured, based on the file being run (centralized or distributed version). The scheduling models for the queues and polices are defined dynamically accordingly to the connections matrix. It was considered *Weighted Fair Queuing* discipline.

The commands for the simulation must be set before simulation start. For instance, the commands for database creation, request processing, and for information printing and saving are set in the end of

the TCL file.

Requests have all their parameters random generated like the source and destination addresses, the individual BW, the CoS, duration time, etc. All the parameters can be found in Table 5.1.

Finally the simulation is set to run.

In order to show more accurate results, every simulation is run 10 times with different TCL and NS seeds each time. Then, the mean values are plotted with the corresponding 90% confidence intervals.

The simulations were run from a bash script file, that calls the TCL configuration file and the NS simulator automatically to obtain all the results. We set the number of seeds, the topology, the number of requests, so on, and the bash runs all the simulations and saves the output files organizing them into folders. Then, we use Matlab[15] to parse those files inside the folders and obtain the plots shown in this chapter.

Time	Arrival time at the Ingress node
Source	Valid Ingress Address
Destination	Valid Egress Address
CoS	Required Class of Service
BW	Required bandwidth
Sid, Fid	Pair Session ID, Flow ID
Traffic Type	CBR, Exponential or Pareto distribution
Duration	Duration time in seconds

Table 5.1: TCL Request Parameters

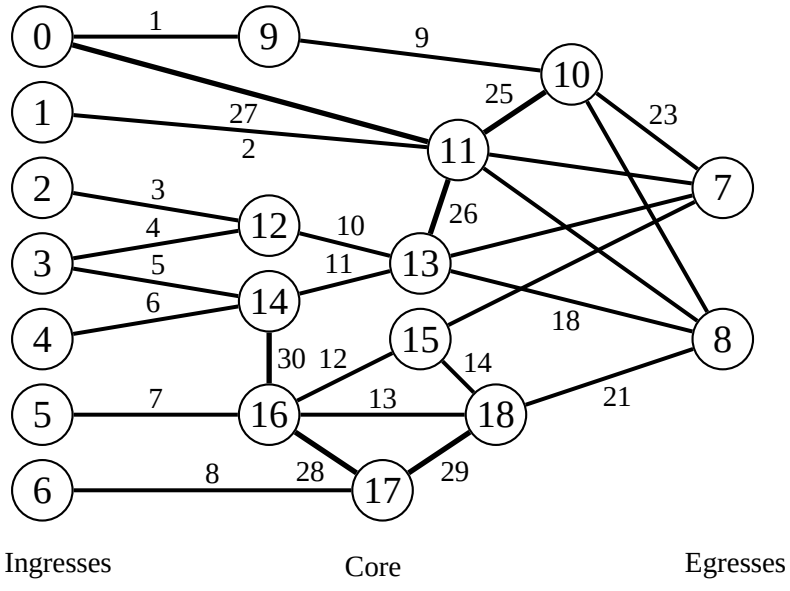


Figure 5.1: Topology 1 – 10Mbps Links

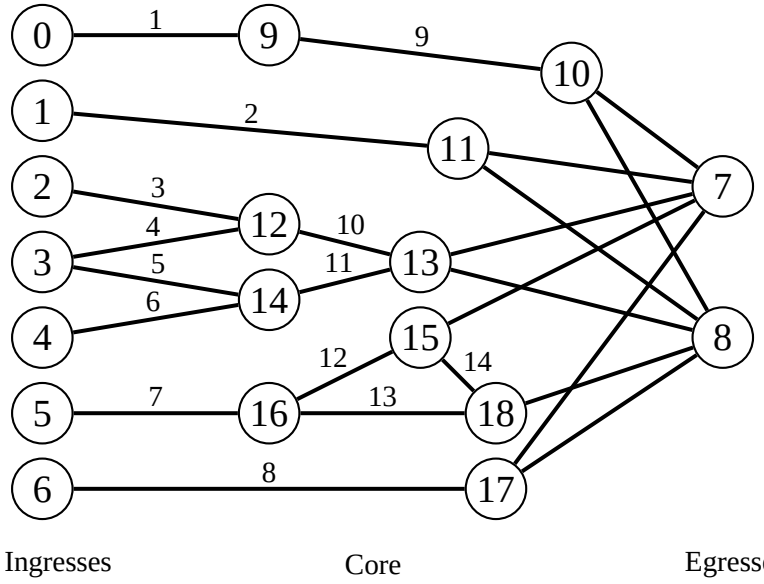


Figure 5.2: Topology 2 – Low Correlation (1Gbps links)

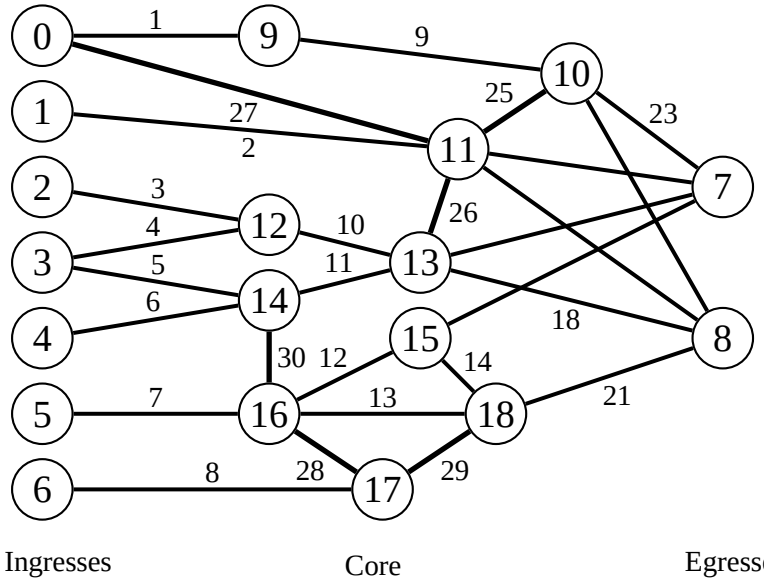


Figure 5.3: Topology 3 – High Correlation (1Gbps links)

5.3 Centralized Scenario

The simulations below were done with the parameters on Table 5.2 fixed.

We aim to analyze the different reservation algorithms by varying the network utilization ratio. We are interested in the number of re-computation events, total message load, and resource utilization efficiency for each algorithm.

The requests are set to an infinite time, meaning that once traffic is admitted it will stay active until the end of the simulation. This makes it easier to compute the approximate value of network utilization ratio. By changing the total number of requests, were in fact analyzing the behavior of the reservation algorithms from a low network utilization to high network utilization ratios.

Topology	Links	Data CoS	Mean Request BW	Number of Requests
3	*1Gbps*	3	*5Mbps* (500K to 10M range)	from 600 to 2600 (400 step)

Table 5.2: Constant Simulation Parameters

We start by looking at the number of requests denied while resources were available, Figure 5.4. We say that a request is denied while resources were available if at the time a request is denied, the total amount of free BW in a possible path is more than the request amount. These events lead to waste of resources and bad network utilization efficiency and therefore should be avoided.

The total number of denied requests is not much relevant because when it is impossible to admit traffic to an already congested path, a high number of denied requests would not translate to a good or bad efficiency of the reservation algorithm. It would simple mean that there are no more resources in the physical network.

COR and A-COR have a zero probability of denying a request if the network has enough resources. As shown in Figure 5.4, the MARA algorithm denies several requests while there is enough BW in the network and therefore wastes resources.

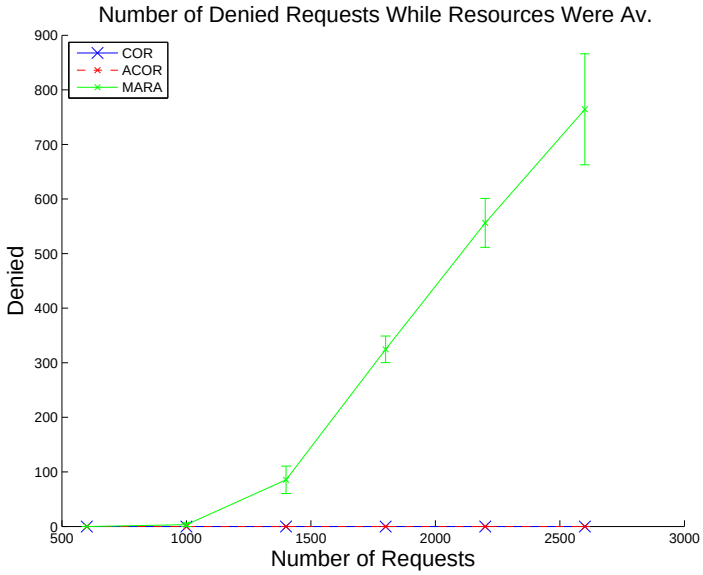


Figure 5.4: Cen. - Requests Denied While Resources Were Available

Figure 5.5 represents the number of times (events) that reservation re-computation took place. On this plot is important to notice the behavior of A-COR, that distinguishes itself from others, by having very few re-computation events at low network utilization. This means a more balanced way of distributing resources among existing CoSs.

An algorithm that has more re-computation events, needs more messages in the network to enforce the changes. So for each event triggered, a message flows along the given path to enforce the new QoS parameters.

Figure 5.6 exhibits the respective load for the total number of the events. A-COR has less load than others, being COR the algorithm that has more overhead, mainly because it triggers more events as well. The different between COR and A-COR is simply because A-COR has less re-computation events and therefore sends less messages.

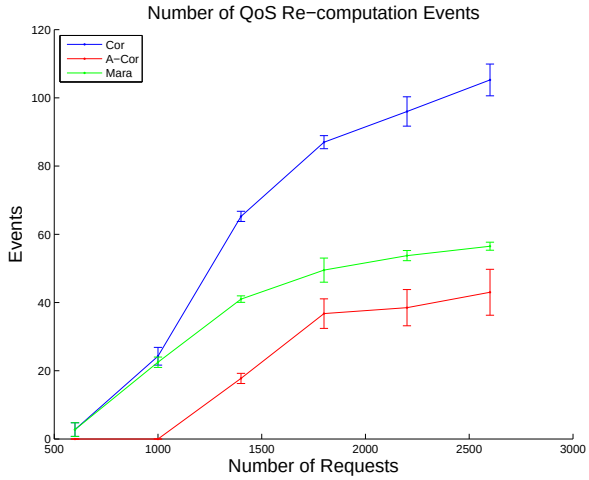


Figure 5.5: Cen. - Reservation Re-computation Events

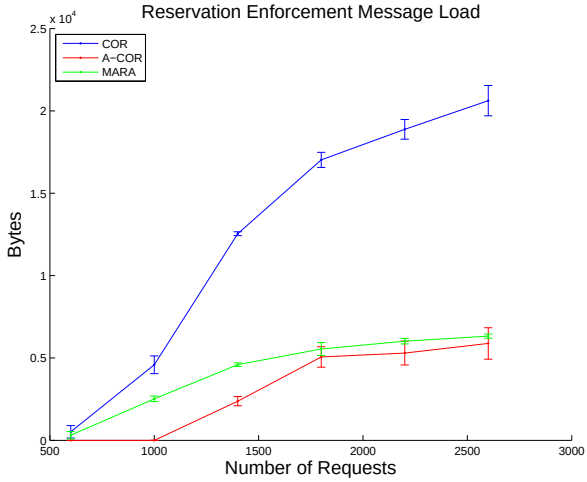


Figure 5.6: Cen. - Total Load of Reservation Enforcement Messages

As we have seen before in figure 5.4 MARA has a non-zero blocking probability even with low network utilization ratios. It means that some requests are denied even when resources are available. For the other algorithms this cannot happen. As a consequence of this fact, it is useful to plot the BW wasted. Figure 5.8 exhibits the mean BW wasted, computed as being the available resources when a request is denied when it should have been admitted. For all the denied requests in this situation, the mean waste and respective confidence interval (at 90%) are shown below.

COR and A-COR do not show waste of bandwidth (always zero). MARA wastes some resources, that decrease as network utilization ratio increases. Waste of resources is a major concern and an algorithm that can fully use the network infra-structure have a big advantage once it allows more traffic into the network (more revenue for network operators).

Another very important variable to take into account is the amount of free resources at the end of the simulation. This is computed as being the remaining BW in each path for all seeds simulated, and then the mean and confidence intervals are computed and depicted in Figure 5.7.

Opposed to MARA, A-COR and COR make a better use of the network infrastructure as they allow allocating all the free resources.

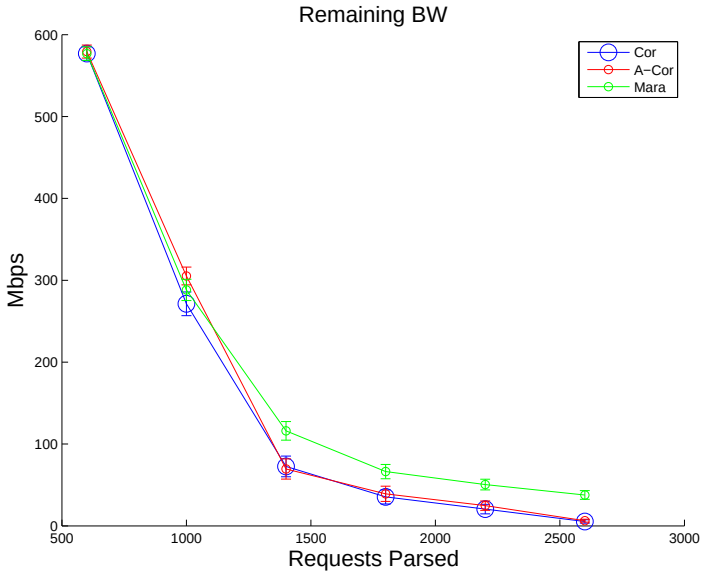


Figure 5.7: Cen. - Remaining BW

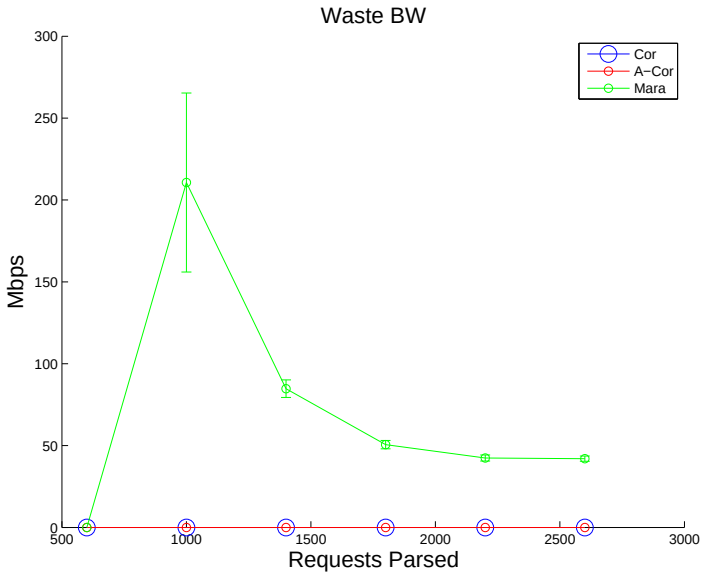


Figure 5.8: Cen. - Waste BW

The last plots to be analyzed are the ones for the QoS requirements: delay and packet drops. For these simulations, the network parameters are different, and defined in Table 5.3. The links capacity has been reduced from 1Gbps to 10Mbps in order to put traffic into the network. Recall that if we used 1Gbps links and put data packets flowing, the simulation time would be to high and the log files size would be huge, hence the reduction. The total number of requests was reduced to be from 50 to 100 requests and the mean BW per request was also decreased.

Topology	Links	Data CoS	Mean Request BW	Number of Requests
1	*10Mbps*	3	*1Mbps* (100K to 2.1M range)	from 50 to 100 (10 step)

Table 5.3: Constant Parameters – traffic simulations

The values plotted are obtained for 10 simulations varying only the NS seed. In the simulations before, we varied the TCL and NS seeds once we wanted requests and configurations to change. Now, we are interested in measuring delay for exactly the same set of requests but with different reservation algorithms, hence we keep the TCL seed constant and vary the NS seed, so that the events in the simulator will still be random (packets arrival and departure times in queues, etc.).

The delay calculations only take into account the data packets, once these are the ones of interest to ensure QoS requirements. The time a packet spends in the network is computed as the time a given data packet reaches an *Egress* node (T_{end}) minus the time that the same packet was sent by the *Ingress* node (T_{start}). Each packet has a unique number in the simulator log file (the trace file) that is used to correctly identify the packet in question to get the times mentioned. So, the overall network delay is computed as equation 5.1 describes.

$$D = \frac{\sum_{i=1}^{N}(T_{end}(i) - T_{start}(i))}{N} \tag{5.1}$$

Where D is the delay result attained, N is the total number of packets accounted for, and T_{end} and T_{start} the times explained above.

To compute the mean delay per CoS we used the same procedure but only considered packets for the specific CoS. The simulator log file (the trace file) has the CoSs each packet was mapped to. To compute the total number of drops, we just run the trace file for each simulation and count the number of drops.

Plot 5.9 shows the number of packet drops for all reservation algorithms. As we have said before, SOMEN ensures that no packets are dropped, so if there are no resources no traffic is admitted, hence no packet is dropped, independently of the reservation algorithm.

The weights of each CoS are equal (the alpha values referred in chapter 3, Section 3.5), so all CoSs should have the same treatment, hence similar delays. Plots Figure 5.10 and Figure 5.11, that represent COR and A-COR respectively, show similar delays between the different CoSs. However, there is a slight longer delay for CoS 3. This may be related to the fact that the requests are random and so is the class and the amount of BW. Therefore, it is possible that for class 3 more request exist, hence more packets, and longer delay (queues more full). Recall that the TCL seed in kept constant so that all the algorithms (COR, A-COR and MARA) have the same set of requests, fact which plots in Figure 5.10 and in Figure 5.11 agree since both have higher delay for class 3.

This would also mean that MARA (Figure 5.12) should have higher delay for class 3, although the behavior of MARA is very different from the behavior of the other algorithms and that heavily influences the paths into which each request is mapped. Therefore, delay is going to be different.

Another aspect to take in consideration is that MARA denies more requests and so has less traffic in the network. Consider plots of Figure 5.14 and Figure 5.15 that represent the remaining and the waste bandwidths. We see that MARA accepts less requests and this means that less packets are flowing.

Considering the overall network delay, found in Figure 5.13, we see that it increases as the network utilization increases, as expected. The MARA values are lower because fewer requests are accepted (final network utilization is less than 70%) and so, less packets are on the network which in turn contribute for lower delays.

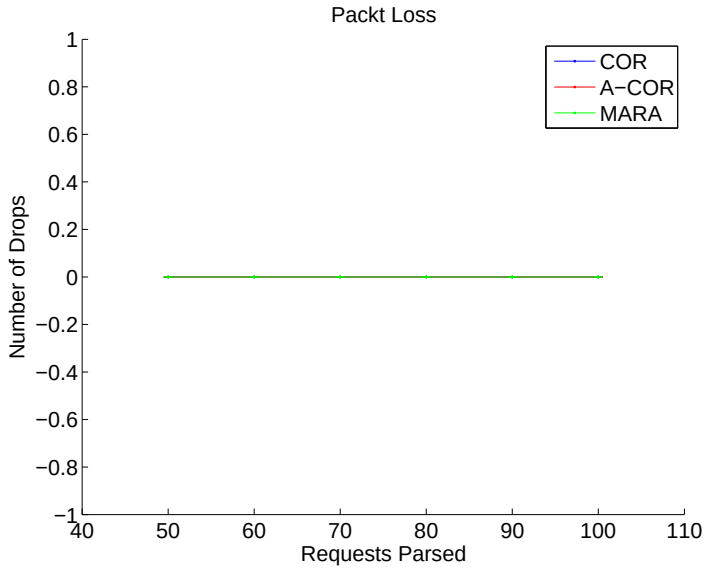


Figure 5.9: T1 - Cen. Mean Packet Drops

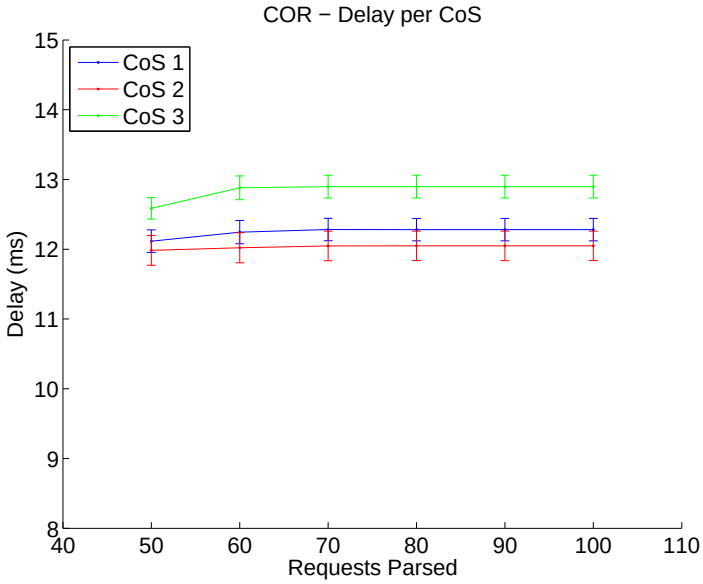


Figure 5.10: T1 - Cen. COR – Mean Delay per CoS

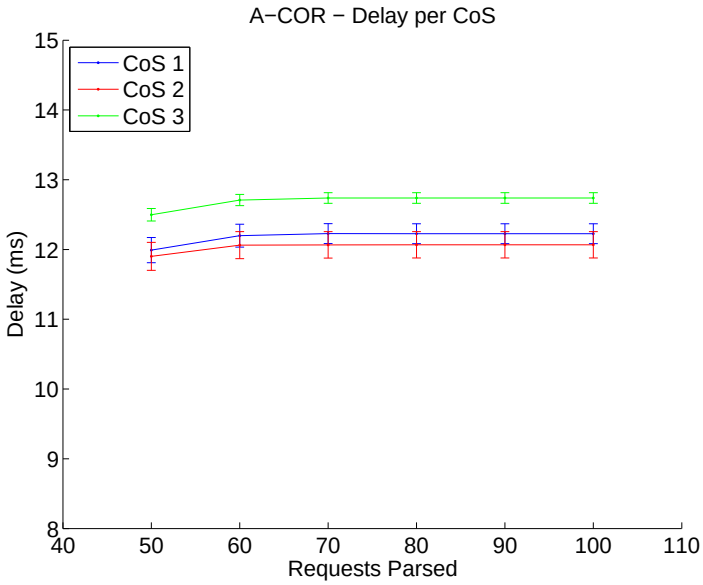


Figure 5.11: T1 - Cen. A-COR – Mean Delay per CoS

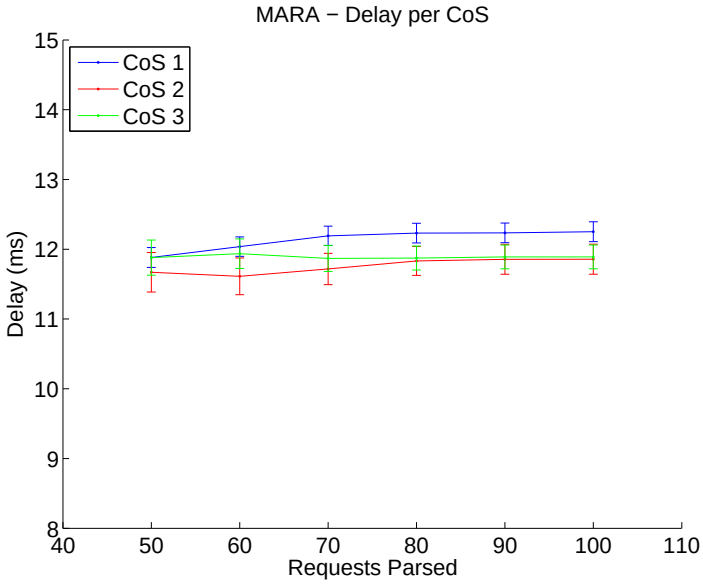


Figure 5.12: T1 - Cen. MARA – Mean Delay per CoS

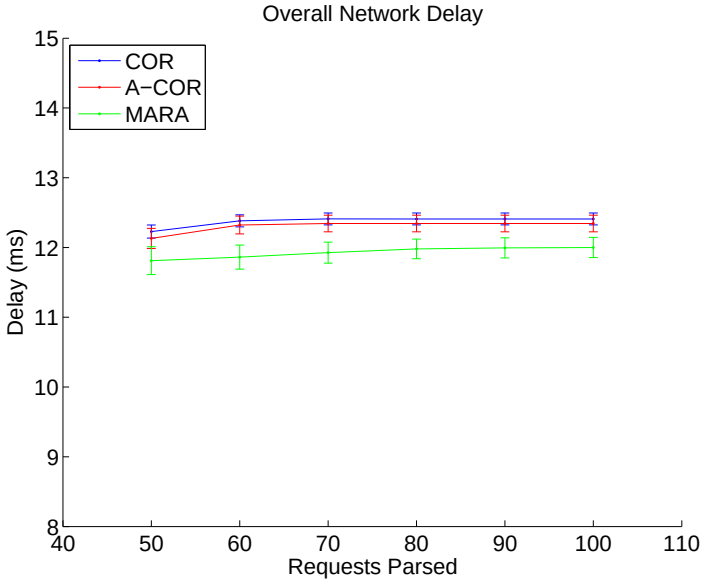


Figure 5.13: T1 - Cen. Mean Overall Network Delay

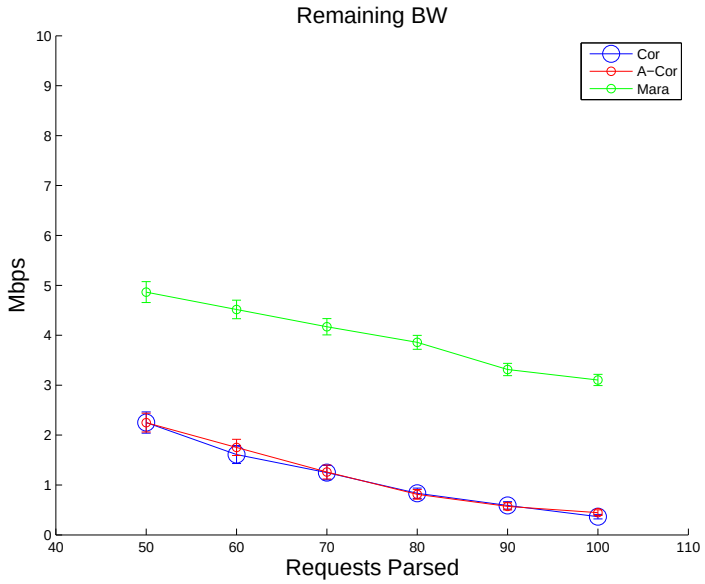


Figure 5.14: T1 - Cen. Mean Remaining Bandwidth

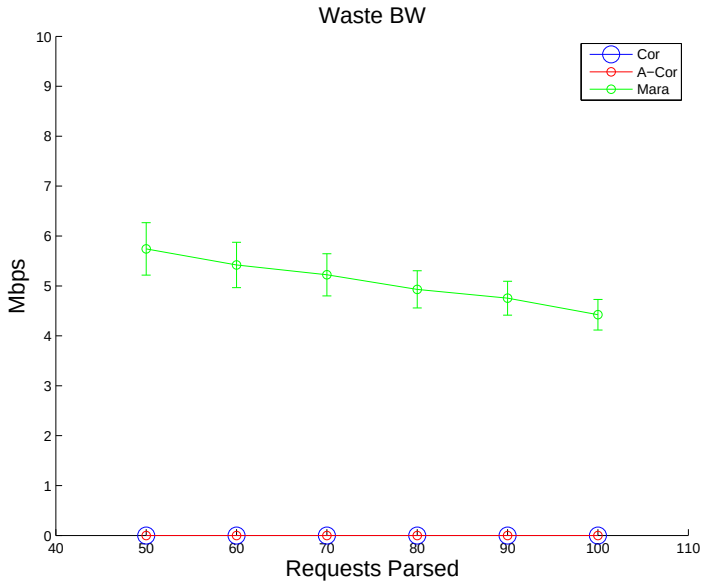


Figure 5.15: T1 - Cen. Mean Wasted Bandwidth

5.4 Distributed Scenario

For the next set of tests, all the results were simulated having the parameters defined in Table 5.2 constant.

We analyze Topology 3 results first. The number of requests was increased at each run so that the point of network congestion is achieved. The simulations were repeated for each reservation algorithm. This way we can compare and evaluate the influence of the reservation algorithm.

First we look at the number of requests denied while resources were available, Figure 5.16. We say that a request is denied while resources were available if at any time a request is denied, the total amount of free BW in a possible path is more than the request. Ideally, this should not happen as it leads to waste of resources and bad network utilization efficiency.

We can see that as the number of request increases MARA denies more and more while it has resources available. COR and A-COR do not have this design problem and therefore it is impossible for those two algorithm to deny while there are resources available.

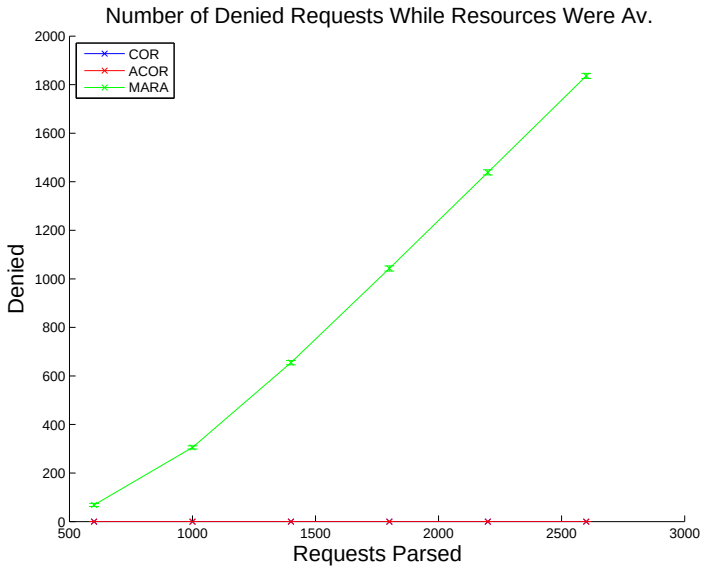


Figure 5.16: T3 - Total Denies While Resources Were Available

While MARA denies more, it also re-computes QoS parameters more often, as we can see in Figure 5.17. In this figure we plot the total number of resource re-computation events and the number of successful ones. The number of tries is growing at a steady pace after the network is fully congested (around 1700 requests). We also see that the percentage of successful re-computations is very low in MARA, and so it leads to a very high waste of resources, as we will see later on. Moreover, we can notice that there is an improvement of A-COR in relation to COR. A-COR is able to admit the same amount of requests using less re-computation events. This contributes to less signaling load.

The load plot in Figure 5.18 brings no surprises once the algorithm with more successful re-computations has more load.

Every time an algorithm needs to re-compute QoS requirements, it must trigger a VOPR Exhausted event. We look into the number of VOPR exhausted events now, in Figure 5.19. As expected from the previous figures, MARA has the highest VOPR exhausted events triggered and A-COR the lowest, although this does not translate directly into the total load of Figure 5.20.

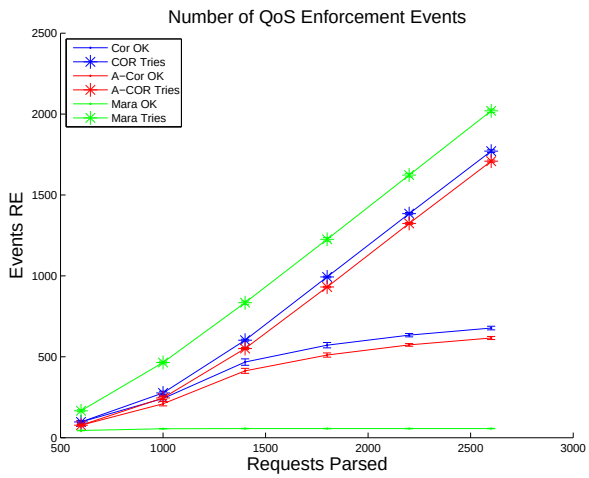


Figure 5.17: T3 - Reservation Enforcement Events

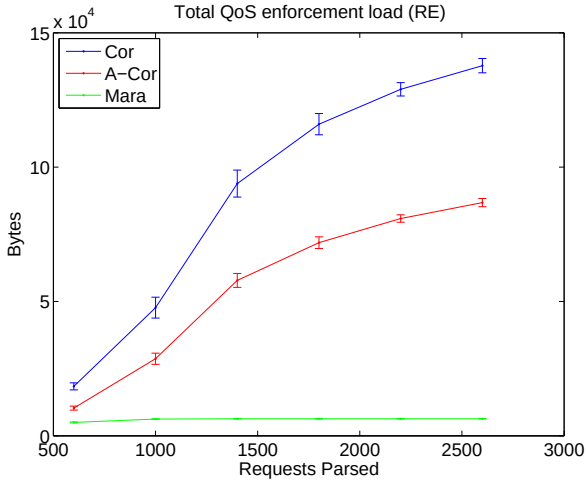


Figure 5.18: T3 - Total RE messages Load

In Figure 5.21 we can se the load per message type to make clear that each message type has a different impact on the sum. Remember that there are 3 types of messages, VOPR exhausted, VOPR Response and Re-Computation ACK. So although MARA uses more VOPR exhausted events, it also fails much more, hence the size of the first message (color blue) is bigger but the size of the responses messages (color green) is lower. The latter have much more impact on the total load. Hence, MARA has less total load than COR but still behaves worse.

The plot in Figure 5.20 is the sum of all the bars. This total load is achieved by summing together the size of all the messages exchanged. This figure is the most important regarding load, since it represents the overall overhead.

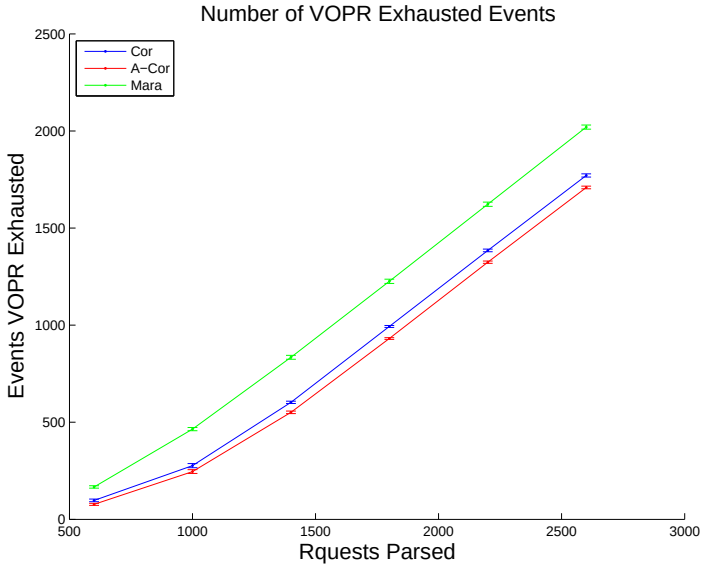


Figure 5.19: T3 - Number of VOPR Exhausted Events

Figure 5.22 and Figure 5.23 represent the network utilization efficiency. We analyze the wasted BW and the total remaining BW in all Paths. These plots take into account the mean for all paths,

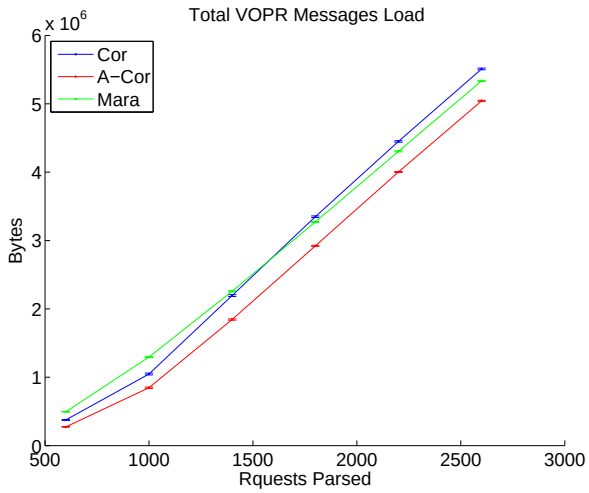


Figure 5.20: T3 - Total VOPR Messages Load

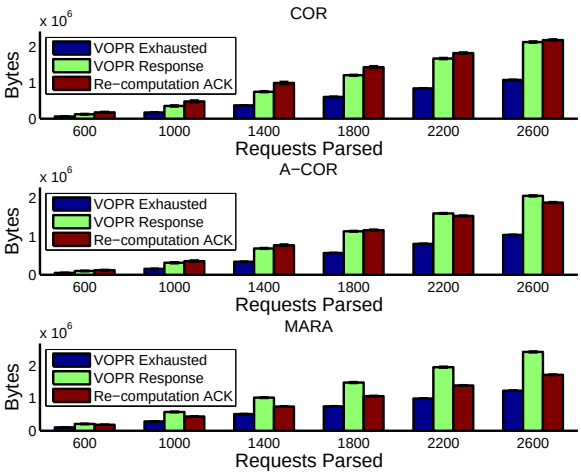


Figure 5.21: T3 - VOPR Load per Message Type

for all seeds simulated. It's obvious that MARA has a huge waste. This is due to the high number of denied requests. It's clear that MARA is not fit for a distributed network control. It was designed having centralized control in mind. From Figure 5.22 we can see that both COR and A-COR are capable of using the total amount of network resources, a major advantage.

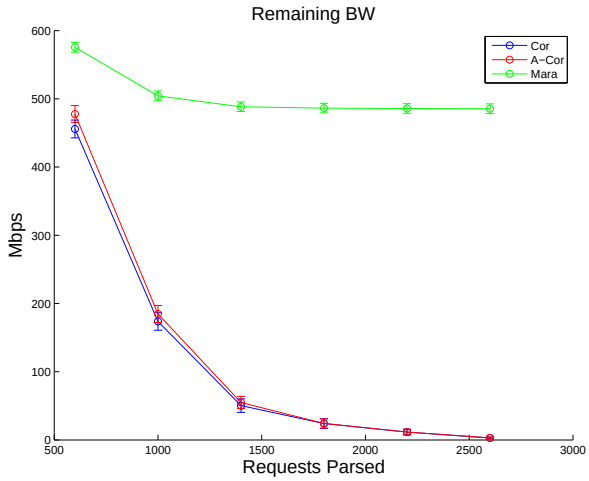


Figure 5.22: T3 - Mean Remaining Bandwidth

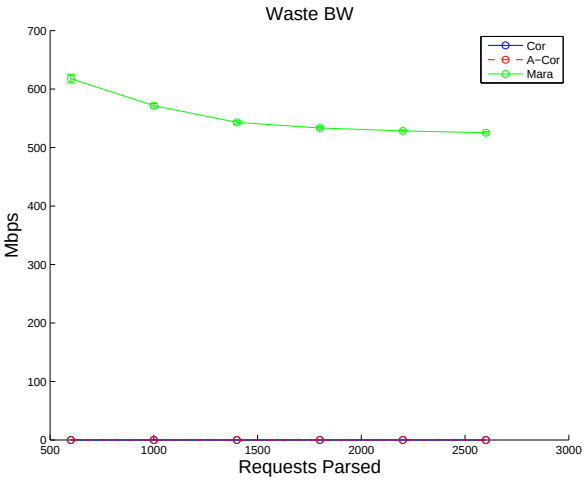


Figure 5.23: T3 - Mean Wasted Bandwidth

An important aspect of the platform studied is the impact of load minimization techniques. For instance, the use of the *LISTS* table greatly minimizes the total overall load in the network. The use of the *CORRELATIONS* table also allows the mechanism to send only the paths IDs instead of link IDs in all the messages exchanged. This reduces message size. With these mechanisms toggled *ON* and *OFF* we re-run the previous simulations and obtain the following plots of Figure 5.24 and Figure 5.25.

Figure 5.24 we can see the minimization obtained. This minimization is obtained using a topology with high correlation patterns (Topology 3). Hence, when one node needs to advertise others about a change in some path, it needs to send the advertisements to several CDPs, generating several responses and so on. If the network is not heavily correlated, then the impact of filtering is much more noticeable, as we will see in Figure 5.26 and Figure 5.27 (using Topology 2).

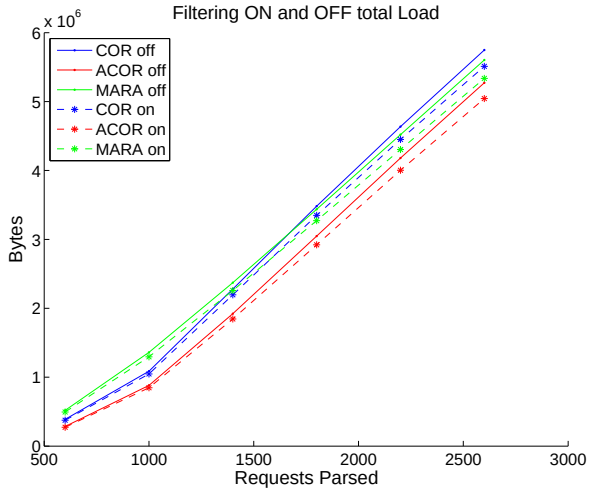


Figure 5.24: T3 - Total Load With and Without Filtering Functions

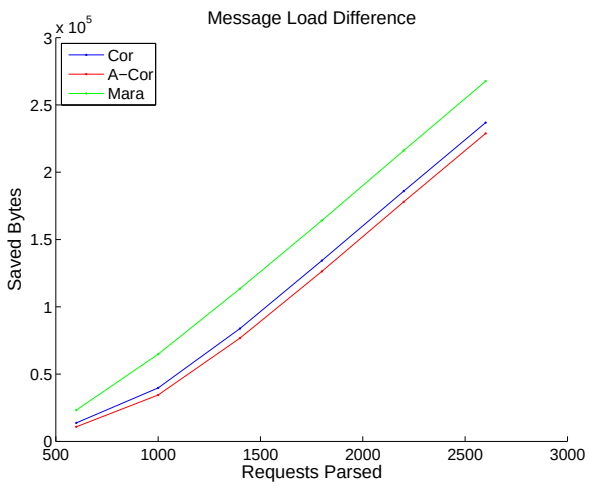


Figure 5.25: T3 - Load without minus Load with, Filtering Functions

This sums up the topology 3 (Figure 5.3) results, a network with high correlation. We can now look into topology 2, the low correlation network (Figure 5.2).

The plotted data for this network is the same as the one for network topology 3, and as such will not be explained thorough. The graphics are presented and the interpretation is similar to the ones above. However, it is important to notice the significant advantage of using Filtering techniques in topology 2, as shown by Figure 5.26 and Figure 5.27.

From Figure 5.26 we can see the minimization is much higher than for topology 3, Figure 5.24, as well the total difference is bigger and we save resources.

Notice that this is a SOMEN functionality and does not depend on the reservation algorithm used. The filtering functions are part of SOMEN and not part of the reservation algorithms. Nevertheless, the algorithm has an important role because if it trigger's less events then, we have less load.

As a small conclusion, the filtering impact may be described in two sentences. The use of the *LISTS* and *CORRELATIONS* tables allows SOMEN to minimize the total signaling load. This difference in load is closely related to the path correlation patterns in the network.

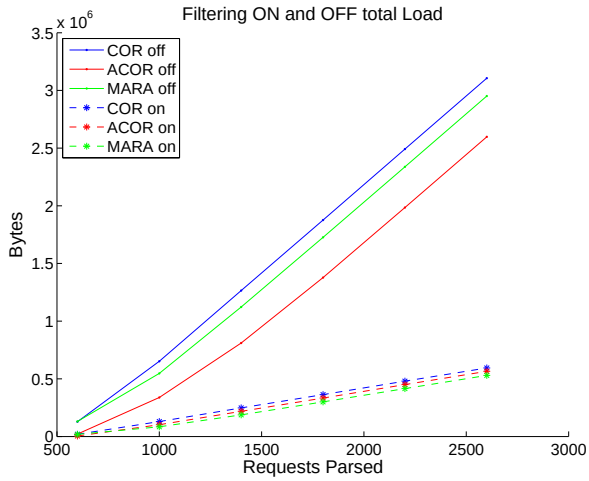


Figure 5.26: T2 - Total Load With and Without Filtering Functions

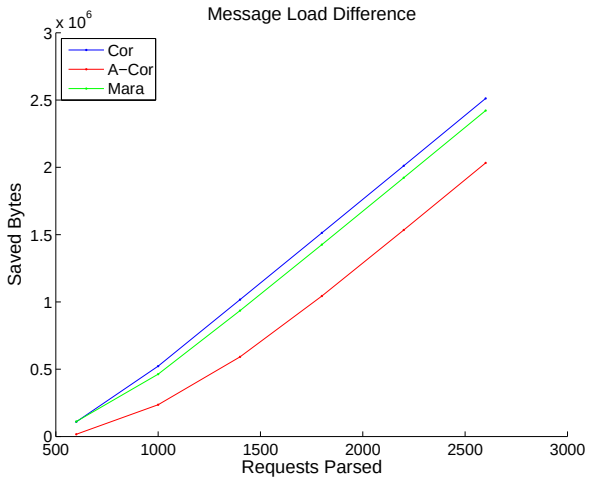


Figure 5.27: T2 - Load without minus Load with, Filtering Functions

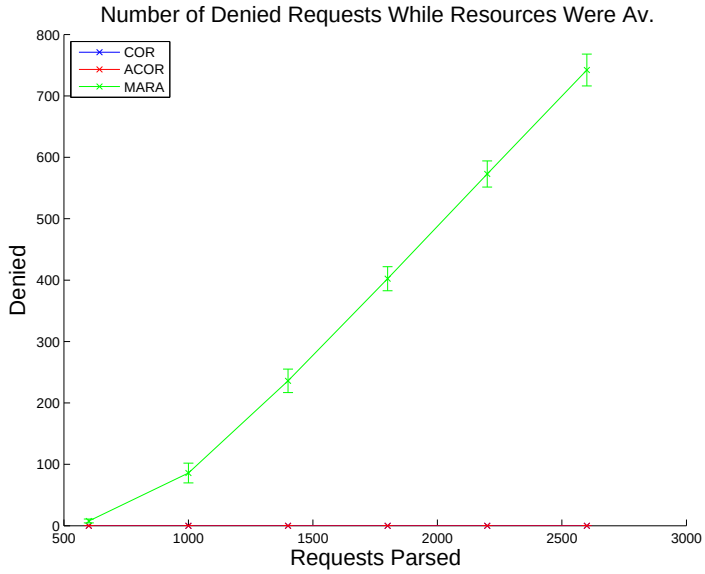


Figure 5.28: T2 - Total Denies While Resources Were Available

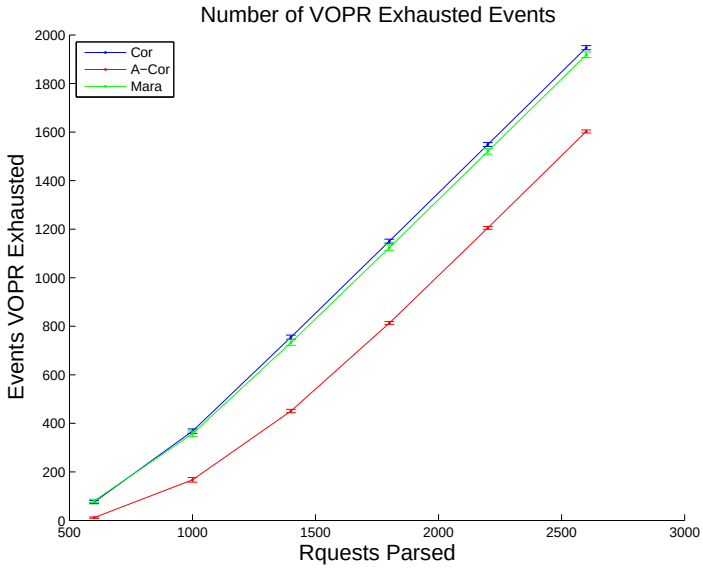


Figure 5.29: T2 - Number of VOPR Exhausted Events

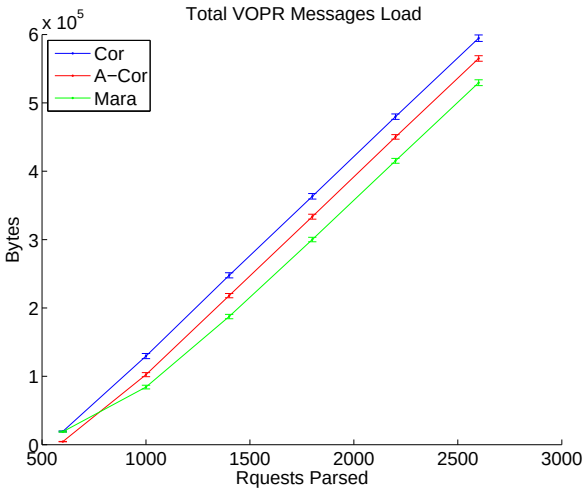


Figure 5.30: T2 - Total VOPR messages Load

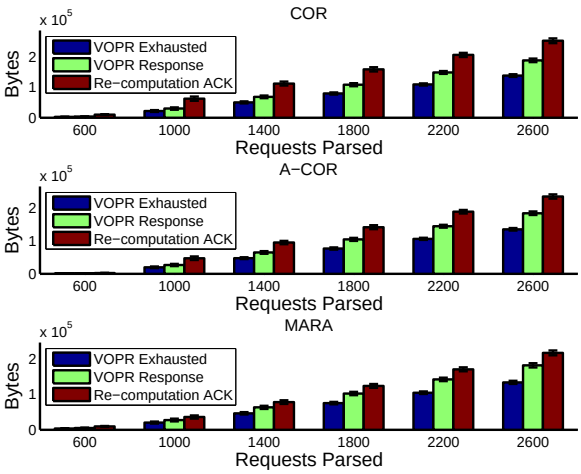


Figure 5.31: T2 - VOPR Load per message type

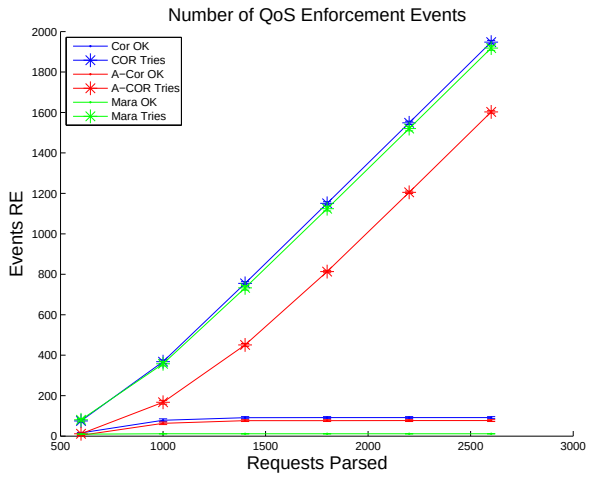


Figure 5.32: T2 - Reservation Enforcement Events

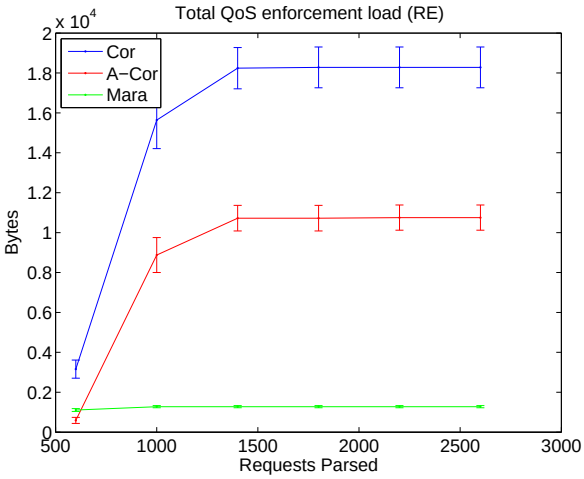


Figure 5.33: T2 - Total Reservation Enforcement messages Load

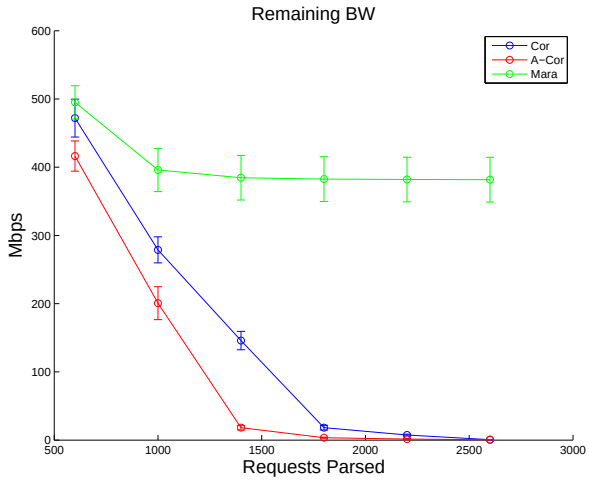


Figure 5.34: T2 - Mean Remaining BW

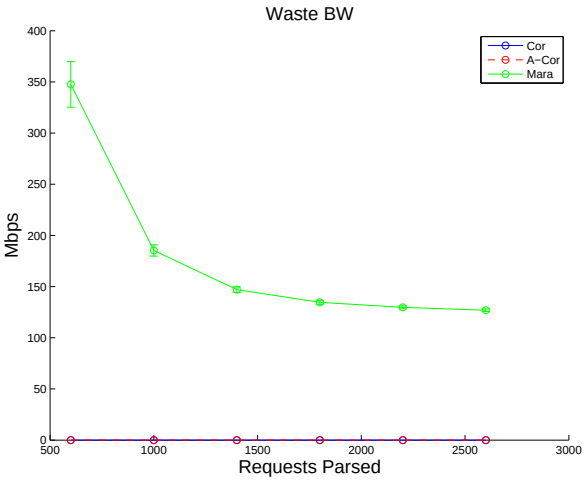


Figure 5.35: T2 - Mean Wasted Bandwidth

The last plots to be analyzed are the ones for the QoS requirements: delay and packet drops. For these simulations, the network parameters are the same as the ones for the centralized scenario topology 1, and defined in Table 5.3. The links capacity has been reduced from *1Gbps* to *10Mbps* in order to put traffic into the network. Recall, as mentioned in the centralized scenario, that if we used *1Gbps* links and put data packets flowing, the simulation time would be too high, hence the reduction. The total number of requests was reduced and the mean BW per request was also decreased.

The values plotted are obtained for 10 simulations as well, varying the NS seed. We keep the TCL seed constant and vary the NS seed, so that the events in the simulator will still be random (packets arrival and departure times in queues for example).

The delay calculations only take into account the data packets, and the overall network delay is computed as equation 5.1 describes. To compute the mean delay per CoS, we used the same procedure as for the centralized version and to compute the total number of drops, we just run the trace file for each simulation and count the number of drops.

Plot 5.36 shows the number of packet drops for all reservation algorithms. As we have said before, SOMEN ensures that no packets are dropped. If there are no resources then no traffic is admitted, hence no packet is dropped, independently of the reservation algorithm.

As for the mean delay per CoS, the weights of each class are equal (alpha values referred in chapter 3, Section 3.5) so they should all have the same treatment, hence similar delays. This is what we see from plots Figure 5.37 and Figure 5.38, that represent COR and A-COR respectively. However, there is a slight longer delay for CoS 2 due to the number of packets in that class. The requests are random and as well as the class and the amount of BW. So it is possible that for class 2 more request exist, or that the existing requests represent larger BW than in other classes, hence more packets and longer delay (queues more full). Recall that the TCL seed in kept constant so that all the algorithms (COR, A-COR and MARA) have the same set of requests, however the requests are randomly picked in that set.

This would also mean that MARA (Figure 5.39) should have higher delay for class 2, although the behavior of MARA in a distributed scenario with *1Gbps* links was already very poor, and in the *10Mbps* links it is even worse. If we look at the last plots, that represent the remaining bandwidth and the waste (Figure 5.34 and Figure 5.35), we see that MARA barely accepts any request. This means that its behavior is far from acceptable and hence no direct conclusion can be taken because the algorithm cannot be considered to perform correctly.

Regarding the overall network delay, found in plot 5.40, we see that it increases as the network utilization increases, as expected. The MARA values are lower because very few requests are accepted (final usage is less than 10% network capacity) and so, very few packets are on the network, which in turn contribute for lower delays.

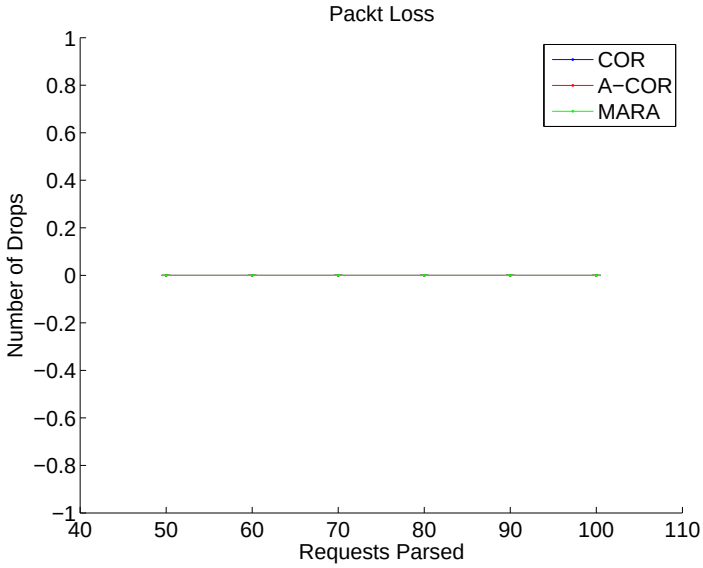


Figure 5.36: T1 - Mean Packet Drops

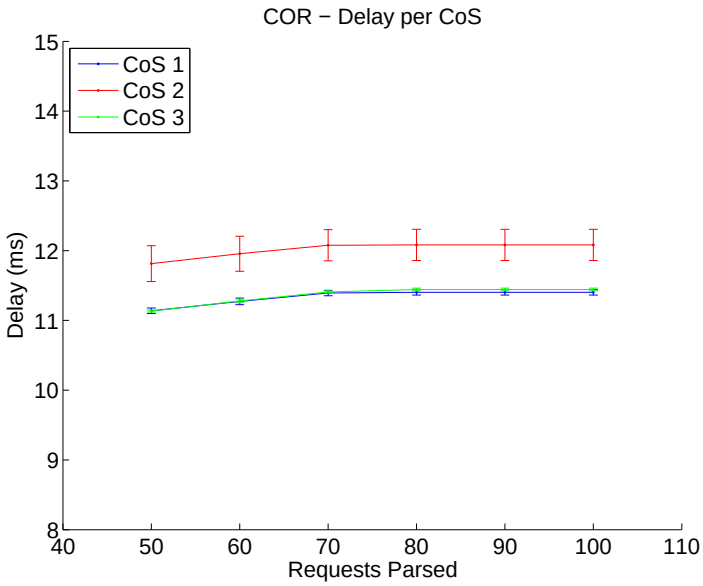


Figure 5.37: T1 - COR – Mean Delay per CoS

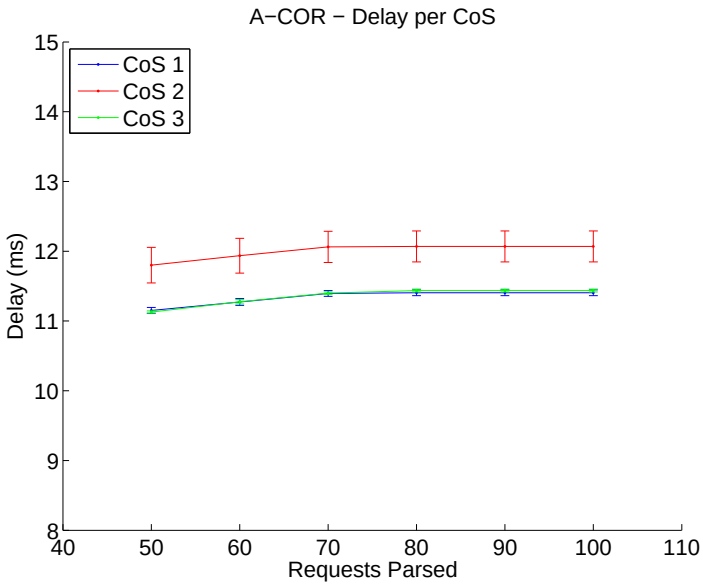


Figure 5.38: T1 - A-COR – Mean Delay per CoS

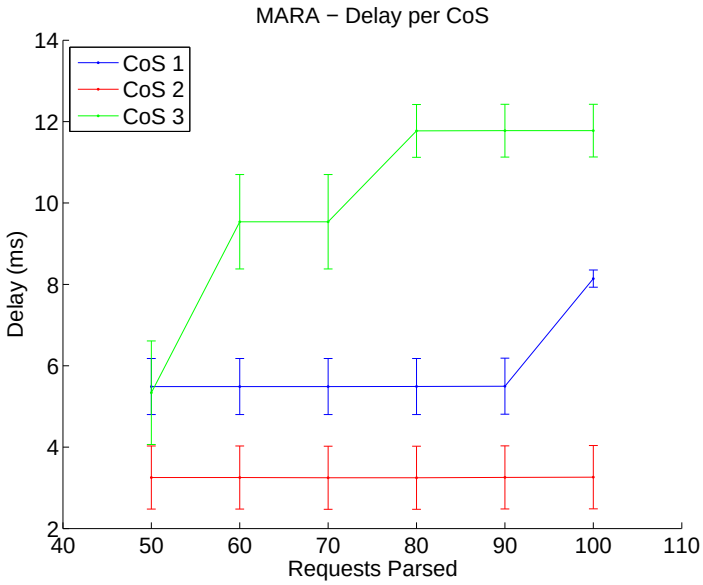


Figure 5.39: T1 - MARA – Mean Delay per CoS

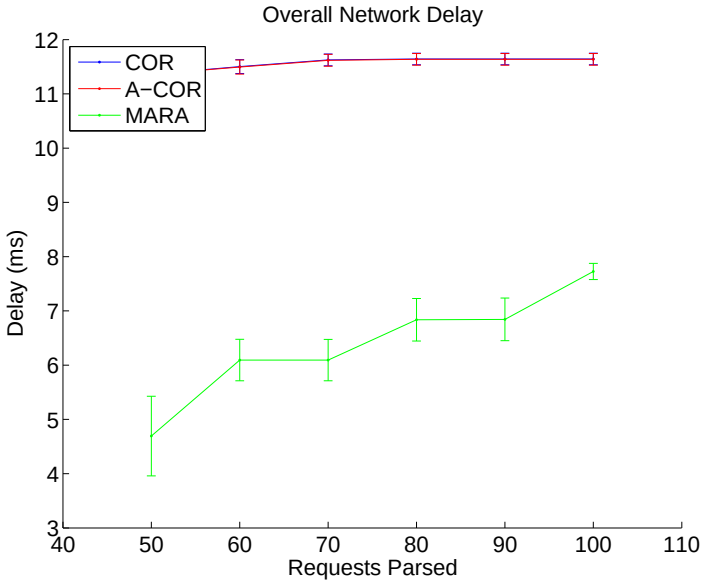


Figure 5.40: T1 - Mean Overall Network Delay

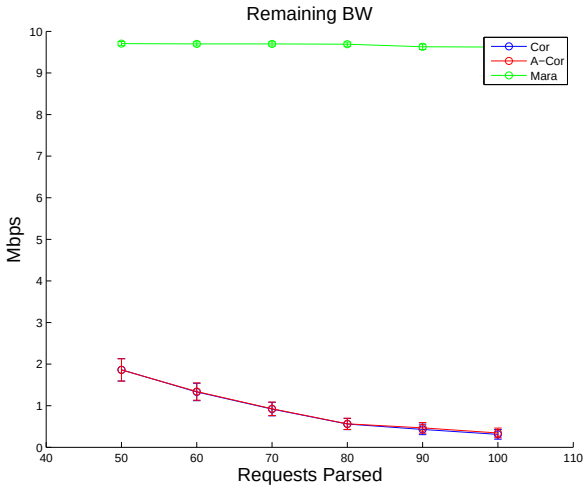


Figure 5.41: T1 - Mean Remaining Bandwidth

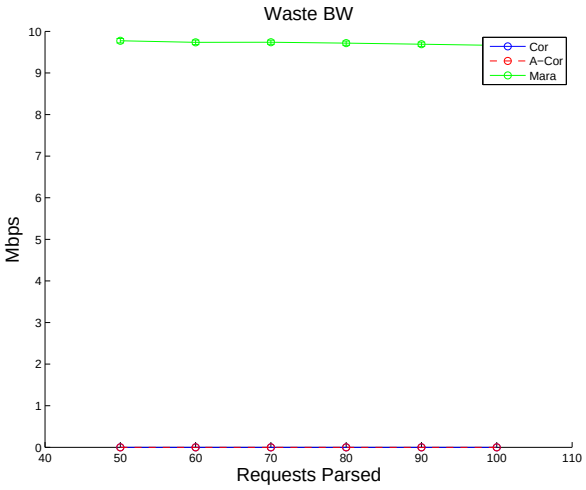


Figure 5.42: T1 - Mean Wasted Bandwidth

5.5 Conclusion

This chapter presented the main results achieved with the implementation of the algorithms described and introduced by chapter 3.

In the centralized scenario, all the algorithms tested behave in similar ways with respect to overall performance. Still, the waste of resources in MARA is a considerable flaw that both COR and A-COR have solved. This leads to a better network utilization by the latter two algorithms.

The load in A-COR is the lowest and in COR it is the highest. However, it is of paramount importance to notice that the MARA algorithm is implemented in SOMEN architecture and therefore, the periodic refreshing messages deployed in MARA, every 30 seconds, are not included in the total load generated by MARA. This means that the accurate MARA's load is supposed to be higher than that shown, as it should take the refreshing message load into account.

As for the distributed scenario, it is shown that A-COR has the best performance by far, followed by COR. It is the best algorithm in terms of control signaling minimization, and resource utilization efficiency. MARA is not suitable for distributed control and cannot be implemented with satisfactory results.

We demonstrate that the SOMEN mechanism effectively gives the network operator the possibility to enforce and take advantage of QoS requirements without any violation and no packet drops.

Chapter 6

Conclusion / Future Work

The main goal of this Dissertation's work, the implementation of the mechanisms described in chapter 3, was achieved. The platform developed (SOMEN) behaves as expected. Both COR and A-COR implementations have promising performances, being A-COR a more balanced and refined approach.

This work addresses some of the issues inherent to over-reservation schemes, with three different algorithms being implemented and tested. An overview about the fundamental aspects of over-reservation, with emphasis about QoS, multicast, and distributed network control was done, being it an integrant part of this work. The main challenges were explored and some of the arguments that make this type of approach promising were discussed.

A special effort was made in explaining the operations of the reservation algorithms used. The main aspects of the implementation were discussed. The key methods and relevant flow charts were introduced with the intent of providing the reader with an enhanced knowledge of the main implementation. The simulation environment was deeply described as well.

As for the results attained, COR and A-COR algorithms behave as expected. Regarding MARA, this algorithm is more inefficient in the distributed scenario than in the centralized one. This may be related to the way the algorithm distributes the available resources among the CoSs, as referred in chapter 5.

In terms of comparison between the reservation algorithms, it is shown that A-COR has the best performance by far, followed by COR. A-COR is the best algorithm in terms of network usage, minimizing control signaling, and resource utilization efficiency.

We demonstrate that the SOMEN mechanism effectively gives the network operator the possibility to enforce and take advantage of QoS requirements without any violation and packet drops, all in a distributed fashion, increasing scalability of the underlying network and keeping low control overhead.

With respect to future work, some functionalities were implemented in this platform but have not been discussed as they were not fully tested yet. These functionalities include an Advertisement Minimization technique that allows overall signaling load to be considerable minimized when network is near full utilization ratio, and an alternative Admission Control mechanism for the centralized scenario integrating an enhanced version of the flow re-routing. Hence, this work will be tested and evaluated in the near future. Moreover, the platform will include the System Resilience Functions (SRF) intended to improve system robustness.

Moreover, the path selection algorithm needs to be improved in order to take more parameters into account (e.g. delay, loss, jitter). For instance, traffic patterns could be analyzed and the paths that

would tend to high utilization could have more restrictions to admit requests.

“I never did anything by accident, nor did any of my inventions come by accident; they came by work. ”
— Thomas A. Edison

Bibliography

[1] Blake, S., D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss: *An Architecture for Differentiated Services*. In *IETF RFC 2475*.

[2] Bless, R.: *DARIS - Dynamic Aggregation of Reservations for Internet Services*. In *Proc. 10th Int. Conf. on Telecommunication. Systems –Modeling and Analysis (ICTSM)*, 2002.

[3] Bless, R.: *Dynamic Aggregation of Reservations for Internet Services*. In *Proc. 10th Int. Conf. on Telecommunication. Systems –Modeling and Analysis (ICTSM'10)*, 2002.

[4] Braden, R., L. Zhang, S. Berson, and S. Herzog: *Resource ReSerVation Protocol (RSVP) – Version 1 Functional Specification*. In *RFC 2205*, 1997.

[5] C++: *Programming Language*, November 2010. `http://www.cplusplus.com`.

[6] Clausen, T. and P. Jacquet: *Optimized Link State Routing Protocol (OLSR)*. In *RFC 3626*, 2003.

[7] Doxygen.org: *Generate documentation from source code*, January 2010. `http://www.stack.nl/dimitri/doxygen/index.html`.

[8] Eckel, Bruce: *Thinking In C++, Second Edition*. Electronic Book in www.bruceeckel.com.

[9] Fall, Kevin and Kannan Varadhan: *The ns Manual (formerly ns Notes and Documentation)*. In *The VINT Project, UC Berkeley, LBL, USC/ISI, and Xerox PARC*, 2000. `http://www.isi.edu/nsnam/ns/ns-documentation.html`.

[10] Greis, Mark: *Tutorial for the Network Simulator NS*, March 2010. `http://www.isi.edu/nsnam/ns/tutorial`.

[11] Liang, S.H.L.: *A New Fully Decentralized Scalable Peer-to-Peer GIS Architecture*. In *ISPRS XXIth Congress*, 2008.

[12] Logota, Evariste, Susana Sargento, and Augusto Neto: *A New Method and Apparatus for Advanced Class-based bandwidth Over-Reservation (ACOR) Control*. Submitted Patent No. 20101000072940 (Portugal), September 2010.

[13] Logota, Evariste, Susana Sargento, and Augusto Neto: *A New Strategy for Efficient Decentralized Network Control*. In *IEEE Global Telecommunications Conference, (IEEE GLOBECOM)*, 2010.

[14] Logota, Evariste, Susana Sargento, and Augusto Neto: *COR: an Efficient Class-based Resource Over-pRovivioning Mechanism for Future Networks*. In *IEEE symposium on Computers and Communications (ISCC), Riccione, Italy*, 2010.

[15] MathWorks: *Matlab vs7.9*, November 2010. http://www.mathworks.com/products/matlab/.

[16] Neto *et al.*: *Scalable Resource Provisioning for Multi-User Communications in Next Generation Networks*. In *Global Telecommunications Conference*, 2008.

[17] Neto, A., E. Cerqueira, A. Rissato, E. Monteiro, and P. Mendes: *A Resource Reservation Protocol Supporting QoS-aware Multicast Trees for Next Generation Networks*. In *Proc. 12th IEEE Symp. on Computers and Communications*, 2007.

[18] Neto, A., S. Sargento, E. Logota, J. Antoniou, and F.C Pinto: *Multiparty Session and Network Resource Control in the Context Casting (C- CAST) project*. In *Second International Workshop on Future Multimedia Networking (FMN 2009)*, 2009.

[19] Neto, Augusto José Venâncio: *Multi-service Resource Allocation in the Next Generation of Networks*. PhD thesis, University of Coimbra, 2008.

[20] NS2.31: *Main web site*, January 2010. http://www.isi.edu/nsnam/ns/.

[21] Pan, P., E. Hahne, and H. Schulzrinne: *BGRP: A Tree-Based Aggregation Protocol for Inter-domain Reservations*. In *Trans. of Communications and Networks Journal*, 2000.

[22] Pereira, V., E. Monteiro, and P. Mendes: *Evaluation of an Overlay for Source- Specific Multicast in Asymmetric Routing environment*. In *IEEE Global Telecom- munications Conference (GLOBECOM)*, 2007.

[23] Pinto, P., A. Santos, P. Amaral, and L. Bernardo: *SIDSP: Simple Inter-domain QoS Signaling Protocol*. In *Proc. IEEE Military Communications Conference*, 2007.

[24] Sargento, Susana and Rui Valadas: *Accurate estimation of capacities and cross-traffic of all links in a path using ICMP timestamps*. In *Telecommunication Systems Journal*, 2006.

[25] TCL: *Script Language*, November 2010. http://wiki.tcl.tk/.

[26] TISPAN, ETSI: *Telecommunications and Internet converged Services and Protocols for Advanced Networking*, July 2010. www.etsi.org/tispan/.

[27] TLDP.org: *Multicast over tcp*, July 2010. http://tldp.org/HOWTO/Multicast-HOWTO.html.