Early-Demultiplexing Techniques for Quality of Service in Serving GPRS Support Nodes

by

Geoffrey Lefebvre

B.A.Sc. University of Sherbrooke, 1997

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Electrical and Computer Engineering)

We accept this thesis as conforming
to the required standard

The University of British Columbia

December 2001

In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical & Computer Engineering

The University of British Columbia
Vancouver, Canada

Date January 21st, 2002

Abstract

The General Packet Radio Service(GPRS) is a wireless data service that sits on top of the existing GSM infrastructure. It provides access to the Internet and packet based billing. It is often referred as a 2.5G system and is designed to evolve to UMTS(3G). The Serving GPRS Support Node(SGSN) is one of two new nodes added to the GSM infrastructure to support GPRS. Its role is to route data to users in its geographical area and provide user mobility, access control and security.

We have studied a SGSN from a system point of view, looking at how it interacts with the operating system. We have found that a SGSN implemented as a user level process will not be able to properly support Quality of Service(QoS). We have looked at three QoS parameters: throughput, delay and precedence and have found that a SGSN will not be able to maintain guaranteed services based on those parameters when heavily loaded. This behavior is due to the fact that operating systems are unaware of the QoS requirements of incoming traffic and are therefore unable to properly queue incoming packets.

We have designed and built a SGSN where the incoming traffic flow is demultiplexed on arrival. The QoS requirements of incoming packets are identified before they are queued. Packets can now be classified and queued properly. Our results show that our Early Demultiplexing SGSN(ED-SGSN) provides large improvements. The ED-SGSN is able to maintain service guarantees in terms of throughput, delay and precedence even under serious loads.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The General Packet Radio Service (GPRS) is a new data communication service that promises to bring packet based network connectivity to the wireless world on a large deployment scale. GPRS sits on top of the existing GSM infrastructure, making it a very attractive solution for existing GSM network operators. GPRS, often referred to as a 2.5G system, is seen as a stepping stone to UMTS/3G, the next generation infrastructure for mobile telecommunication. Combined with the fact that GSM is by far the most widely used cellular technology in the world today, clearly, GPRS will have an important role to play in wireless data communication in the near future.

In order to become popular, GPRS must be able to provide Quality of Service (QoS). A system that supports QoS is able to offer services with certain guarantees and is able to hold to these guarantees in all circumstances. This is becoming increasingly important today as the usage of applications requiring such services grows. Applications such as video conferencing, packetized voice and streaming multimedia, all of which require some QoS support from the network, are already quite popular on the Internet today and it is a good assumption that these applica-

tions will also be popular with GPRS users.

Most of the QoS research in the area of networking and communication has focused on efficient link utilization and scheduling of packets so that service guarantees can be met. Often nodes that interconnect the links are modeled as black boxes with a simple processing delay and a scheduler. The fact is that system software is complex. The interaction between user level software, the operating system, and the underlying hardware is not always well understood. This is especially true in cases where Input/Output (I/O) intensive applications need to support QoS. In the area of system research, some work has been done to build operating systems with support for QoS in the I/O subsystem. However not much work has been done to try to bring networking and system research together and analyze how the behavior of a real system used as a node in a network can affect its overall performance.

In this thesis, we have built and analyzed a Serving GPRS Support Node (SGSN). The SGSN is one of two new types of nodes added to the GSM infrastructure in order to support GPRS. The SGSN is connected to the Base Station System (BSS) via Frame Relay and to the Gateway GPRS Support Node (GGSN) via an IP based GPRS backbone network. The role of the SGSN is to serve mobile users in its geographical area. It routes packets to the right cells and keeps track of the location of each user within its serving area. It also performs tasks such as mobile authentication and cyphering.

For software engineering and economic reasons, an SGSN is often implemented as a user level process using facilities provided by the operating system. This provides memory protection between processes and access to built-in libraries for the most common tasks such as memory allocation, inter-process communication and network access. The other option is to implement the SGSN as a kernel

level server. In this case the SGSN is implemented as an extension of the operating system.

Our goal was to analyze how an SGSN interacts with the operating system network subsystem and how this interaction affects the overall performance of the SGSN and the service received by each user, especially in terms of meeting QoS related guarantees. To achieve this goal, we designed and built an SGSN running under the Linux operating system. We have also built a testbed that simulates a GPRS network to test the SGSN. We believe building an actual system instead of using simulation packages is the right approach to analyzing the behavior of a user level SGSN especially under heavy traffic loads.

We have found that an SGSN implemented as a user-level server will be unable to offer guaranteed services independently of the load on the system. The quality of such service will degrade as the load on the system increases. The problem is caused by the fact that operating systems do not have a facility to allow user level processes to communicate their QoS requirements about incoming network traffic. Because they lack the proper information, operating systems are unable to classify and queue incoming network traffic according to their QoS requirements. A simple First-In First-Out (FIFO) buffering mechanism is used instead, where the QoS attributes of each packet are simply forgotten.

This problem is accentuated in GPRS by the usage of tunnels between GPRS nodes in the GPRS backbone network. Every user's traffic is aggregated into a single flow of data. Once encapsulated in such a manner, it becomes very hard to differentiate individual flows and identify their QoS requirements. The operating system hosting an SGSN is unable to do so and must resort to queuing all incoming traffic in the input queue in a FIFO manner.

The problems caused by such behavior are numerous. First, high priority packets are delayed by being queued behind lower priority packets. Second, once the queue fills up, additional incoming traffic is dropped in a tail-drop manner. This is done without accounting for the precedence or the importance that some packets might have over others. Third, packets are read from the input queue in the same order they were received and processed through the SGSN protocol stack, in that order as well. This leads to problems related to isolation of individual users' traffic when processing resources on the SGSN become saturated. Misbehaving users steal processing resources from well-behaved users and affect their throughput. Since exceeding traffic generated by misbehaving users is never sent out on the output link, the resources consumed by such traffic are wasted. Thus, overall throughput of the SGSN is affected.

To avoid theses problems, the FIFO behavior of SGSN's input queue must be eliminated. Incoming traffic must be demultiplexed on arrival and queued in a manner that allows all QoS requirements to be taken into account. Also, a scheduling framework is necessary to make sure the SGSN processing resources are distributed properly among all mobile users attached to the SGSN.

We have built a version of our SGSN where incoming traffic is demultiplexed on arrival. We identify the owner and the Qos requirements of each incoming packet before the packet is queued. This allows us to queue or drop packets according to those QoS requirements and avoid the FIFO behavior.

In this framework, packets are no longer processed in a FIFO manner. A packet scheduler decides which packet should be processed next by the SGSN instead of simply processing the first packet at the head of the queue. This allows for proper scheduling of the SGSN resources, providing better isolation between users, avoiding

unecessary delay for high priority traffic and eliminating resource wastage.

This thesis is organized as follows: Chapter 2 presents an overview of GPRS, focusing on the SGSN. A comparison of user and kernel level processes is given with an in depth presentation of the problems mentioned earlier. Chapter 3 gives an overview of some the research that has been performed on GPRS and early-demultiplexing. Chapter 4 describes the software components used to build our SGSN, such as, a memory allocator and a message passing mechanism. Chapter 5 describes our implementation of the SGSN and of the other nodes on our testbed. The most important differences between our SGSN and a commercial implementation are given in this chapter. Chapter 6 describes the design and implementation of our early-demultiplexing mechanism. Chapter 7 describes our experiments and results. The conclusion is presented in Chapter 8.

Chapter 2

Model and Problem

2.1 Introduction

The General Packet Radio Service (GPRS) is a new wireless data communication service. It sits on top of the existing GSM infrastructure and provides a higher data transfer rate than previously available. The Serving GPRS Support Node (SGSN) is one of two new nodes added to the GSM architecture to provide support for GPRS. The SGSN provides data transfer to and from mobiles in its geographical area. It keeps track of mobiles' locations and provides other management functions, such as authentication and billing. The SGSN functionality is complex, and is often implemented in software as a user-level process. We will see that such an approach can lead to problems when it comes to supporting Quality of Service (QoS).

We will first describe GPRS, its goal and functionality. We will focus mostly on the SGSN since it's the node we have studied and modeled. We will look at a possible implementation of an SGSN as a user-level server, focusing on how such software interacts with the operating system underneath.

2.2 Definitions

2.2.1 Downlink and Uplink traffic

Downlink traffic refers to traffic that travels from an external network, such as the Internet, towards mobile users. Uplink traffic travels in the opposite direction, from the mobile users to the external network.

2.2.2 Throughput

The overall throughput represents the total number of packets per second the SGSN can output. Since we are mostly interested in traffic in the downlink direction, the overall throughput usually refers to the total number of packets per second sent out on the Gb link. This number is dependant on the input load on the Gn interface, the available bandwidth on the Gb link, and on the processing capacity of the SGSN.

The throughput of a user or mobile (again in the downlink direction) represents the number of packets per second that can be transferred from the external network to the mobile user. This number depends on the individual load on the Gn interface, on the allocated bandwidth for this user and on the processing resources available on the SGSN.

2.2.3 Exceeding Traffic and Misbehaving Users

Exceeding traffic is the portion of traffic generated by a user that exceeds the user's pre-allocated bandwidth. A misbehaving user is a user that generates exceeding traffic.

2.2.4 Delay and Jitter

Delay represents the length of time a packet spends at the SGSN. Delay is due to a limitation of physical resources, such as link or processing capacity, which causes packets to be queued for a certain amount of time. Jitter represents the variation in delay. Jitter is caused by variable queuing time experienced by packets traversing the SGSN.

2.2.5 Precedence

Precedence determines how service should be maintained under abnormal situations, such as network congestion. When packets need to be dropped, the decision of which packet to drop is based on the packet's precedence. Packets with low precedence are always dropped before high precedence packets.

2.2.6 Reliability

Reliability defines the probability of errors such as packet loss, duplication, out of order delivery and corruption. Traffic with high reliability requirements should be transferred using the most reliable transport mechanism (acknowledgement, error detection, etc) and should belong to the highest reliability class.

2.3 The General Packet Radio Service (GPRS)

GPRS provides access to X.25 networks and IP based networks such as the Internet at speeds between 9 and 150 kilobits per second. For this, two new nodes are added to the GSM architecture: the Serving GPRS Support Node (SGSN) and the Gateway GPRS Support Node (GGSN). Both are anchor points for mobile users. The SGSN

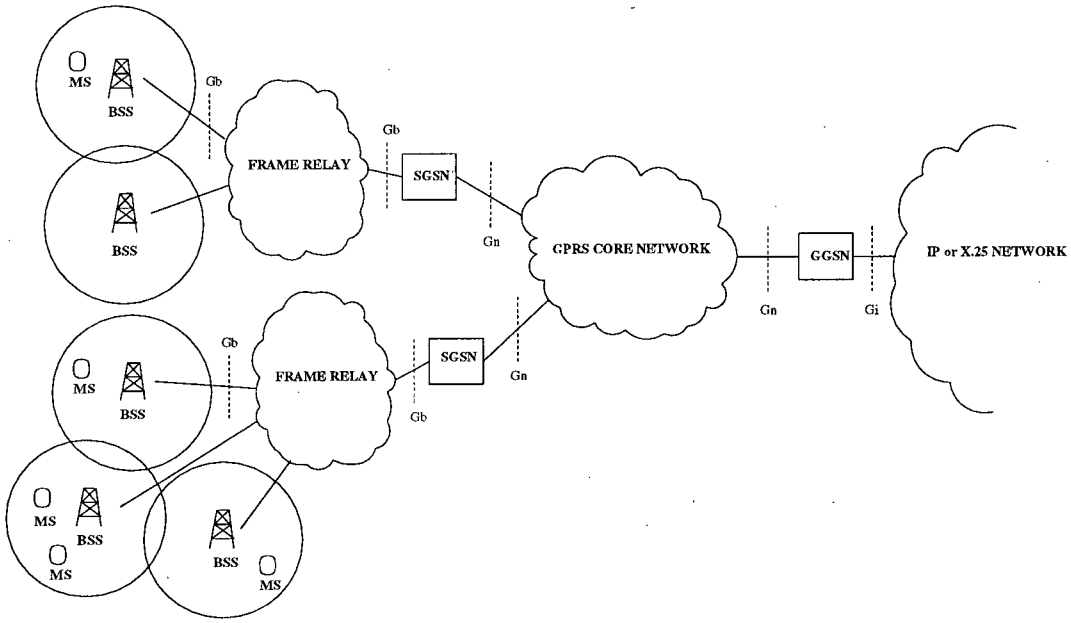


Figure 2.1: GPRS Network

is a geographical anchor. It will route packets for all mobiles in its geographical area. Still, mobiles can change geographical area and dynamically switch from one SGSN to another. The GGSN is a network anchor. It acts as a gateway between the GPRS network and an external network such as the Internet. A mobile station (MS) assigned with a packet data protocol (PDP) address (i.e. an IP or X.25 address) always communicates with the external network via the same GGSN. From the point of view of the outside network, the GGSN appears as a regular router; the GPRS network is not visible. The SGSN and the GGSN are connected together via an IP based GPRS backbone network. The SGSN is connected to the base stations using Frame Relay.

The GPRS protocol can be divided into two parts: the transmission plane and the signaling planes. The transmission plane is used for data transmission be-

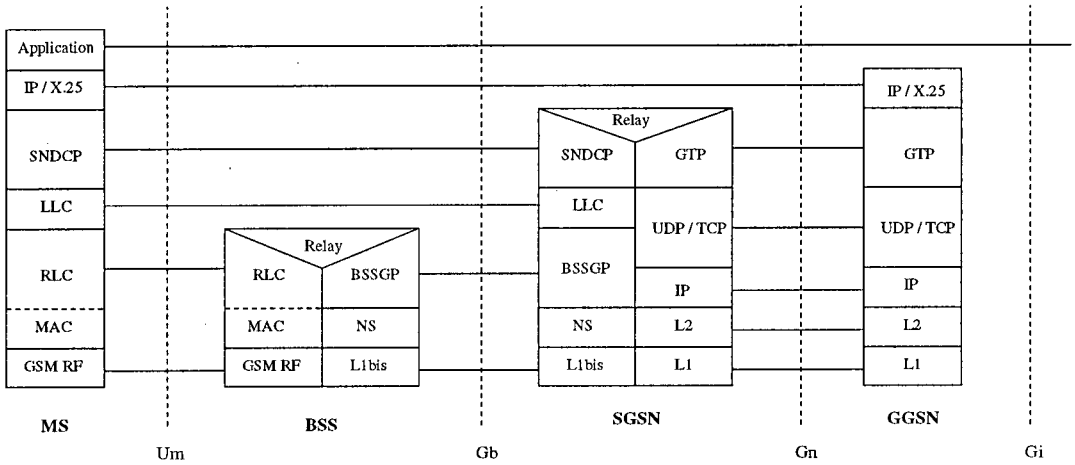


Figure 2.2: GPRS Transmission Plane

tween mobiles and the external data network. It involves four nodes: the MS, the base station subsystem (BSS), the SGSN and the GGSN. The transport of network protocol data units (N-PDU) between the mobile and the GGSN is completely transparent. Network layer packets are encapsulated and tunneled through the GPRS network. GPRS headers are added to properly route packets through the GPRS network independently of the network layer protocol. The signaling planes are used for management of the transmission plane. Tasks such as network access, user authentication and mobility control, are accomplished using the signaling planes. Signaling between the MS and the SGSN is done with the GPRS mobility management (GMM) and the session management (SM) protocols. Signaling between GSN (SGSN and GGSN) nodes is done with the GPRS tunneling protocol (GTP).

Signaling also involves additional nodes, part of the existing GSM infrastructure, such as the Home Location Register (HLR), the Mobile Switching Center (MSC) and the Visitor Location Register (VLR). Signaling between GSN and GSM nodes is done via SS7.

A mobile that wishes to connect to an external network using GPRS, must first attach itself to the SGSN. The attach procedure is part of the GMM protocol. This procedure usually requires the mobile to authenticate itself, and leads to the establishment of a logical link between the mobile and the SGSN and the creation of a mobility management (MM) context on the SGSN. The logical link is identified by a Temporary Logical Link Identifier (TLLI). The MM context is used to store the state relative to mobility management. Information, such as the International Mobile Subscriber Identity (IMSI) and the current cell used by the MS, are stored in the MM context.

An attached mobile that wishes to transfer data with an external network must first obtain a PDP address. This is done via the PDP activation procedure, part of the session management protocol (SM). This procedure involves the SGSN and the GGSN. GPRS supports the assignment of static and dynamic PDP addresses. The address assignment is made by the GGSN. A PDP context on the SGSN is created to store information such as the Network layer Service Access Point Identifier (NSAPI), the PDP address, the IP address of the GGSN and the Quality of Service (QoS) profile.

GPRS allows the multiplexing of networking protocols and addresses over a single logical link. A single MS could be assigned two different PDP addresses from two different networks and/or protocols and use them simultaneously. With each PDP address a NSAPI is assigned at the MS, the SGSN and the GGSN. The NSAPI serves as an access point to the logical link between the mobile and the SGSN. The TLLI/NSAPI pair allows for the proper routing of N-PDUs between the mobile and the SGSN. Between the SGSN and the GGSN, the IMSI/NSAPI pair is used to uniquely identify a PDP context.

GPRS supports Quality of Service (QoS). The requirements specify guidelines for support of QoS. It defines 4 parameters: precedence, reliability, delay and throughput[14] and defines classes of traffic for each parameter. GPRS defines 3 precedence classes and 3 reliability classes.

Four delay classes are defined in the requirements. Three of them have maximal delay that should be experienced by traffic belonging in each class. The fourth delay class has no bounded delay and represents a best effort service. The values given in the requirements for mean and maximal delay represent end to end delay across the entire GPRS network, not only the SGSN.

Throughput is split into 2 parameters: peak and mean throughput. Peak throughput is calculated in bytes per second and represents the maximum rate that can be transferred by a PDP context. There is no guarantee that the peak rate can be achieved or sustained. Mean throughput is calculated in bytes per hour and represents an average value that a PDP context can expect over its lifetime. In both cases, a GPRS network can limit the throughput, even if additional capacity is available. GPRS defines 9 peak throughput classes and 19 mean throughput classes.

Each PDP context is assigned a Qos profile, which is a combination of the five parameters. An implementation does not need to support all possible combinations of parameters[8]. There is one QoS profile per PDP context. The QoS profile to be used by a PDP context is negotiated and assigned during the PDP activation procedure. An example is the support of a real-time traffic class. Traffic belonging to this profile would be part of the lowest delay and reliability class and be assigned to some throughput and precedence classes.

2.4 The Serving GPRS Support Node (SGSN)

The task of the SGSN is to serve all mobiles in its geographical area. It must keep track of the location of each mobile and route packets accordingly. For security and privacy reasons, new mobiles connecting to the network must be authenticated, and the link between the mobile and the SGSN is ciphered. The SGSN supports user mobility within its geographical serving area and also between SGSNs. The SGSN is designed as a multiple layer network stack. Each layer serves a specific purpose and allows the transparent transport of N-PDUs between the GGSN and the mobile.

2.4.1 GPRS Tunneling Protocol (GTP)

The GPRS Tunneling Protocol (GTP)[11] is used to tunnel network layer PDUs between the SGSN and GGSN. Packets from all PDP contexts are multiplexed into a single stream. GTP supports acknowledged and unacknowledged transmission modes. In unacknowledged mode, GTP PDUs are transported using UDP[32]. The acknowledged mode uses TCP[33] as its transport protocol. TCP is recommended for network layer protocols requiring a reliable link such as X.25. UDP is used for protocols that don't need a reliable link such as IP. The GTP header contains a Tunnel Identifer (TID) which is a combination of the mobile's IMSI and an NSAPI. The TID identifies the mobile and the PDP context to which a packet belongs.

2.4.2 Sub Network Dependant Convergence Protocol (SNDCP)

The Sub Network Dependant Convergence Protocol (SNDCP)[13] allows the multiplexing of multiple network protocol contexts onto a single logical link provided by the LLC layer. NSAPIs are used as access points to the underlying link. SNDCP also provides an adaptation layer for the above network layer. SNDCP supports

fragmentation so packets can conform to the maximal frame length requirement of the LLC layer. SNDCP supports acknowledged and unacknowledged operations. These in turn use the corresponding service provided by the LLC layer. To maximize the usage of the radio interface, SNDCP provides data compression. The SNDCP layer currently supports TCP/IP header compression[19] and V.42bis data compression[18].

2.4.3 Logical Link Control (LLC)

The Logical Link Control (LLC) layer[12] provides a reliable ciphered logical link between the MS and the SGSN. The LLC layer can operate in acknowledged or unacknowledged mode. The retransmission and acknowledgement are performed at the LLC layer. Frames are protected from corruption with a 24 bit Cyclic Redundancy Check (CRC), calculated over the entire frame and appended at the end. In unacknowledged mode, two protection modes are supported: protected mode, which calculates the CRC over the entire frame, and unprotected mode where the CRC is only calculated over the LLC and SNDCP header. Confidentiality is provided with encryption. Encryption is performed over the entire LLC payload and the field containing the CRC. The security provided by the LLC layer is equivalent to what is currently provided by GSM.

2.4.4 BSS GPRS Protocol (BSSGP)

The role of the Base Station Subsystem GPRS Protocol (BSSGP)[9] is to provide a communication path between the SGSN and the Base Station Subsystem (BSS). BSSGP only supports unconfirmed transfers of data. Reliability, if needed, is to be provided by the LLC layer. BSSGP provides BSSGP Virtual Connections (BVC)s

that are established using Frame Relay Permanent Virtual Connections (PVC). A BVC represents a connection between an SGSN and a cell. Individual mobile logical links are multiplexed over BVCs. Along with data, packets are sent with QoS related information. This information can be used by the BSS radio resource scheduler to manage the radio interface.

BSSGP supports flow control between the SGSN and the BSS for traffic in the downlink direction. There are no requirements for flow control in the uplink direction. For each cell or BVC, there is a downlink buffer on the BSS. This buffer is controlled by a leaky bucket. The BSS also controls the rate of each MS using a leaky bucket. Flow control is performed at the SGSN on each MS and on each BVC to make sure the capacity of any leaky bucket on the BSS is never exceeded. The SGSN is kept up to date regarding changes in capacity and leak rate of each bucket by the BSS.

Flow control is first performed on each MS's traffic. PDUs that pass the mobile flow control are then subjected to the BVC flow control mechanism. Once passed this second stage, the PDU can be sent to the BSS. If a PDU cannot be sent immediately, it must be delayed on the SGSN until enough room is available at the BSS.

The flow control mechanism on the SGSN also acts as a scheduler. It selects the next packet to send to the BSS. This selection is based on the QoS profile of the PDU. The requirements do not specify an algorithm for the scheduler. This is left to the implementation.

2.4.5 Network Service (NS)

The Network Service (NS)[10] provides a link layer transport mechanism using Frame Relay Permanent Virtual Connection (PVC). This transport mechanism is used to establish BVCs. Along with data transmission, NS provides load sharing and virtual connection management.

2.5 User Space Vs Kernel Space Implementation

Most servers are usually designed to run as regular processes in user space. There are sound advantages to this, both economical and practical. There are also disadvantages, not only related to performance but also to Quality of Service support. An SGSN protocol stack can also be implemented as a user level process. This is a very likely solution, given the complexity of the requirements and the costs of software development. The alternative would be to implement it or parts of it in kernel space as an extension to the operating systems.

A user level server is a process that runs in user space. This means it runs in its own address space protected from illegal memory access from other processes. If a process wants to communicate with other processes or access hardware, it must do so using facilities provided by the operating system such as system calls.

A kernel level server runs in kernel space. It doesn't have an address space of its own. It uses the address space of the operating system kernel. This address space is also shared by device drivers, file systems, the virtual memory manager and so forth. Kernel space software has direct access to hardware registers and interrupts. It can run in process context or in system context[44]. When kernel software runs in process context, it acts on behalf of the currently running user level process, such as

when the process executes a system call or causes an exception. Interrupt handlers run in system context. They do not act on behalf of the currently running process; they are system wide. System context is often referred to as interrupt context.

There are many advantages to developing a server as a user level process. User level processes are easier to develop. Many libraries are available. Operations such as network communication, disk access and thread related operations are easier to perform from user space. User space processes are also easier to debug. Standard debuggers can be used and the system integrity is not threatened by misbehaving actions from incorrect processes. Illegal memory access in kernel space and other memory corruption bugs can easily crash an entire system.

Software designed to run in user space is also more portable. Standardized Application Programming Interfaces (APIs), such as POSIX, make it possible to design software that can be ported from one platform to another with only minor modifications. Kernel APIs are tied to the operating system they belong to, making the design of portable software a lot harder. Another important fact is, that the number of software developers who possesses kernel development skills and experience is limited. All of these factors make user-level software easier and quicker to develop, and less expensive.

Operating systems provide resource multiplexing to multiple uncooperative user level processes. User level servers do not have to worry about sharing access to system resources; this is the operating system's role. This is not true in kernel space. Servers running in kernel space must run in a cooperative manner. Resources are not multiplexed, they must be shared and contention must be explicitly resolved.

Kernel level servers do have a performance advantage. Because they reside in kernel space they do not have to copy data back and forth from user space. Data

copying is expensive because it uses the memory bus extensively, which runs at a fraction of the CPU speed nowadays. This causes numerous processor stalls. Such memory intensive operation tends to run at the speed of the memory bus instead of the speed of the processor. Copying also causes cache pollution. Still, user level servers are often seen as a more viable solution because of their software engineering and their cost advantages.

2.6 Problem

The usage of the Internet for transmission of voice, audio and video in real-time is getting more and more popular. With GPRS and soon UMTS, multimedia also comes to the wireless world. For GPRS to become a broad success, it must provide users with a pleasant experience. It needs to support Quality of Service (QoS). There are requirements for QoS in GPRS. Different parameters (delay, precedence, etc.) are defined, and for each of them a set of classes. The guarantees that each class should receive are also specified but the algorithms to provide such guarantees are not specified.

A typical user level implementation of an SGSN will use the operating system UDP/IP stack on the Gn interface. Downlink packets will be read from a UDP socket on the Gn interface and processed through the SGSN stack. They will then be scheduled according to their QoS profile at the BSSGP level. Uplink packets will be processed through the protocol stack and sent out on the Gn interface using the same UDP socket.

We argue that a user level implementation of an SGSN will not be able to support Quality of Service (QoS) under heavy loads. Support for guaranteed service in terms of priority, precedence and throughput will be affected by the load on the

system. As the load on the SGSN increases, proper support for traffic prioritization and precedence will degrade. Isolation between the different throughput classes will not be maintained when subjected to misbehaving traffic. The overall throughput of the SGSN will also be affected by the presence of misbehaving traffic.

The sources of these problems are twofold. First, data transfer between GSNs is done using tunnels with the GTP protocol[11]. The individual traffic of each user is aggregated in a single stream of data. This makes it hard to differentiate between the various flows of traffic and their QoS requirements. Tunneling has also been identified as a source of problems to support QoS in the core network[34].

Second, the First-In First-Out (FIFO) behavior of the UDP socket buffer is the source of priority inversion and other QoS related problems. Incoming packets on the Gn interface are put on the queue by the operating system independently of their priority. The operating system cannot avoid this behavior since it is unaware of the QoS requirements of incoming packets. There is currently no mechanism that allows an application to specify its QoS requirements to the operating system regarding incoming network traffic so that packets can be properly classified on arrival. The fact that the buffer behaves in a FIFO manner is not bad in itself as long as all packets in the queue have the same QoS requirements. It's the FIFO behavior combined with the usage of tunnels in the core network that causes QoS and throughput related problems.

As the load on the system increases, the average input queue length grows, increasing the priority inversion experienced by high priority traffic. Worse, when this queue fills up, additional incoming packets are dropped independently of their precedence. This problem is known as QoS cross-feeding noise[43]. Cross feeding noise occurs when different classes of traffic are multiplexed together into a single

flow which is processed independently of the QoS requirements of the individual flows that compose it.

Because packets are not classified on arrival, they are processed through the stack despite of the fact that the flow they belong to may have exceeded its allocated bandwidth. The flow control unit will drop all packets that should not be sent over the Gb link but still, these packets will already have consumed important processing resources. This is not so important when the CPU is not saturated, but once the CPU becomes the bottleneck in the system, the impact of processing packets that will be dropped at a later point in the stack has a negative impact on the throughput of the system.

Once saturated, the SGSN can only process a certain fraction of the incoming traffic. This fraction will consist of data coming from well behaved users, but can also include a certain amount of exceeding traffic coming from misbehaving users. The amount of CPU cycles wasted on packets that will be dropped at the flow control unit is proportional the quantity of exceeding traffic. The diminution of the overall throughput will also be proportional to the quantity of exceeding traffic. Another problem that appears when the SGSN becomes saturated, is that isolation between users is no longer maintained. Exceeding traffic steals processing resources from well behaved users and the latter see their individual throughput affected by this.

Using a mechanism to read packets at a lower level, such as a raw socket to skip parts of the IP stack, will not solve the problem because incoming packets are still queued in a FIFO fashion. The problem lies in the fact that incoming packets are queued independently of their QoS requirements.

A kernel level implementation can easily avoid this problem by demultiplex-

ing the incoming GTP flow in interrupt context, avoiding input FIFO queuing. Incoming packets are identified on arrival and de-multiplexed early. They can be queued according to their QoS profile and then scheduled to be processed through the reset of the SGSN stack accordingly.

We have designed and built an SGSN where packets are de-multiplexed in interrupt context and queued accordingly. The order in which packets are processed through the SGSN stack can then be scheduled according to QoS parameters, user status and network status. This provides QoS support independently of the load on the system, and provides efficient resource utilization. The Early-Demultiplexing SGSN (ED-SGSN) is built as a split user/kernel implementation. The kernel component named KSGSN identifies and queues incoming packets in system context. Most of the SGSN is still in user space. This allows the preservation of the economical and software engineering advantages of a user level implementation. The scheduling of packets to be processed through the user level SGSN stack is done in process context by the KSGSN module, when the SGSN reads a packet from the kernel. This means that only valid packets are copied to user space. Exceeding traffic is never copied to user space and processed through the SGSN stack.

Chapter 3

Background

Most of the research on support for Quality of Service (QoS) in GPRS has focused on the radio interface. This is understandable since the air interface is the most limited resource. Some work has also been done on studying QoS support over the core network. Although work has been done on evaluating different scheduling algorithmis for specific nodes in the GPRS infrastructure, an analysis of the actual performance of an SGSN has only been done in [25]. Most of the research analyzing and evaluating specific system software has been focussed on the Web[22][30].

We first present a brief overview of some of the research related to GPRS and Quality of Service. We then focus in more detail on some of the key publications in the operating system and networking areas on issues related to early demultiplexing.

[25] presents a platform architecture for the implementation of an SGSN. An overview of a potential hardware and software platform is described with an analysis of its performance for the processing of voice packets. The performance analysis is based on a simple queuing model based on hardware performance. Our approach is different. We focus on the analysis of the software architecture of the transmission plane and its interaction with the operating system. Also we opt for a testbed system

instead of a model using a simulation package. We believe that system software, because of the complex interaction with the operating system and hardware, is very hard to model using queuing theory and simulation packages although results from our testbed could be used as input in a large scale simulation.

In [35], an overview of the QoS infrastructure is given for GPRS phase 1. The current infrastructure is compared with the Universal Mobile Telecommunications System (UMTS) and its shortcomings are identified. Recommendations are made for future release of GPRS.

[31] presents an overview and analysis of different packet scheduling algorithms for GPRS. The algorithms are evaluated using a simulation model with a variable number of GSM time slots available for GPRS. [45] introduces a scheduling algorithm which dynamically allocates radio resources depending on the level of radio interference. This allows more effective use of the radio link. [42][40] present models and analysis of the GPRS radio interface.

In [38], the behavior of Web traffic and E-mail over a GPRS network is simulated. A model of the radio interface is used to account for cell usage, slot contention and transmission time. Sharing with voice users is also simulated. The wired part of the network from the base station to the Web or mail server is modeled as a simple delay. QoS metrics such as Web page download times are derived and results are compared with simulated modem access.

[34] focuses on QoS support in the GPRS core network. They identify two problems with the current GPRS architecture related to QoS support. The use of tunnels between GSN nodes (SGSN and GGSN) aggregates all individual flows and makes QoS of support in the core network non-trivial. Second, QoS profiles are assigned on an IP address basis, not on a per flow basis. We also have identified the

use of a tunnel between the SGSN and GGSN as a problem to support QoS within the SGSN. They propose two solutions, one based on differentiated services[4], the other based on integrated services using the Resource Reservation Protocol (RSVP)[5]. A model of a GPRS core network was built to analyze the integrated service solution. The model supports Web, FTP, audio and video traffic. Inter SGSN mobility is also accounted for in the simulation. The overhead of integrated services signaling is analyzed and shown to be acceptable.

Although not related to GPRS, research has been done on the need for early demultiplexing support in operating systems. This research focuses on QoS, real-time support, and on maintaining service during overload periods.

Tennenhouse [43] was one of the first to identify the problem of layered multiplexing and express the need for early demultiplexing in order for systems to properly support Quality of Service (QoS). Network protocols are designed in a layered fashion. Each layer offers service to the layer above, along with access point or associations. Associations are necessary to identify the layer or application above from the point of view of the layer below. Because all traffic goes out through the same interface as a single bit stream, all associations must eventually be aggregated. This often means individual associations are multiplexed together into a single aggregated association. This aggregation can lead to QoS cross talk. QoS cross talk occurs when "the network performance experienced by an application is unduly affected by the traffic pattern of parallel tributaries with which it is multiplexed."

The problem arises when QoS requirements are assigned to individual application associations. Layers below are unaware of these requirements and therefore unable to take them into account when processing the aggregated traffic. On reception, packets must go through every layer before their QoS requirements are known.

High priority packets are delayed because they are processed as regular traffic until they reach the application. The problem gets worse when misbehaving application are involved. Once multiplexed, misbehaving traffic is indiscernible and therefore cannot be dealt with by the lower layers.

According to Tennenhouse, multiplexing should be done once at the lowest layer. Each application association should have its own logical vertical stack. Outgoing packets should only be multiplexed at the data link layer. Incoming messages should be immediately demutliplexed on an application association basis and passed to the corresponding protocol stack. This allows proper isolation of independent traffic flows and minimizes QoS cross talk. It is now possible to do the processing of each incoming packet at a priority relevant to the target application's priority. A threaded model is suggested where a thread is assigned per application or per association to process packets through the network stack in the context of the application.

The main ideas behind using early demultiplexing in an SGSN are based on the concepts derived in this paper. Our goal is to improve QoS support by providing better isolation between individual mobiles. We demultiplex incoming traffic on arrival on a per mobile association in interrupt context. Instead of using a threaded approach we used a single thread with a packet scheduler. The packet scheduler acts globally on behalf of all mobiles to determine the next packet to be processed through the SGSN stack. We believe a single thread approach is a more scalable and efficient approach. The usage of a multithreaded approach could be justified in a hard real-time scenario.

Other people have used early demultiplexing to solve system instability during overload periods and the lack of resource accounting in the network subsystem

of most operating systems.

Lazy Receiver Processing (LRP)[7] is a network subsystem designed to maintain system performance during overload periods. In UNIX and many other operating systems, the network subsystem is interrupt driven. This gives the processing of incoming packets a higher priority than any other application in the system. During overload periods, this leads to wasted resources and possible starvation of user processes. LRP moves the processing of incoming packets into the context of the destination application. This leads to better resource accounting and maintaining throughput during overload periods. The paper describes a new implementation of the TCP/IP stack using LRP.

Under UNIX and many other Operating Systems, an interrupt signals the arrival of a packet from the network. The interrupt handler copies the packet from the network adapter to memory and posts a software interrupt. The software interrupt's role is to process the packet through the protocol stack and put the packet in the buffer of the target application. If the queue is full, the packet is dropped. Both hardware and software interrupts run in system context and have strict priority over any user processes. Finally, the target process copies the packet from the kernel buffer into its address space. This is done in user context, when the application makes a system call. The processing time for both interrupts is charged to the application running at the moment the interrupts occurred, not to the destination application.

Packet reception, handled as described before, leads to several problems. First, resources are wasted when packets are dropped because their target buffer is full. Packets are dropped after a non-negligible amount of resources has been invested in processing it. In extreme overload situations, this can lead to a state

known as receiver livelock. A system is in this state when it spends all its time in interrupt context processing incoming packets, only to drop them because their target input queues are full. The queues are full because user processes never get a chance to run. The amount of useful work done by the system drops to zero.

The lack of accounting for processing done in interrupt context leads to an unfairness problem. Processes which make heavy usage of the network subsystem end up stealing processor time from other processes. By moving the processing of packets in the context of their targets, most of these problems are solved or their impact is greatly reduced. Incoming packets are mapped to their destination process on reception and queued immediately. If the queue is full, the packet is dropped with a minimum of wasted processing time. This prevents situations such as receiver livelock. When the application runs, it processes the packet through the protocol stack and copies it into its address space. Processing is done in user context, meaning resources consumed are charged to the destination process and the priority of the process is taken into account.

To be able to process incoming traffic in user context, incoming packets must be mapped to their target application as soon as they arrive. To demultiplex incoming traffic, a hardware and a software solution are proposed. The hardware solution uses a specialized network adapter with a programmable processor to do the demultiplexing. The software solution does the demultiplexing in the interrupt handler.

The goals of LRP are different from ours. Their goal is to avoid wasting resources within the network protocol stack so that the receiver livelock state and denial of service attack can be avoided. Both of these situations occur in extreme cases of overload. Because incoming packets are still processed through the IP stack

in interrupt context, the ED-SGSN could still be prone to a receiver livelock in the case of a severe overload. This is unlikely since the GPRS backbone network is a private network, but, if this problem was to be avoided, a hardware based solution using a similar programmable processor as in LRP would be appropriate.

The goal of ED-SGSN is to properly schedule packets so that wasting resources and priority inversions are avoided on the Gb side of the SGSN protocol stack. Priority inversion can still occur on the Gn side of the SGSN protocol stack. This is acceptable in our case since this priority inversion is about an order of magnitude less than the potential priority inversion than can occur on the Gb side of the protocol stack. This priority inversion can be eliminated by using a programmable network adapter to do the demulitplxing.

Other research projects in the area of systems have identified early-demultiplexing as an important requirement to support QoS. In the Scout operating system[27], the importance of segregating work early to have separate input queues instead of a single shared queue is deemed to be a critical requirement for support of Quality of Service. Real-Time Mach[21] puts the network protocol into libraries that are linked into the application address space. Processing of packets is done at the priority of the application in user context. A packet filter is used to demultiplex incoming traffic to its target application. This provides more accurate resource usage and allows the inclusion of network packet processing as part of the overall system scheduling. Early-demultiplexing and LRP are used in the QLinux Multimedia Operating System[41] to support accurate resource accounting. In [17], real-time upcalls (RTU) are used to efficiently implement network processing at the user level. RTUs are non preemptible, like interrupts, but their resource usage is accounted for and guaranteed by the operating system. Again, early-demultiplexing

is used to associate incoming packets with the right RTU.

Mogul and Ramakrishnan[26] present a different approach to avoiding receiver livelock in interrupt-driven systems. Their solution is geared towards host-based routers and firewalls. During overload periods, they propose to disable interrupts and switch to a polling thread. A network card only issues an interrupt once to notify that it has received one or more packets. Once this interrupt is processed, interrupts are not re-enabled on the card until the polling thread has processed the packets on the card. This solves the receiver livelock problem caused by hardware interrupts. The polling thread processes packets through the IP protocol stack and queues them on the proper destination queue. This queue can belong to a user-level application or to a network interface in the case of a host-based router. If this queue is full the packet is dropped. This is a waste of the resources that have been invested in the packet so far. If the output queue belongs to a network interface, the problem is not so bad because the hardware will keep sending and draining packets in its queue but if the packets are destined to a user process, the receiver livelock problem arises again, this time caused by the polling thread. To avoid this situation, they propose a feedback mechanism to notify the polling thread when an application queue is full. Incoming packets are no longer read from the network card until the queue is below a certain level. Of course, this only works in cases where every incoming packet in the system must be queued on the same application queue, such as a user-level firewall.

This is similar conceptually to the feedback mechanism used in the ED-SGSN. The goal is the same but the mechanism in the SGSN is more fine grained. The feedback mechanism works on a mobile basis. The flow control unit can block any individual input queue representing a single mobile.

In [15], Floyd describes a problem referred to as congestion collapse from undelivered packets. Congestion collapse is defined as a situation where an increase in traffic results in a lower throughput. Congestion collapse from undelivered packets arises when bandwidth is wasted by delivering packets through the network that will be dropped before reaching their final destination. A regular user level SGSN can experience a similar problem by wasting processing resources on packets that will be dropped by the flow control unit. This can lead to a throughput reduction on the SGSN.

Chapter 4

System Software Architecture

In order to build an SGSN, we first designed software components used as building blocks on which the SGSN's GPRS functionality was implemented. These software components were designed to handle all the necessary operations in a multi-layer user-level network stack. These were designed to be fast and efficient to maximize throughput. We will first look at the overall architecture and justify why we used such an architecture. We will then describe the most important components we have built and explain why they are required and how they fit in this general software architecture.

4.1 Event-Driven Process

Our SGSN is designed as a single-thread event-driven process. In an event-driven process, a single thread waits for events and processes them as they arrive. When an event occurs, the thread determines the nature of the event and calls the appropriate handler. The event is processed to completion and the system goes on to handle the next event. This is in contrast to a threaded server where a new thread is created

to handle each request.

Single threaded event-driven servers are fast [3] [30] because they require no synchronization primitives and have no context-switching overhead. They are efficient and usually perform better than multi-threaded servers[29]. On symmetric multi-processor (SMP) machines, they require one thread per processor to take advantage of the parallelism offered by the hardware. In this case synchronization primitives are required to protect critical sections.

Although efficient, the event-driven approach imposes some requirements that must be met in order for the system to perform well. The multi-threaded approach doesn't have such requirements and is often easier to program but is inadequate in our case since the number of requests is high and the requests are short lived. The large number of threads would waste substantial resources due to operating system overhead. The requirements of an SGSN are closer to those of a router than to those of a Web server, making it more suited to the event-driven approach.

For the server to use only a single thread, there must be only one location where event notification can occur, for instance, where the thread can wait to be notified of incoming events. This requires the operating system to support an event multiplexing mechanism, such as a system call that allows a thread to block and wait for any event the process is interested in. Under UNIX[37], the *select()* and *poll()* system calls can be used as a single event handling point[39]. They allow registering a list of file descriptors and events of interest (ready to read, ready to write) for each of them. A file descriptor represents a handle to an input/output (I/O) entity. It may represent a file, a network connection (socket) or a communication channel between two processes (pipe, message queue), and so forth. When a thread calls *select()*

or *poll()*, it will sleep until an event occurs on one or more of the file descriptors it registered. The call returns with a list of file descriptors on which events have occurred. The application can then process each event.

A single thread event-driven process can never block anywhere except at its event handling point. If so, the entire process blocks and no other events can be processed during this period. A thread can block for two reasons: when a blocking system call is performed or when it has caused a page fault. The first cause can be avoided by using non-blocking system calls for potentially blocking operations. System calls such as *read()* and *write()* will usually block if there is no data available or no space to write data. Non-blocking system calls return an error message instead of blocking. File descriptors can be set to operate in a non-blocking fashion using the *fcntl()* system call[39].

A thread that causes a page fault will block if the page in question needs to be brought from disk[1]. It is possible to lock the memory used by a process using *mlockall()*[16]. This prevents locked pages from being swapped out to disk but doesn't entirely prevent page faults. Page mapped into the process's address space after calling *mlockall()* will still generate page faults when accessed for the first time. Usually such page faults only require the operating system to map a page frame to the process's address space but if memory resources are scarce, the operating system might have to swap out a page to disk from another process to accommodate the faulting process. This is of primary concern for the heap, which often accesses new pages at runtime. To avoid this, all dynamic memory the application plans on using should be allocated at initialization before calling *mlockall()*.

The SGSN locks its address space to avoid getting any of its pages swapped to disk. To prevent page faults at run time, all memory is pre-allocated when the

SGSN is started. Runtime memory allocation is managed using our own memory management system, which grabs memory from the pre-allocated pools. The SGSN memory management system is described in section 4.4.

One of the complex aspects to an event-driven process is dealing with events that cannot be processed to completion. Sometimes, to complete the handling of an event, the process must block until another event occurs, such as a timer expiration. In a threaded server, this is very easy to handle. The thread handling the event simply sleeps until the required event occurs. An event-driven process is not allowed to sleep anywhere except, at its event-handling point. To handle this case, the event handler must somehow put the partially processed event aside and come back later to finish processing it. The SGSN uses a service queue, which is described in section 4.2, to deal with events that cannot be processed to completion.

4.2 Event Scheduler

The event scheduler is the core component of our SGSN architecture. It manages the dispatching of all incoming events and the completion of delayed events. It monitors all file descriptors using a variation of *select()*. All event handlers are called by the event scheduler. Sometimes an event cannot be processed to completion without blocking or be delayed. A good example is flow control. If a packet cannot be sent out immediately because there is no bandwidth available, it needs to be delayed until there is enough room to send it. The handler cannot sleep or spin until then. It must allow other events to be processed. The handler must somehow terminate immediately and come back later to try again. To remedy to this situation, the event scheduler uses a service queue.

The service queue allows the processing of events to be completed at a later

time. Any module that can potentially block implements a service method. The service method is meant to be called at a later time to finish processing incomplete events. When a module notices that it cannot finish processing an event it registers itself with the event scheduler. The event scheduler adds the module to the service queue. At every service interval, the service timer expires and the service queue is processed. Currently the service queue is processed every 50 milliseconds. The service methods of each module on the queue are called one after the other. The service methods are also not allowed to block. If they cannot complete the event, they must terminate and ask to be put on the service queue again for the next round.

4.3 STREAMS Message

To pass packets from one protocol layer to the next in a uniform fashion and simplify adding headers and trailers to packets as they traverse the protocol stack, a messaging framework is necessary. The SGSN's messaging framework is derived from STREAMS[36]. STREAMS was originally developed by Dennis Ritchie. It was designed as a general framework to implement character device drivers. We use the STREAMS message data structures. We also borrow the concept of the service queue explained in Section 4.2, from STREAMS. The STREAMS message data structures are well suited for layered applications such as an SGSN protocol stack. They allow the easy adding of headers and trailers to a message without copying the message into a new, larger buffer, or having knowledge of the total space to be required by a packet when its first buffer is allocated. The data structures also allow virtual copying of messages and queuing of packets.

A STREAMS basic message is composed of three objects: a struct *msgb*, a

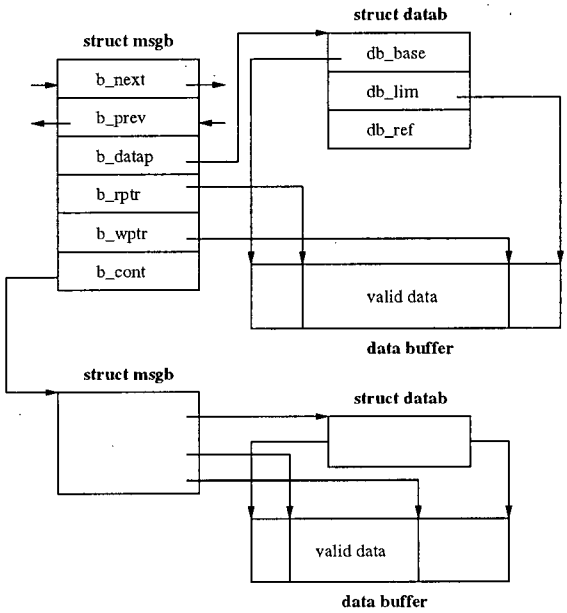


Figure 4.1: STREAMS message

struct *datab* and a data buffer. The *msgb* provides a handle to a message. It is the data structure used to pass around messages. The *datab* contains information about the actual buffer used to store the data. The data buffer is simply an allocated zone of memory where the actual data is stored. The data structures we use are very similar. They have the same name but are missing some fields, compared to their STREAMS equivalents. Figure 4.1 shows our version of a STREAMS message. We only use the fields necessary for our purposes.

In the *msgb* data structure, the *b_next* and *b_prev* pointers lare used to link multiple messages. Message queues are implemented as double linked lists using these two fields. The *b_cont* pointer is used to chain the different parts of the same message. This is how headers, the payload, and trailers of a message are linked together. When a message is passed around, only a pointer to the first *msgb* structure is required. The *b_datap* fields point to a struct *datab*.

The *db_base* and *db_lim* fields of the *datab* structure point respectively to the beginning and end of the data buffer. These two pointers never change for the lifetime of the data buffer. The *b_rptr* and *b_wptr* field in the *msgb* structure point to the beginning and end of the useful data contained in the buffer. When a message is allocated, both *b_rptr* and *b_wptr* point to the beginning of the buffer (*db_base*). As data is written in the buffer, *b_wptr* is moved accordingly. If data is removed from the front of the buffer, *b_rptr* is moved toward *b_wptr*.

The *db_ref* field contains the reference count of the *datab* and its associated data buffer. It allows multiple *msgb* to share the same *datab*, providing virtual copies of the same message. The packet fragmentation done at the SNDCP layer uses this facility to avoid copying each packet fragment into a separate buffer.

4.4 Memory Management

As explained in section 4.1, dynamic memory allocation may cause page faults when the operating system needs to map page frames into the application's address space. If no page frames are available, the operating system will need to swap some pages to disk to obtain the needed space. This is a problem since our entire process is blocked during this time. To avoid dynamic memory allocation at runtime, all memory is pre-allocated when the SGSN is started and runtime buffer allocation is managed by the server's own memory allocation system. This system is based on the McKusick-Karels[24] allocator, which is used in UNIX variants such as 4.4BSD[23].

Instead of using one large block of memory to handle every request, buffers are allocated from different free lists, depending on the requested size. There is a free list of buffers for each power of two size. The free lists are maintained in an array indexable by the size of the buffer in powers of two. The free list is a linked list of

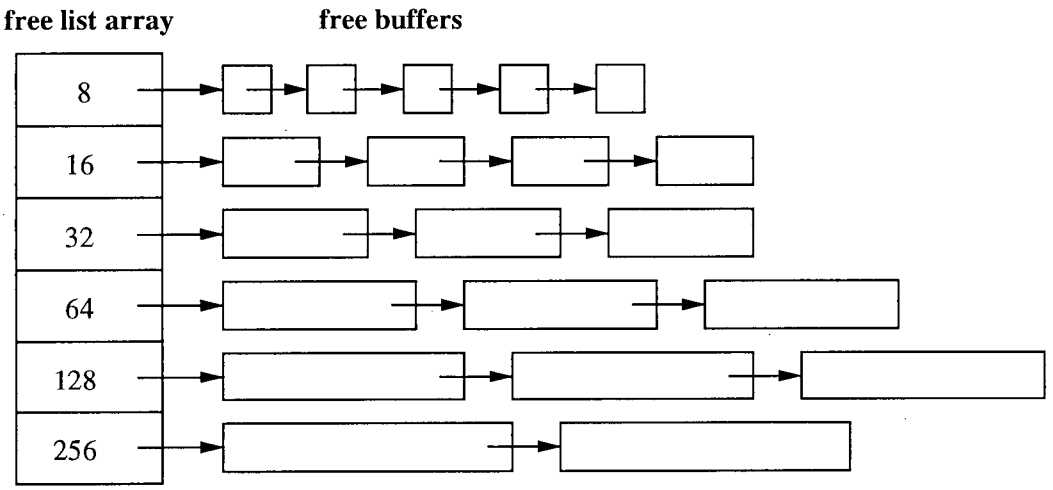


Figure 4.2: Memory allocator

buffers. When a buffer is in the free list, its first four bytes are used to store the link list pointer. This system is fast because allocation and de-allocation operations are in the order of O(1). The main disadvantage of this system is that it cannot handle arbitrarily large memory requests since it is limited by the maximum allocated buffer size. This is not really a problem for an SGSN, since the maximum packet size is known. The memory allocation system is integrated with our STREAMS message framework. We use our memory allocator to build free lists of *msgb* blocks and *datab* blocks

When a buffer is requested, its size is rounded to the next power of two and a buffer is removed from the front of the corresponding free list. A struct *datab* is also allocated from the *datab* free list. The *datab* pointers are set and a pointer to the *datab* is returned.

To release a buffer, we need to re-insert it into the free list corresponding to its size. A buffer is always released by giving a pointer to its struct *datab*. This makes it easy to determine the size of the allocated buffer. The size of the buffer

is rounded to the next power of two to get the index in the array of free lists. The buffer is then re-inserted at the front of its free list. The *datab* is also put back on the *datab* free list. Figure 4.2 gives an illustration of how the memory allocation mechanism works.

Chapter 5

GPRS Implementation

The GPRS tesbed runs on personal computers (PC) under Linux. There are three software components in the testbed. The SGSN, The GGSN/GN and the MS/GB. The GGSN/GN and the MS/GB are used as traffic generators and/or traffic sinks. In our current lab setup each component runs on a different machine connected via switched Ethernet. It is also possible to run all components on one machine or run multiple instances of each component on multiple machines.

5.1 SGSN

The SGSN implementation is based on the requirements described in [8]. Since our goals are only experimental and not commercial, only a subset of the SGSN functionality is implemented. We will discuss our implementation and the differences with a commercial SGSN. The testbed SGSN supports data compression and TCP/IP header compression. Cyclic redundancy check and a very basic form of encryption are implemented. The SGSN was built using the components described in Chapter 4. To keep things modular, we have implemented each layer as a sepa-

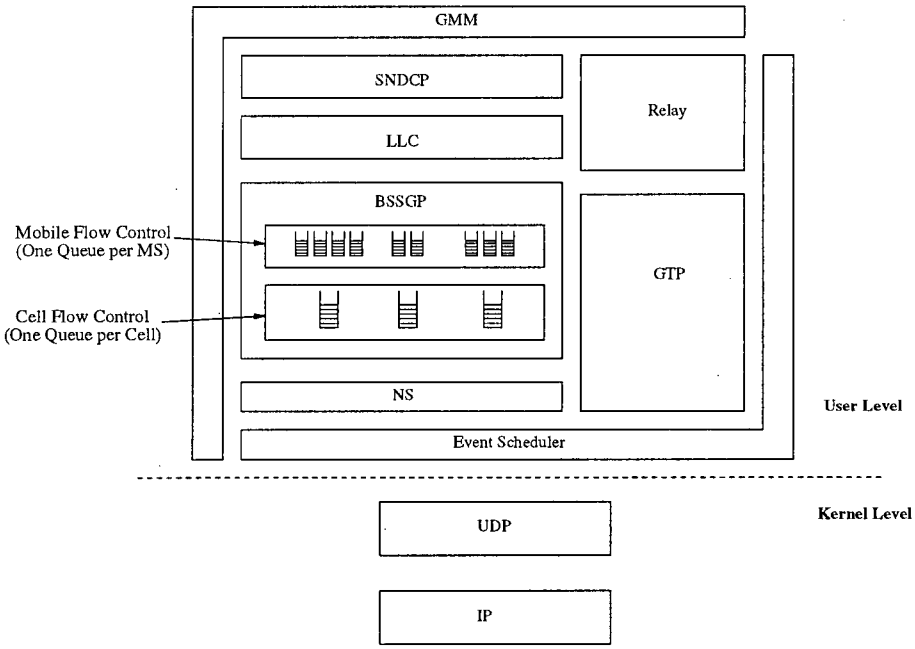


Figure 5.1: SGSN software architecture

rate software module. Still, interlayer communication is done through function calls for performance reasons[6]. Figure 5.1 shows an overview of the different software module that compose the SGSN.

The two main differences between a commercial system and our implementation are the omission of the signaling planes and the use of Ethernet instead of Frame Relay on the Gb side. Our main interests in these experiments were related to the transmission plane, so signaling was not a major issue. Because Frame Relay was replaced with Ethernet on the Gb link, the two bottom layers of the stack (BSSGP and NS) are quite different than that specified in the GPRS requirements. Because we limit the transmission rate over the Gb link using flow control, the difference in link capacity between Ethernet and Frame Relay is not an issue in our experiments. Only the unreliable, unacknowledged data transmission mode is implemented, and

re-ordering of packets is not supported. This simplifies our implementation because it reduces the complexity of the state to be kept for each mobile. We believe the lack of retransmission doesn't significantly affect our results since we are mainly interested in analyzing the transfer of packets inside the SGSN software and its interaction with the operating system kernel.

5.1.1 Context Management

The state of each mobile is stored in a mobility management (MM) context and a packet data protocol (PDP) context. The MM and PDP contexts are kept in the GPRS mobility management (GMM) module. In our SGSN implementation, the MM and PDP contexts are created at initialization. A configuration file containing information about each mobile (QoS profile, PDP address, IMSI, NSAPI, current cell location, etc.) is loaded when the SGSN is started and is used to set up the initial state of the system. Calls are made to the GMM subsystem to initialize the MM and PDP contexts of each mobile loaded in the system. In a commercial implementation, a mobile uses signaling messages to connect to the GPRS network. A mobile who wishes to connect to an external network using GPRS must first initiate the Attach procedure and then the PDP Activation procedure[8].

Packets coming from the Gn interface are mapped to the mobile they belong to with their Tunnel IDentification (TID)[11]. The TID is part of the GTP header and is a combination of the mobile's IMSI and a NSAPI. Packets coming from the Gb interface are identified by the mobile's TLLI. The state of a mobile must be retrievable using its TLLI or the IMSI. The GMM data structures are designed to retrieve the state of a mobile using either a TLLI or IMSI in O(1). The GMM module is implemented as a fixed size table initialized on startup. The maximum

number of mobiles supported by the system is decided at compilation time.

5.1.2 SNDCP and LLC implementation

The testbed implementation of the SNDCP layer is based on [13]. It supports V.42bis data compression[18] and TCP/IP header compression[19]. Packet fragmentation and reassembly is supported. The SNDCP layer keeps some state for each mobile/NSAPI pair. The TCP/IP header compression state, the V.42bis dictionary and the SNDCP sequence number and fragment status, are kept at the SNDCP level. This state is accessed using a mobile's TLLI and NSAPI in O(1). The SNDCP implementation only supports the unacknowledged transmission mode.

The implementation of the LLC layer is based on the requirements described in [12]. The implementation performs the frame check sequence (FCS). The FCS is a 24 bits cyclic redundancy check (CRC) code. It follows the guidelines described in [12]. The encryption implementation is different than what is specified in the requirements. We have implemented a simple form of encryption to impose on each packet traversing the stack, a penalty equivalent to the real encryption mechanism in terms of processing time and memory usage. A typical implementation will have to touch every byte in a packet in order to encrypt it. This requires transferring each byte from memory to the processor and back to memory. Our encryption mechanism imposes a similar burden on the system. The LLC layer also keeps some information for each mobile. This state is indexable in O(1) with the mobile's TLLI. The LLC frame sequence number is stored in this table.

The implementation of the LLC layer only supports the unacknowledged mode of operation. There are no error recovery or reordering mechanisms. Corrupt frames will be detected with the FCS, duplicate frames will also be detected and

discarded using the frame sequence number. In the case of missing frames, the error will be detected but not recovered. Our implementation supports both protected and unprotected modes. In protected mode, the FCS is calculated over the entire frame. In unprotected mode, the FCS is only calculated over the LLC frame header and the first bytes of the payload. This number of bytes corresponds to the size of the SNDCP header.

5.1.3 Flow Control

The testbed SGSN supports flow control on the Gb link as specified in [9]. Data is flow controlled on a per mobile basis and on a per cell basis. The requirement specifies that the base station use a leaky bucket to control the rate of traffic for each mobile and each cell. The SGSN is not allowed to send packets to the base station that would overflow one of the leaky buckets. If a bucket at the base station is full, the SGSN will delay packets for a short period of time. The bucket capacity at the base station and the buffering capacity on the SGSN change depending on the experiments running. Usually the capacity is set to buffer one second of traffic at each location (base station and SGSN) for both cells and mobiles.

We use the service queue explained in Section 4.2 to implement flow control. Before a packet is sent over the Gb link it is first sent to the mobile flow control unit. The mobile flow control unit identifies the owner of the packet from its TLLI and takes one of the following actions depending on the state of the connection:

- If there is room at the base station for this packet, then the packet is sent immediately.

- If there is currently no room at the base station for this packet, we need to control this mobile's flow. The packet's transmission needs to be delayed until

there is room at the base station. The packet is buffered by the mobile's flow control module and the module is added to the service queue.

- If the mobile's queue on the SGSN is full, the packet is dropped.

When a mobile's flow is controlled, the transmission of buffered packets is done at a later time when the service queue is processed. When this occurs, the flow control module determines how much space is now available at the base station and sends as many packets as allowed. If there are still packets to send then the module is put on the service queue again.

Once a packet has gone through the mobile flow control unit, it goes down to the cell's scheduler and flow control unit. The scheduler works with the cell flow control unit. If the destination cell is not flow controlled, then packets simply bypass the scheduler and are sent right away. If the cell needs to be flow controlled then packets are buffered by the scheduler and the scheduler is put on the service queue. When the service queue is processed, the scheduler decides which packets should be sent to the base station. The room available at the base station determines how many packets should be sent. If there are still packets to to send, the scheduler is put back on the service queue.

In the basic configuration, the scheduler is simply a First In - First Out (FIFO) queue. Packets are queued on a first come, first served basis, independently of their QoS profile.

We have configured the scheduler to support real-time traffic by using a priority queue. The priority queue currently supports only two classes of traffic: real-time and best effort. The priority queue is implemented as two FIFO queues, one for each traffic class. The reader of the priority queue always reads the real-

time queue first and only removes packets from the best effort class if there are no real-time packets available. This ensures strict priority for real-time traffic.

A priority queue is one of the possible mechanisms discussed in [20] to support the Expedited Forwarding (EF) service. EF is part of the Differentiated Services architecture[28][4] and is designed to bring support for a guaranteed low-loss low-delay low-jitter service on the Internet. A real-time traffic class has very similar requirements to those provided by the EF service. In [20], the authors mandate the usage of a rate limiter, such as a token bucket, to prevent starvation of other traffic classes when a mechanism such as a priority queue is used.

We currently do not support a rate limiter for the real-time class at the cell level. The only rate limiter we currently support is the mobile flow control mechanism. This will limit the rate of each mobile but will not enforce a limitation on the aggregated flow of all mobiles in a given cell. This means that a mobile cannot exceed its pre-allocated rate even if it belongs to the real-time class. The problem is that multiple real-time mobiles could starve other non-real time mobiles in the same cell since the maximum rate for the real-time class is not defined at the cell level. Since our system is only experimental, this is not really a problem. We always control the number of mobiles in each cell and we always ensure that our real-time traffic is well behaved.

Support for different precedence levels can also be configured. Two precedence levels are currently supported: normal and high precedence. Normal precedence packets are always dropped before high precedence packets. In a case where a high precedence packet comes in and the queue is full, the high precedence packet will replace a normal precedence packet and the latter will be dropped. High precedence packets can still be dropped when their owners exceed their allocated bandwidth.

5.1.4 BSSGP and NS implementation

Because of the hardware difference on the Gb link, our BSSGP and NS layer implementations are quite different. We use only a very simple BSSGP header. It contains the mobile's TLLI, QoS profile and the size of the packet. This information is also contained in a real BSSGP header with other information relevant only in a real system. In a real implementation, frame relay time slots are assigned to create a BSSGP Virtual Connection (BVC) between a cell and the SGSN. There is one BVC per cell. In our case, BVCs are mapped over UDP connections. We have one UDP connection per BVC. Although UDP is connectionless, it is possible to bind a UDP socket to a source port and to connect it to a remote port, therefore creating an unreliable connection.

The NS layer simply acts as an abstraction layer for sockets. Taking care of tasks such as binding the socket to an IP address and connecting it to a remote IP. The NS layer also does scatter-gather I/O, avoiding making an extra copy of the packet to group all headers, payload and trailers into a single buffer.

5.2 MS/GB and GGSN/GN

The MS/GB simulates the Gb side of the GPRS network. It simulates mobile stations and cells. Except for the lowest layers of the protocol stack, the packets sent by the MS/GB are identical to those a real SGSN would receive from actual mobiles.

The GGSN/GN simulates the Gn side of the GPRS network. It simulates a GGSN and the end nodes connected to it. An end node is the other half of a communication with a mobile. An end node sends and receives data to/from a

mobile. The packets generated are identical to those a real SGSN would receive from a GGSN via the Gn interface.

The MS/GB and the GGSN/GN were both built using the framework developed for the SGSN. They also use the parts of the SGSN stack that are relevant to them (The GGSN/GN uses the Gn side of the SGSN stack and the MS/GB uses the Gb side of the stack). On top of the stack sits the packet generator. The role of the packet generator is to send and receive packets and keep statistics on them.

Depending on how the testbed is configured, generated traffic can flow in the downlink, uplink or in both directions simultaneously. If a node is configured to send data, then a packet generating function runs every 10 milliseconds to generate the desired traffic load.

The same configuration file used by the SGSN to load its state is used by the packet generator to configure itself and its underlying GPRS stack. The file in addition to the GPRS related configuration information, also contains the packet size and packet rate for each user. A user has both a mobile and an end node associated with it, and depending on the traffic direction configured for the testbed, the mobile, the end node, or both generates traffic at the rate specified in the configuration file.

When the packet generating function runs, it builds and sends a burst of packets for each user. The size of the burst depends on the data rate assigned to the user and the distribution used. Currently, uniform and exponential distributions are supported. When the exponential distribution is used, each user possesses its own distribution which is initialized with a different seed for each user. To avoid traffic patterns caused by the order in which each user's traffic is sent, the sending order is randomly selected.

Chapter 6

Early Demultiplexing

As mentioned previously, we believe an SGSN implemented as a user process will experience problems related to Quality of Service(QoS). These problems are caused by the usage of tunnels to transfer data between GSNs combined with the FIFO behavior of the input queue. First, we will describe how early demultiplexing can solve the problems described in chapter 2. A overview of the Early Demulitplexing SGSN(ED-SGSN) will be given and then the design of the demultiplexing kernel module will be described in detail.

6.1 Overview

To avoid the problems described in chapter 2, the incoming aggregated traffic on the Gn interface must be demultiplexed as early as possible. The PDP context and QoS profile of each packet must be retrieved in interrupt context so that packets can be classified and queued accordingly, as they arrive. Once packets are queued properly, the order in which they are processed through the stack can be easily scheduled. The selection of the next packet should be based on the bandwidth on

the QoS profile and on the status of the Gb link. To avoid processing packets that will be dropped because their output queue is full, a feedback mechanism is required to inform the packet scheduler of the current status of the output queues. Packets whose target queues are full are not scheduled to be processed. No resources are wasted on packets that can be processed through the SGSN but dropped by the flow control mechanism.

By combining early demultiplexing with a feedback mechanism, a SGSN can discard exceeding traffic early with a minimum amount of resources wasted. The number of packets processed by the SGSN remains fairly constant with or without the usage of early demultiplexing, but because no exceeding traffic is processed through its stack, the ED-SGSN throughput remains fairly constant when faced with large amounts of exceeding traffic, even when its processing resources become saturated.

The packet scheduler allows the fair distribution of CPU resources among all users according to their QoS profiles. This way, proper isolation is provided between users, and exceeding traffic from misbehaving users does not affect the throughput of well behaved users.

An early demultiplexing mechanism is only implemented for downlink traffic. There are no requirements in the GPRS technical specification for flow control and packet scheduling on the SGSN for uplink traffic. This means that packets flowing in the uplink direction are basically processed and sent out on a First-In First-Out(FIFO) fashion. This is acceptable since the bottleneck is most likely going to be in the downlink direction because of the bandwitdth difference between the Gb and Gn interface.

Depending on how incoming packets are queued on the Gb interface by the

Frame Relay device driver, early demultiplexing could also be very important on the Gb interface to ensure proper prioritization and precedence of traffic. It can also be used to ensure proper resource allocation within the SGSN between uplink and downlink traffic. Because we believe the amount of traffic will be greater in the downlink direction than in the uplink direction, we have focused on the Gn interface. An early demultiplexing mechanism on the Gb interface is left for future work.

To support real-time traffic, an SGSN must be able to offer support for a low delay low jitter traffic class. Traffic belonging to this class should experience a small and constant transit delay at the SGSN. In a user level SGSN, incoming packets on the Gn interface are queued by the operating system in a UDP socket buffer. This buffer acts as a FIFO queue and doesn't take into account the priority of each packet. This means that the delay experienced at the SGSN will be dependant on the input queue length, which is dependent on the load on the system. Because the load on an SGSN is never really constant, the delay will vary, causing considerable jitter. A user level SGSN can minimize the delay at the output queue by using an appropriate scheduling mechanism, but is unable to reduce the delay at the input queue.

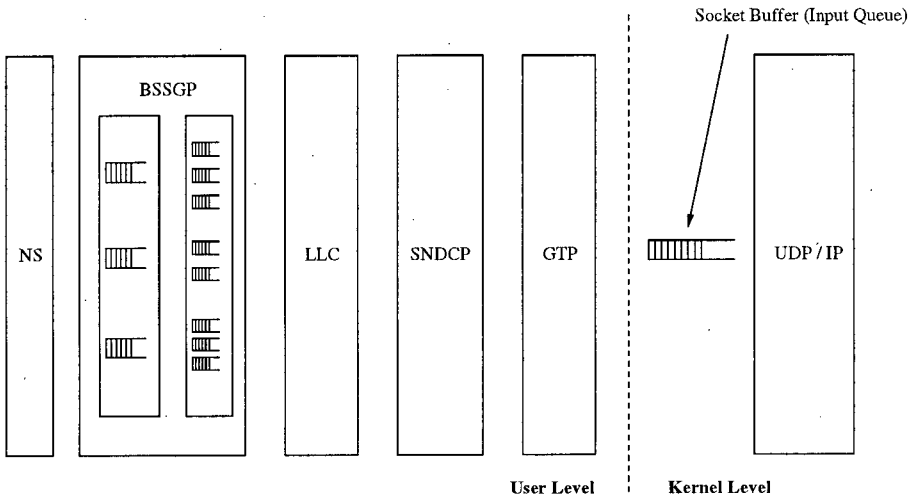
By demutliplexing the traffic before it is queued, it is possible to identify and process high priority packets ahead of lower priority traffic. This is necessary to provide support for real-time traffic. This way, high priority packets are prioritized both at the input and output queues. This provides a transit delay independent of the load on the system, since high priority packets are never queued behind other classes of traffic. Because all incoming traffic is identified on arrival, it is possible to determine the precedence class of a packet when it arrives. This allows the system to take the appropriate action, instead of simply behaving in a tail drop fashion.

6.2 Early Demulitplexing SGSN

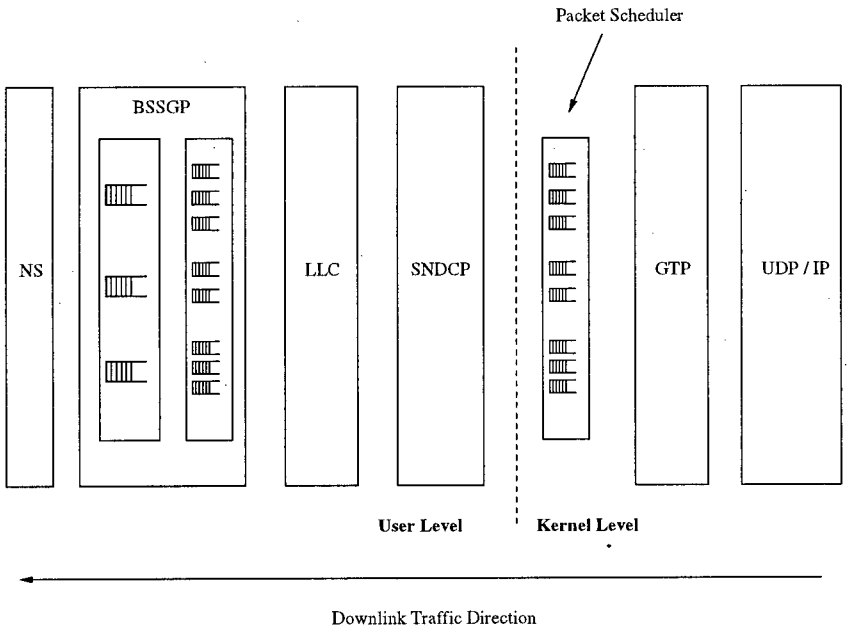
We have built an SGSN where the UDP input queue of a user-level SGSN is replaced by a demulitplexing mechanism which classifies and schedules incoming traffic on the Gn interface. Packets are identified on arrival in interrupt context and queued on a per mobile basis. Figure 6.1 shows a user level SGSN and our early demultiplexing implementation called ED-SGSN. Although the internal software architecture is different, the interfaces of the ED-SGSN have not changed.

To be able to identify incoming packets on the Gn interface, the GTP header needs to be read to retrieve the user's MM and PDP contexts. Because we want to identify packets before they are queued, this must be done in interrupt context. Since all work done in interrupt context is done in kernel mode, the processing of the GTP header and the retrieval of the MM and PDP contexts for incoming packets is now done in kernel mode, after the IP and UDP header processing. Once identified, packets are passed to the packet scheduler.

The packet scheduler operates on a per mobiles basis. The scheduling should actually be done on a per PDP context basis, since QoS profiles are associated with PDP contexts and not with individual mobiles. Since our implementation only supports one PDP context per mobile, scheduling mobiles or PDP contexts is exactly the same. The scheduler has one queue for each mobile registered on the SGSN. Incoming packets, once identified, are queued in interrupt context in the appropriate queue. The actual scheduling occurs in user context when the user level portion of the SGSN makes a system call to read a packet. The scheduler picks the next packet to be processed based on its scheduling policy and the status of the output queues. If the output queue of a mobile or a cell is full, the corresponding input queue will be blocked by the feedback mechanism. Blocked input queues are skipped by the



(a)



(b)

Figure 6.1: SGSN(a) and ED-SGSN(b)

scheduler. The standard scheduling policy is round robin. This policy is appropriate for cases where all users have been assigned the same throughput and have the same QoS profile.

Although parts of the GTP layer have been moved to kernel space, most of the SGSN software remains unchanged. The other main differences are that once read, incoming packets on the Gn interface are passed to the SNDCP layer instead of the GTP layer. Support for the feedback mechanism has been added to the flow control unit. System calls were added to notify the SGSN kernel modules that certain mobiles should be blocked or unblocked. The transmission of packets on the Gn interface is unchanged and still uses a UDP socket.

The ED-SGSN can be configured to support real-time traffic. It uses the same priority queue mechanism in the flow control unit to schedule the transmission of packets on the Gb interface. This prioritization mechanism is there to ensure that high-priority packets are not delayed at their output queue. More details on the usage of a priority queue are available in section 5.1.3. The kernel-level packet scheduler also uses a prioritization mechanism. Mobiles in each traffic class are served by a separate round robin scheduler. Again, two traffic classes are supported: real-time and best effort. Priority is always given to the real-time scheduler, ensuring strict prioritization of real-time traffic. Again individual flows are controlled but the aggregated real-time flow is not. As we have stated previously, this is significant in our case but would be in a commercial system. The prioritization mechanism in the kernel level packet scheduler is there to avoid delay at the input queue. This delay is most likely going to be more important than the output queue delay because the input queue is used by every user attached to the SGSN. The packet scheduler in the flow control mechanism only schedules packets among users located in the same cell.

The number of users in a given cell will always be smaller than the total number of users attached to the SGSN, so the potential for long delay is alway greater at the input than at the output.

The ED-SGSN can also support the two-level precedence mechanism described in section 5.1.3. A similar mechanism is configured inside the kernel-level packet scheduler to support the different precedence levels. The packet scheduler always accept high precedence packets unless their individual queue is blocked because the owner has exceeded his allocated bandwidth. In the case where a high precedence packet will replace a low precedence packet, the low precedence packet to be dropped is taken from a user located in the same cell as the owner of the high precedence packet.

The packet scheduler also needs to increase the priority of high precedence users from time to time. The scheduler increases the priority of a high precedence user when its queue is above a high watermark. In this case, unless blocked, the queue is processed ahead of the others by the packet scheduler. This is necessary when the ED-SGSN is overloaded to make sure queues belonging to high precedence users do not grow indefinitely. When the ED-SGSN is overloaded, the fraction of processing resources a high precedence user may receive from the scheduler might not be enough to sustain the user's incoming traffic rate. In this case, since high precedence packets are always accepted, the high precedence user's queue will grow unbounded if its priority is not raised.

6.3 KSGSN

The early de-multiplexing module named KSGSN was implemented in kernel space using the *Netfilter* architecture[2]. It is built as a group of kernel modules that run

both in interrupt and user context. It acts as a packet scheduler and replaces the UDP input queue on the Gn interface. It supports system calls for configuration and data transfer. The ED-SGSN uses KSGSN instead of a regular UDP socket for packet reception on the Gn interface. Packet transmission on the Gn side is unchanged and still uses a regular UDP socket. The *Netfilter* architecture is new in Linux 2.4. It is meant to replace the firewall and address translation implementation. These two used to be implemented separately even if they had much functionality in common. They are now unified in *Netfilter*. *Netfilter* provides hooks at different point in the network stack. It currently provides five hooks in the IPv4 stack. On top of *Netfilter* sits *Iptables* which provides the capability to attach kernel modules to any of the *Netfilter* hooks. These kernel modules can observe, modify or even grab packets away from the network stack and process them in any desired way. *Iptables* allows to specify matching rules for a kernel module. If a packet matches the rule, it is forwarded to the module.

KSGSN is implemented as a group of kernel modules. It can be dynamically loaded and unloaded from the Linux kernel at runtime. It is composed of four main components each implemented as a separate kernel module. Figure 6.2 shows the architecture of KSGSN and the links with the user space SGSN. Incoming GTP packets are intercepted by *Netfiler/Iptables* and passed to KSGSN.

The main component is the packet scheduler named *ksched*. Its task is to queue incoming packets and to decide which packet to pass to the SGSN when requested. The scheduler has one FIFO queue per mobile, which buffers incoming packets. Packets are inserted at the back of a mobile's queue in interrupt context and are removed from the front of the queue in user context. The selection of the next packet to be removed from its queue and passed to user space is based on the

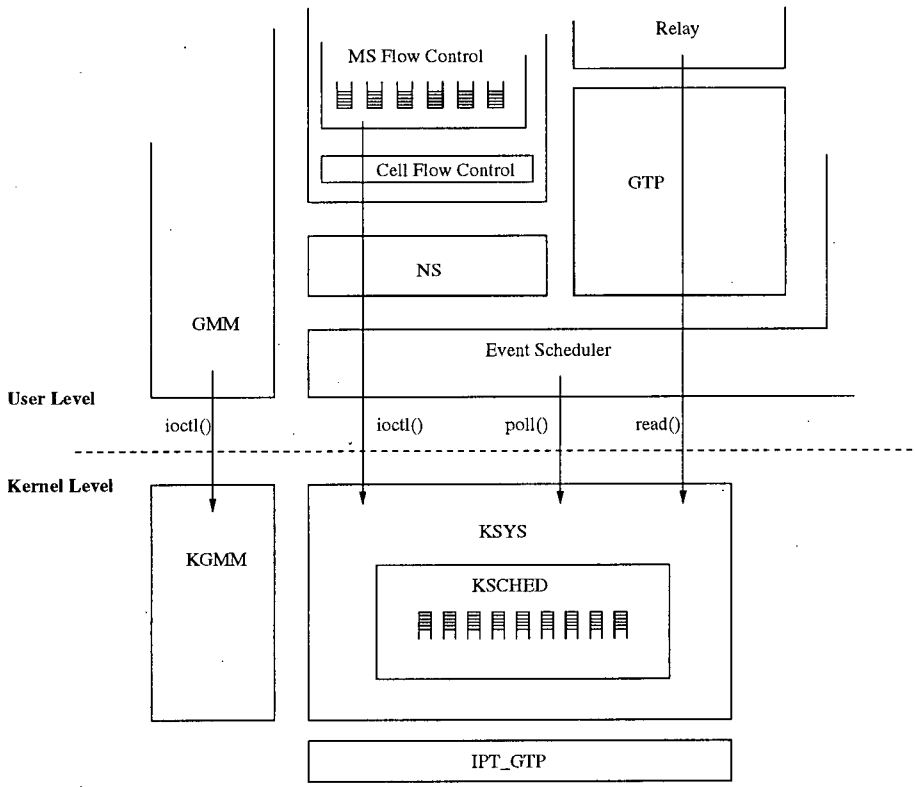


Figure 6.2: KSGSN software architecture

scheduling policy.

The *kgmm* module is the kernel level part of the user level *gmm*. It keeps the state necessary to KSGSN for each mobile. The *kgmm* is configured by the gmm with *ioctl()* system calls, passing down information for each mobile. The MM and PDP context are retrieved from the *kgmm* module in interrupt context for each incoming packets using the Tunnel ID(TID) from the GTP header.

The *ksys* module serves as a wrapper around the scheduler. Its task is to translate all the system calls into calls to the scheduler and process incoming packet before passing them to the scheduler. The *ksys* module supports *ioctl()* system calls for configuration and to block/unblock a mobile's queue in the scheduler. The *read()* system call is used to transfer packets from kernel space to user space and the *poll()* system call is implemented allowing the SGSN to sleep when there are no packets to read.

The *ipt_GTP* module is the kernel module registered with *Iptables* to receive all GTP packets. It performs some basic error checking and passes the packet to the *ksys* module. Since our modules bypass the UDP layer, we have to make sure packets are not corrupted. This task is usually performed at the UDP level by verifying the checksum in the UDP header. The Ethernet card used in our testbed performs UDP checksum in hardware allowing us to bypass this step.

Chapter 7

Experimentation and Results

7.1 Overview

As stated earlier in Chapter 2, we believe a user level SGSN will not be able to truly support Quality of Service (QoS). By truly, we mean that it will not be able to provide proper isolation between the different classes of traffic and provide guaranteed services independently of the load on the system. We have studied this problem by looking at three QoS parameters: throughput, delay and precedence. These parameters are part of a GPRS QoS profile and are defined in [14]. In this section, we show that a user level SGSN fails to provide guaranteed service with respect to these QoS parameters under heavy loads. We will also show that our early-demultiplexing SGSN (ED-SGSN) is able to maintain guaranteed service under heavy loads and degrades gracefully when unable to meet all guarantees.

7.1.1 Lab Setup

The GPRS testbed setup used to run all experiments is shown in Figure 7.1. All nodes are connected using switched Ethernet running in duplex mode. The SGSN

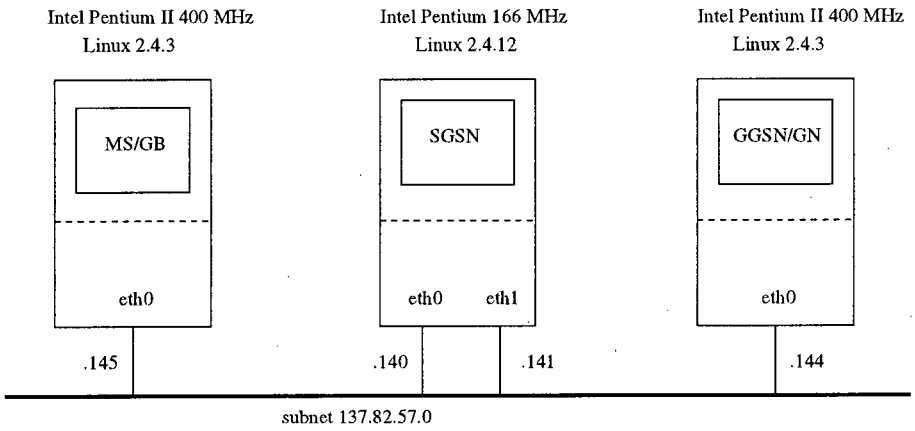


Figure 7.1: Lab setup

runs on a Pentium 166Mhz and both GGSN/GN and MS/GB run on Pentium II 400 MHz. The Ethernet cards used are 3Com 905TX PCI. They are able to perform scatter-gather DMA and TCP/UDP checksum in hardware.

The SGSN uses two Ethernet cards. One for each interface (Gn and Gb). This is done to make sure the hardware does not become congested when faced with high transmission rates. To make sure packets are sent to the right network interface on the SGSN, ARP entries are added to the GGSN/GN and MS/GB machine. On the SGSN, to force packets to go out on the desired network interface, entries are added to the routing table.

Each PC runs RedHat Linux. The SGSN uses the Linux kernel 2.4.12 and the other nodes use kernel 2.4.3. The GPRS software is written in C and is compiled with the GNU C compiler. Because it uses some of the GNU extensions to the C language, the GNU C compiler must be used. Since it uses POSIX APIs, the user-level code should be easily portable to other UNIX platforms that support the GNU C compiler.

7.1.2 Traffic Generator

All traffic is generated using the packet generator. The packet generator builds and sends bursts of packets every 10 milliseconds. To avoid timing drifts, the frequency of the generator needs to be a multiple of the system clock resolution. The system clock frequency on most PCs is 100 Hz[16]. The higher the frequency, the more accurate the packet generator will be, but setting the frequency at a higher rate will lead to serious timing drifts and inaccuracies. The frequency of the generator was set to 100Hz for all experiments. Every time the generator runs, a burst is built for each user and is sent out. The size of the burst depends on the pre-configured rate of each user and on the type of traffic distribution configured for each user. The packet generator supports two types of traffic distribution: exponential and fixed.

All experiments are done with unidirectional traffic. All traffic is generated by the GGSN/GN and travels in the GGSN/GN to MS/GB direction. Except for two-way applications, such as video conferencing and voice over IP, most applications on the Internet are largely unidirectional. Web surfing and multimedia streaming have far more data travelling from the server to the client than in the other direction. We assume that the usage of GPRS will be similar. Most of the traffic will be in the downlink direction, travelling from the Internet to the mobiles.

To simplify our experiments, all generated packets are identical. They are the same size and the random content of the payload is hard coded. This allows the destination host to know the content of incoming packets and to easily detect errors. Generating the payload at runtime would slow down the packet generator. We do not believe that using the same content in each packet has any influence on the SGSN performance and on the outcome of the experiments.

V.42bis data compression was turned off for all experiments. The perfor-

mance of the compression algorithm had too much impact on the overall performance of the SGSN. This is acceptable since V.42bis is an optional feature in GPRS. The payload carried in each packet is 378 bytes. With the SNDCP, LLC and BSSGP header, the total size of the packet being sent over the Gb interface becomes 400 bytes.

To model regular traffic, we used a Poisson distribution. This allows us to generate traffic with a certain burstiness since the interval between packets is exponentially distributed. Because of the limitation of the system clock resolution, generated traffic cannot be truely exponential. It is impossible to send each packet at the exact instant dictated by the distribution. Instead, packets are grouped and sent out as a continuous burst. Each time the packet generator timer expires (every 10 milliseconds), the packet generator calculates the size of the next burst to send for each user. The size of the burst is determined by adding all packets that would have been sent in the next 10 milliseconds according to the exponential distribution. Once the size of the burst is determined, the packets are sent out one after the other. The rate at which each packet in a burst is actually sent out is determined by the speed of the hardware and operating system. This operation is repeated for each user. Once every user has sent his data, the generator sleeps until the next interval.

The packet generator also supports a uniform distribution. This allows traffic to be generated at a fixed rate. The uniform distribution is used to model real-time traffic, such as streaming multimedia.

For both distributions, the order in which each user's packet burst is generated and sent out is randomly determined at each interval. This is to avoid potential traffic patterns that could be caused by using the same order over and over again.

7.2 KSGSN Overhead

Modifying an SGSN stack by moving some of it to kernel space can affect the stack performance. We have to make sure this operation does not add too much overhead. The time spent by packets traversing the stack cannot increase significantly or the overall performance of the system will suffer.

We have measured the stack traversal using *Netfilter*. We have written a kernel module that captures incoming and outgoing GPRS packets. This module measures the traversal time inside the SGSN for each packet belonging to a specific mobile. The module is configured with the mobile's IMSI and TLLI. Every time a packet with the target IMSI comes by on the Gn interface, the module takes a time stamp. When a packet with the corresponding TLLI leaves on the Gb interface, the module takes another timestamp. The incoming timestamp is taken in interrupt context just before being sent to the KSGSN's packet scheduler. The outgoing timestamp is taken in process context just after the UDP protocol layer. Separate counters are kept for incoming and outgoing packets allowing measurements to take place even when there is more than one packet in the system at a given time. The timing is done using the CPU cycle counter. This counter is incremented at each CPU cycle, allowing very precise measurements.

We ran each test three times for both the SGSN and ED-SGSN. Each test is made with no load on the system. Only one user is registered, sending at a rate of ten packets per second. The uniform distribution was used to make sure the flow control would not intervene and delay packets, distorting our results. Each test ran for 1000 seconds. In total, over 10000 measurements are taken in each run, but the module only keeps the last 2048 measurements. Early measurements have to be discarded anyway since the system is starting up and might not be in a steady

state.

	SGSN avg	SGSN std dev	ED-SGSN avg	ED-SGSN std dev
test 1	413.26	2.56	407.55	0.29
test 2	406.33	0.27	412.05	0.69
test 3	409.64	8.65	408.42	1.69
avg	409.74	3.83	409.34	0.89

Table 7.1: Stack traversal timing (microseconds)

Table 7.1 shows the average and standard deviation of the three tests for both SGSN and ED-SGSN. All results are in microseconds. As the table shows, the differences between the two are minimal. This is understandable since no additional functionality is added, but simply moved from user space to kernel space. The standard deviation is low for all tests, meaning that the measurements are fairly constant.

To make sure comparative tests were fair, both SGSNs were configured with the same buffering capacity. In the ED-SGSN, each mobile possesses its own input queue. The capacity of this queue is set to buffer one second of traffic at the mobile's allocated rate. The size of the input queue on the SGSN was set to be equal to the total number of mobiles times the capacity of an individual mobile queue on the ED-SGSN.

7.3 Througphut

Our first experiment was to measure the throughput of the system and see how the SGSN behaved when submitted with variable amounts of exceeding traffic. We look at two things, the overall throughput and individual user throughput. Our results show that exceeding traffic affects the overall throughput, of both the SGSN and

ED-SGSN but that the ED-SGSN always performs better, up to 42 % in certain cases.

Our results also show that the SGSN fails to maintain isolation between individual users' traffic when processing resources become saturated. Well-behaved users are penalized by misbehaving users. The ED-SGSN, because it allocates processing resources based on users' pre-allocated bandwidth, always maintains isolation between users and is not affected by exceeding traffic. Once processing resources are saturated, users within the same QoS profile are penalized equally.

7.3.1 Overall Throughput

We first look at the overall throughput of the system. Although not related to Quality of Service per say, the behavior of the overall throughput of the system when overloaded can give a good idea of how guaranteed services are going to be affected under such conditions. The overall throughput of the system is defined as the total number of packets per second that the SGSN is able to output on the Gb interface.

We define the theoretical throughput as the throughput of an SGSN with infinite processing and buffering capacity. Even with infinite processing capacity, an SGSN would still drop packets belonging to mobiles which exceed their allocated bandwidth, but wouldn't drop packets because of buffer overflow in their input queue. When an SGSN is not saturated, the actual throughput will be equal or very close to the theoretical throughput. Once saturated, an SGSN will not be able to keep up and its throughput will be lower than the theoretical throughput. The theoretical throughput allows us to see how efficent an SGSN actually is.

The input load coming from the Gn interface on an SGSN can be split into

two parts: the well-behaved traffic generated by all users and the exceeding traffic generated only by misbehaving users. The theoretical throughput is always equal to the amount of well-behaved traffic.

We have measured the throughput of both SGSN and ED-SGSN with three different traffic models. In all three traffic models, every user has the same QoS profile (i.e. the same pre-allocated bandwidth) and the number of mobiles per cell is equal for every cell. The only difference between each model is the fraction of exceeding traffic. For a given number of users, the theoretical throughput of each model is identical, only the quantity of exceeding traffic varies. The fractions of exceeding traffic for each model are respectively 0, 33.3 and 50%. These fractions represent the ratio of the exceeding traffic over the total amount of incoming traffic (well behaved and exceeding). In a case where the theoretical throughput is 2000 packets per second and the fraction of exceeding traffic is 33.3% for example, the total input on the system will be 3000 packets per second. In every traffic model, the exceeding traffic is always generated by half of the user population.

Figures 7.2, 7.3 and 7.4 show the throughput for the SGSN and ED-SGSN under various loads for each traffic model. On each graph, we can clearly see the saturation point for the SGSN and ED-SGSN. The saturation point represents the load at which an SGSN is no longer able to process all incoming traffic. It's the point where the actual throughput is no longer equal to the theoretical throughput. The graphs show that the ED-SGSN saturates between 1900 and 2150 packets per second depending on the additional load of exceeding traffic generated by misbehaving users. On the other hand, the SGSN saturates between 960 and 1600 packets per second depending on the amount of exceeding traffic. Although our results show that the ED-SGSN performs better, even when not subjected to exceeding load from

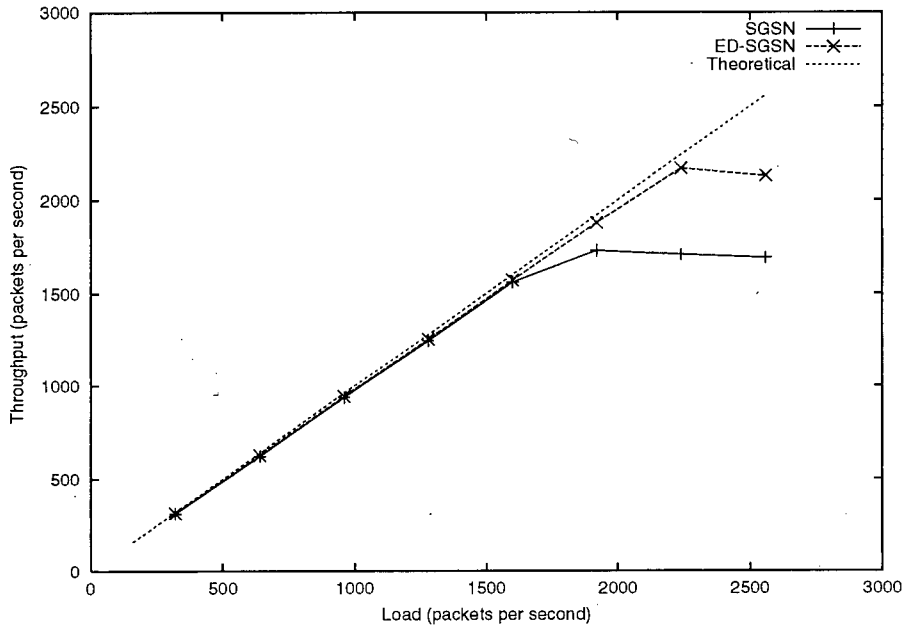


Figure 7.2: Throughput of SGSN and ED-SGSN with no exceeding traffic

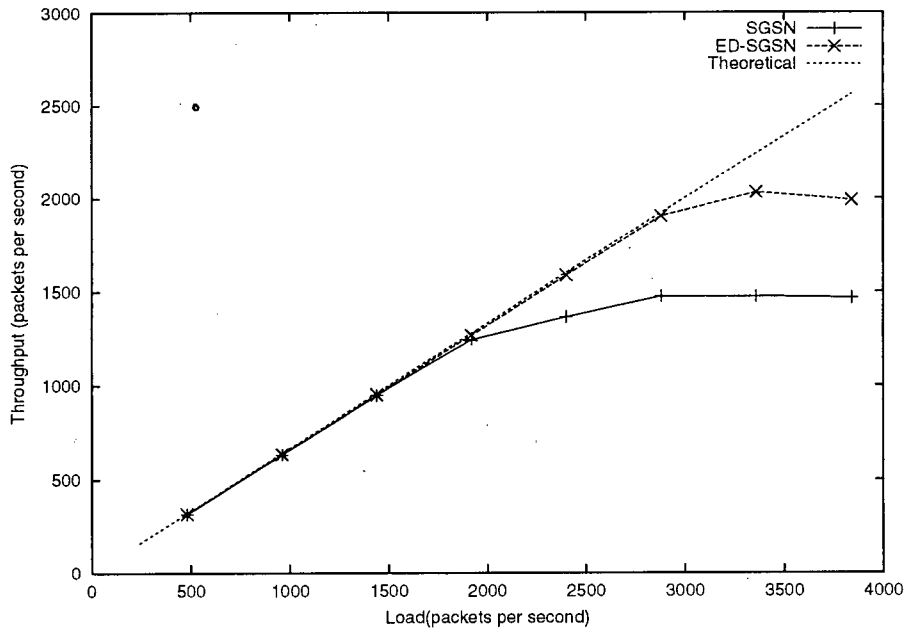


Figure 7.3: Throughput of SGSN and ED-SGSN with 33.3% exceeding traffic

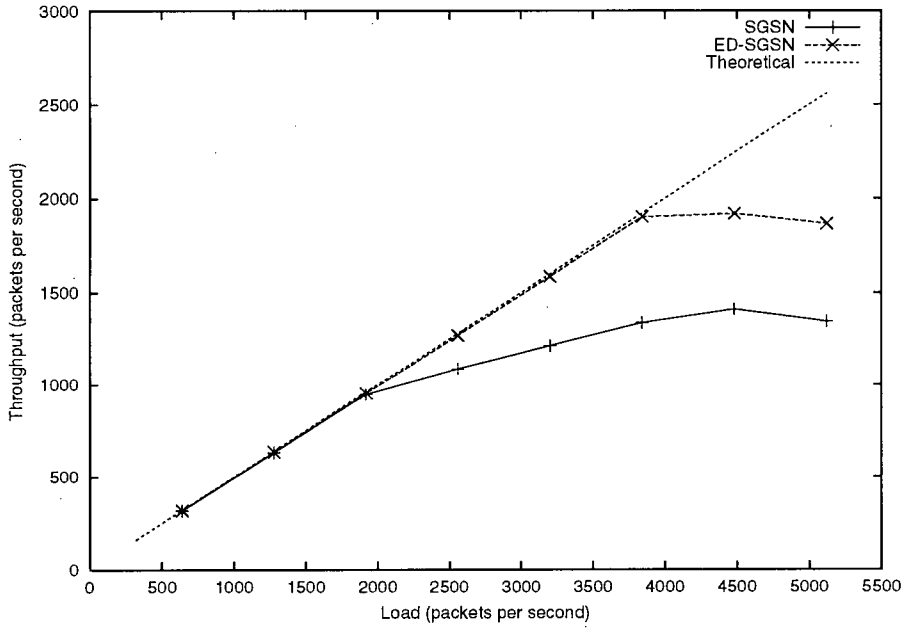


Figure 7.4: Throughput of SGSN and ED-SGSN with 50% exceeding traffic

misbehaving users, the important thing here is the fluctuation of the saturation point experienced by both the SGSN and ED-SGSN. When a traffic model with a 50% fraction of exceeding traffic is submitted, the SGSN's saturation point diminishes by 40%. In the same situation, the saturation point of the ED-SGSN only diminishes by 12%.

The large reduction in throughput experienced by the SGSN can be explained by the fact that the SGSN processes packets on a first-come first-served basis. This is done independently of the status of the output queues. When a packet's output queue is full, the packet is dropped, but only after it has been processed through most of the SGSN protocol stack. At this point, a substantial amount of resources have been spent on this packet and, therefore wasted. When the SGSN is saturated, this waste of resources leads to a diminution in throughput. Wasted resources could

have been used on packets that would have been successfully sent out on the Gb interface. Instead, some of theses packets end up being dropped at the input queue because of lack of processing resources.

The ED-SGSN doesn't suffer this fate because it takes into account the status of the output queues when scheduling which packet should be processed through the protocol stack. When a given output queue is full, no packets destined for this queue are read and processed through the stack. This ensures that no resources are wasted on processing packets that will be dropped at their output queue.

The difference in performance between the ED-SGSN and the regular SGSN when there is no exceeding traffic, can be attributed to better software efficiency and reduced overhead. Since the ED-SGSN uses less flow control, the service queue is not used as heavily, allowing more resources to be dedicated to the processing of packets.

The reduction in throughput experienced by the ED-SGSN past the saturation point is caused by the processing of incoming packets through the IP, UPD and GTP protocol layers. This processing is done in interrupt context before the ED-SGSN determines if a packet should be queued or dropped. This means that every packet dropped by the ED-SGSN still wastes a small amount of resources. This small wasteage of resources adds up and becomes noticeable when the ED-SGSN is heavily loaded and a large quantity of packets is dropped.

7.3.2 Fairness and Isolation

It is our belief that users with the same QoS profile (i.e. same throughput class) should receive the same amount of bandwidth. If either processing or bandwidth resources become scarce, then the system should degrade gracefully. The reduction

in available bandwidth should be felt equally by all users with the same Qos profile. Misbehaved users should not receive more bandwidth than well-behaved users under any circumstances.

Unfortunately, when the SGSN reaches its saturation point and resources are scarce, the SGSN does not degrade gracefully. The difference in received bandwidth between well-behaved and misbehaved users is flagrant. Because packets are queued and processed in FIFO order and because of the tail-drop behavior of the operating system, once the SGSN is saturated the resources allocated to each user are directly related to the fraction each user occupies in the input queue. This means that the more packets a user sends, the larger the portion of resources he will receive regardless of whether the user is exceeding his allocated bandwidth or not.

Of course, all packets in excess of their pre-allocated bandwidth are dropped, but still, they have consumed precious processing resources. Well behaved users see their throughput diminish substantially because they are unable to obtain the necessary resources to process their packets, while misbehaving users still manage to obtain their full bandwidth.

To avoid this, resources must be allocated fairly among users. The fraction of resources a user receives must depend on his allocated bandwidth and QoS profile, not on the fractions he occupies in the input queue. Because the ED-SGSN identifies the owner of each packet on arrival, it can queue packets on a per mobile basis. Packets can then easily be scheduled so that resources are allocated fairly between all mobiles. Since there is one queue per mobile, it's easy to discard traffic on an individual basis. Misbehaving traffic is discarded on arrival, wasting a minimum of system resources and well-behaved users are never penalized. Of course, when the system is saturated, packets are dropped, but the system degrades gracefully. All

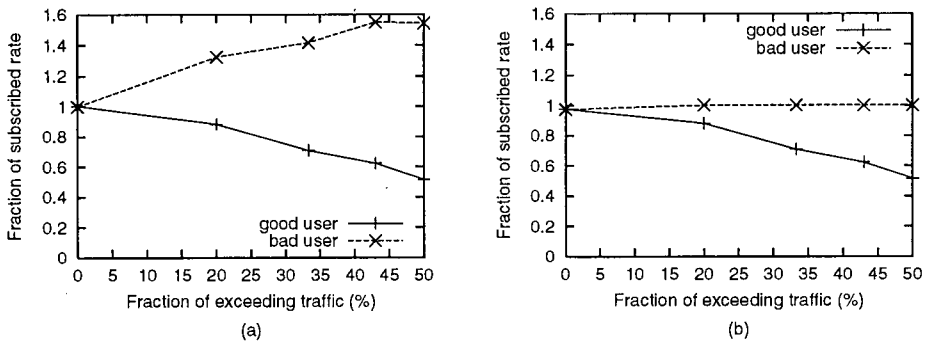


Figure 7.5: Input/Output of Good and Bad Users for the SGSN (160 Users)

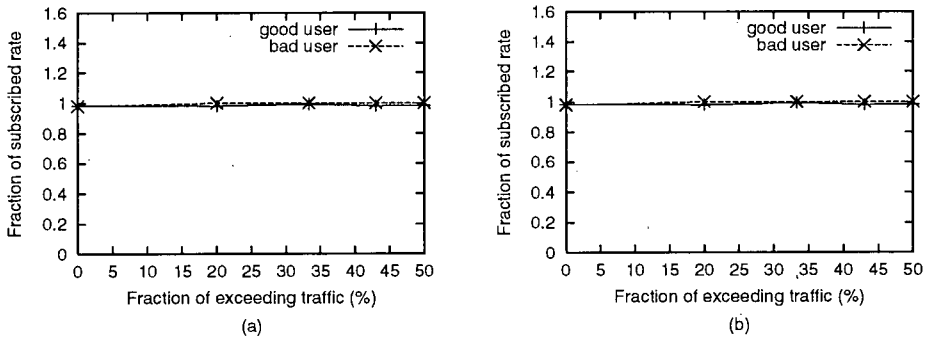


Figure 7.6: Input/Output of Good and Bad Users for the ED-SGSN (160 Users)

users with the same QoS profile are penalized equally.

To analyze isolation and fairness between users, we have compared the service received by two kinds of users: good and bad. A good user is a well-behaved user who does not generate exceeding traffic. A bad or misbehaving user generates a variable quantity of exceeding traffic. Both types of users have the same Quality of Service profile meaning they have the same allocated bandwidth. We have looked at how much resources each kind of user consumes and what fraction of their allocated

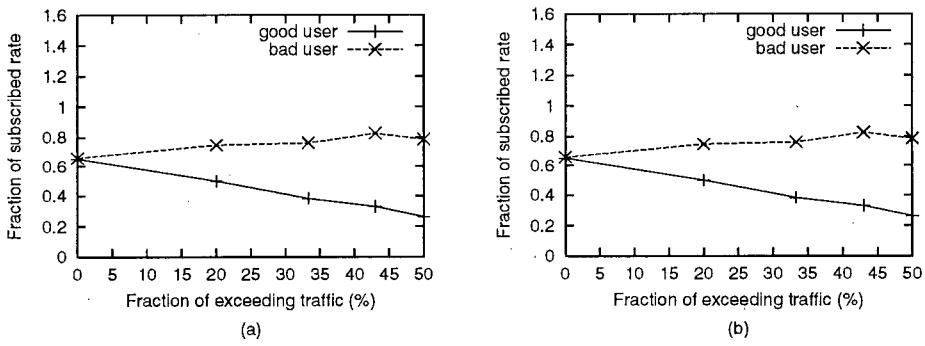


Figure 7.7: Input/Output of Good and Bad Users for the SGSN (256 Users)

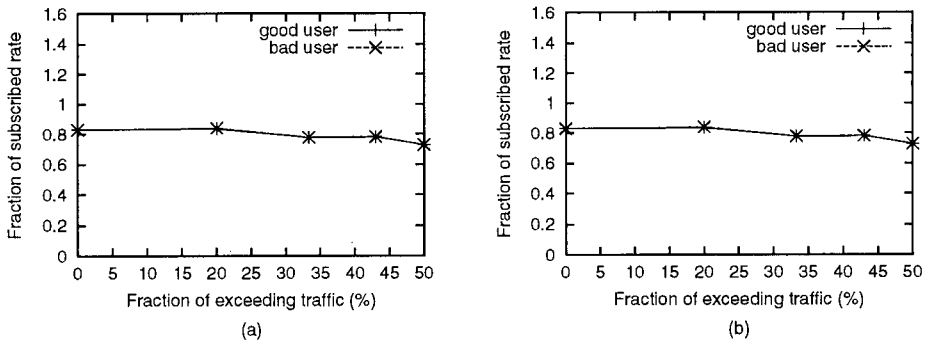


Figure 7.8: Input/Output of Good and Bad Users for the ED-SGSN (256 Users)

bandwidth they receive. The 160-user scenario corresponds, when no exceeding traffic is present, to a load slightly below the saturation point for the SGSN and ED-SGSN. When no exceeding traffic is present, both good and bad users receive their allocated bandwidth. The SGSN becomes saturated as the amount of exceeding traffic increases. The scenario with 256 users represents a load above the saturation point for both the SGSN and the ED-SGSN even when no exceeding traffic is present. In both scenarios, half of the user population consists of well-behaved users and the other consists of misbehaving users.

For each scenario and for each SGSN (SGSN and ED-SGSN) we vary the amount of exceeding traffic generated by misbehaving users. As in the overall throughput experiments, the amount of exceeding traffic is given as a fraction of the total amount of incoming traffic.

Figures 7.5 and 7.6 show the SGSN and ED-SGSN respectively, with 160 mobile users connected. Figures 7.5-a and 7.6-a show the amount of traffic read from the input queue by the SGSN or ED-SGSN. This quantity is expressed as a fraction a user's allocated bandwidth. The number of packets read and processed represents the amount of SGSN processing resources consumed by a user. Figures 7.5-b and 7.6-b show the amount of traffic transmitted on the Gb interface, again expressed as a fraction of a user's allocated bandwidth. Each graph shows a curve for good and bad users.

Figure 7.5-a clearly shows that as the load of misbehaving traffic increases on the SGSN, so does the quantity of resources consumed by misbehaving users. This is understandable since the SGSN processes packets in FIFO order and doesn't take into account whether packets will be discarded or not before processing them. Since misbehaving users' traffic occupies a larger portion of the input queue than

the traffic of well-behaved users, misbehaving users receive more resources than they should.

When the SGSN reaches its saturation point, resources become limited. At this point, the fact that misbehaving users receive more resources than they should affects the throughput of well behaved users. Again, Figure 7.5 clearly shows this behavior. The resources consumed by well-behaved users diminish as the amount of exceeding traffic increases. This affects their throughput. Figure 7.5-b shows that well behaved users only receive about 50% of their allocated bandwidth when the fraction of misbehaving traffic reaches 50%. Misbehaving users still manage to receive 100% of their allocated bandwidth. This shows that a user level SGSN is not able to provide proper isolation between users when it becomes overloaded.

Because it manages resources properly and is able to discard misbehaving traffic early, the ED-SGSN is able to remain below its saturation point and not be affected by the exceeding traffic. Figure 7.6 shows this behavior. The amount of resources consumed by both well-behaved and misbehaving users is identical. Both types of users are also able to get their entire allocated bandwidth, regardless of the amount of exceeding traffic.

Figures 7.7 and 7.8 show the same graphs for the 256-user scenario. In this case both SGSN and ED-SGSN are overloaded. The same behavior as in the 160-user scenario is observed for the SGSN. The numbers are lower because of the severity of the overload but the fact remains that misbehaving users receive more resources and more bandwidth than well-behaved users. The ED-SGSN, although affected by the overload created by a large amount of users, is not really affected by the quantity of misbehaving traffic. Resources, although saturated, are distributed evenly among all users and each user receives the same bandwidth. The throughput

of each user remains fairly constant regardless of the quantity of exceeding traffic.

These results show that the ED-SGSN is able to provide proper isolation and fairness among users independently of the load on the system and of the quantity of misbehaving traffic generated by its users.

7.4 Delay and Support for Real-time Traffic

Multimedia applications such as audio streaming and video conferencing are becoming very popular. There is a good chance those applications will also be popular with GPRS users. In order to support such applications, GPRS must be able to offer a guaranteed low delay, low jitter service.

7.4.1 Real-Time Requirements of Multimedia Traffic

Multimedia streaming is characterized by the need to output a signal at fixed intervals. This signal can be audio or video. The multimedia application processes incoming data from the network to build the signal. If the application cannot output the signal at the right moment because the required data is not available, then the quality of the output suffers; video frames are missing or there are glitches in the audio track. This data is packetized at the server and transmitted over the network to the client application. The moment in time where a packet must be available to the application, in order for the signal to be outputted correctly, is defined as the packet's deadline.

Transmitting information over a packet-based network inevitably introduces fluctuation in the delay between each packet. This fluctuation, known as jitter, is caused by the variable queuing delay that packets experience in their journey through the network. Jitter can cause packets to arrive late and miss their deadline.

To eliminate problems caused by jitter, an application can buffer data for a certain time before starting the playback of the signal. Buffering works well with non real-time one-way streaming applications, where the signal is pre-recorded and startup delay is unimportant, but has very limited use in real-time, two-way streaming applications, such as video conferencing or packetized voice transmission. Because buffering adds additional delay, it causes substantial lag in such applications.

To support applications with such real-time requirements, the underlying network must be able to offer a low delay, low jitter service. Such service must be able to prioritize packets belonging to real-time applications in order to minimize their queuing delay at each node on the network.

7.4.2 Low Delay Traffic Class

GPRS has provisions for a low-delay traffic class. Delay is one of the four parameters defined in the GPRS QoS specification [8] [14]. The technical specifications define four delay classes that can be used by an operator to define different QoS profiles. A network operator wanting to support a real-time traffic class provides a QoS profile using the lowest delay class. The GPRS specifications do not specify or recommend an implementation, nor a scheduling algorithm for providing such a service.

To ensure that real-time traffic obtains a guaranteed low delay, low jitter service, forwarding nodes such as a SGSN must be able to support a forwarding behavior that is independent of the traffic load at the node. This can be done by using a forwarding mechanism such as a priority queue to make sure the delay experienced by real-time traffic is minimized and bounded.

We show that a user-level SGSN will be unable to provide such a service even if it supports the appropriate scheduling mechanism. This problem is caused

by the queuing of incoming packets in the UDP queue. Because incoming packets are queued in FIFO order, packets belonging to a real-time traffic class are not prioritized and suffer a delay proportional to the queue length. This delay is completely independent of the scheduling algorithm used by the SGSN to support QoS. Depending on the algorithm used, an additional queuing delay may be added on top of the UDP queuing delay.

We also show that the ED-SGSN does not suffer from this problem. Because packets are identified on arrival, it is very easy to provide proper traffic prioritization. This allows the real-time traffic class to avoid the FIFO queuing behavior of the UDP input queue. This minimizes the delay real-time traffic experiences while traversing the SGSN protocol stack.

7.4.3 Experiments and Results

The goal of this experiment is to show the load-dependant latency real-time traffic will experience traversing a user-level SGSN. We also show that the ED-SGSN doesn't suffer from this problem.

The traffic model we use consists of one real-time mobile, plus additional best effort mobiles, to create an extra traffic load on the system. The real-time mobile receives data at its maximal configured rate. A uniform distribution is used to generate the traffic for the real-time mobile. Exponential distributions are used to generate the additional best-effort load.

To measure the delay experienced by each packet, the traversal time of real-time packets is measured. The same module described in Section 7.2 is used to measure the traversal time. Each experiment is run for a period of 1000 seconds and only the last 2048 measurements are kept.

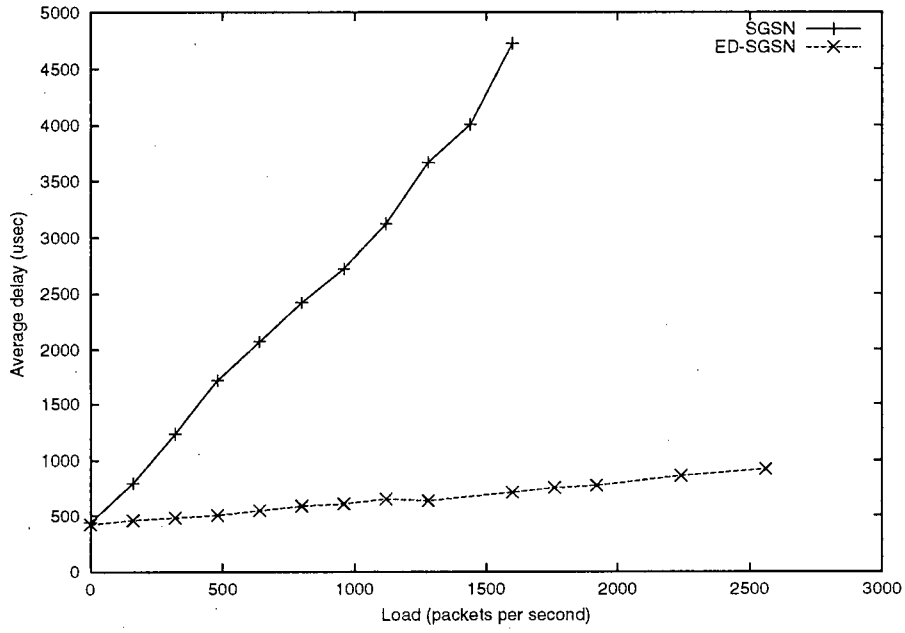


Figure 7.9: Average delay for real-time class

Figure 7.9 shows the average delay experienced by a real-time user in both the SGSN and ED-SGSN. We can clearly see that the delay experienced by this user on the SGSN grows quickly as the load on the system increases. We are unable to measure the delay experienced by real-time packets past 1600 packets per second on the SGSN, because at this point the SGSN starts dropping too many packets in the UDP socket buffer, making it impossible to measure the delay. There is no reason to believe the delay would not keep growing in a similar fashion past 1600 packets per second. The ED-SGSN probably starts dropping packets in some of its input queues at a similar load, but since real-time packets are prioritized on arrival, these packets do not suffer the same fate as on the SGSN. Packets belonging to the best-effort class are probably dropped instead.

The behavior experienced by real-time traffic on the SGSN is quite undesir-

able. The problem is not only that the delay becomes large when the system is fully loaded but that this delay varies by an order of magnitude (from 0.5 milliseconds to almost 5 milliseconds) between the no-load and the fully loaded situations. Having the delay as a function of the load on the system can cause large amounts of jitter. Over a long period of time, the load on an actual system will vary substantially. This will cause large variations in the delay experienced by real-time traffic at the SGSN. Jitter is undesirable and problematic because it makes it hard for real-time applications to tune themselves to the conditions of the network. A larger but constant delay is often better than a varying delay because it is easier for applications to adjust themselves to a constant delay. Parameters such as buffer requirements or knowing if it is even possible to run the application are easily determined when the delay is fairly constant.

Most delay, if not all, experienced by real-time traffic at the SGSN comes from time spent in the input queue. By using a priority queue as a packet scheduler in the flow control unit, the SGSN is able to properly prioritize packets being sent out on the Gb interface. The use of a priority queue in the flow control unit is described in detail in Section 5.1.3. Unfortunately, there is nothing a user-level SGSN can do about the delay at the input queue. The operating system, which queues incoming traffic, knows nothing about the priority of each of the incoming packets.

By classifying packets on arrival before they are queued, the ED-SGSN is able to properly prioritize real-time traffic. When there is very little load on the system, the delay is the same on both the SGSN and ED-SGSN. The difference is that the ED-SGSN is able to keep this delay low under any load. The increase in delay from a no load situation to a load of 1600 packets per second is around 285

microseconds. This is still an increase of 67%, but 285 microseconds is a fairly small amount of time when put in the context of end-to-end delay experienced in wide area networks, and is probably quite acceptable. Compared to the increase in delay experienced at the SGSN for the same load interval, this is an improvement of about 1500%.

The increase in delay experienced at the ED-SGSN is probably caused by the increase in the number of interrupts. As the load on the system increases, the frequency of interrupts generated by the network interface card also increases. Since interrupts have the highest priority in the system, they will always preempt the processing of packets through the SGSN protocol stack. The probability of having an interrupt preempt the processing of a packet increases with the load on the system and so will the delay. Still, the ED-SGSN is able to provide a low delay low jitter service, which is by far superior to what a regular user-level SGSN can provide.

7.5 Precedence

7.5.1 Overview

Precedence defines the importance of a packet or a user. Packets belonging to low precedence traffic classes are always dropped before packets belonging to high precedence classes. Supporting different levels of precedence enables the definition of different services based on their importance. One can think of a premium service that is of greater importance than a regular service. A user subscribing to such a service will always get priority in terms of maintaining service under abnormal conditions, such as link congestion or overload periods. Regular subscribers' packets

are always dropped before premium service subscribers' packets.

Precedence is one of the parameters defined in the GPRS QoS specifications [8] [14]. GPRS supports three different levels of precedence allowing a network operator to define various services of different importance.

An SGSN wishing to support different levels of precedence will implement support for it in the packet scheduler of the flow control unit. This will allow it to maintain high precedence services ahead of other services when cells become congested. Unfortunately, a user-level SGSN will be unable to properly support different levels of precedence once its processing resources become saturated. The reasons for this are closely related to why a user-level SGSN cannot support real-time traffic.

The operating system is unaware of the QoS requirements of incoming traffic. All incoming packets are treated identically by the operating system. When the input queue fills up, additional traffic is simply dropped in a tail-drop manner, regardless of the precedence of each packet.

The ED-SGSN can avoid this problem because it sees every incoming packet. The ED-SGSN can make informed decisions when selecting which packet should be dropped. This allows the ED-SGSN to properly support different precedence levels even when overloaded. In the case of a user-level SGSN, the operating system sees every packet, but the SGSN process only sees the packets it is able to read from the socket buffer.

7.5.2 Experiments and Results

We have measured how well precedence is supported in both the SGSN and ED-SGSN by looking at how traffic is controlled by the SGSN in an overloaded cell.

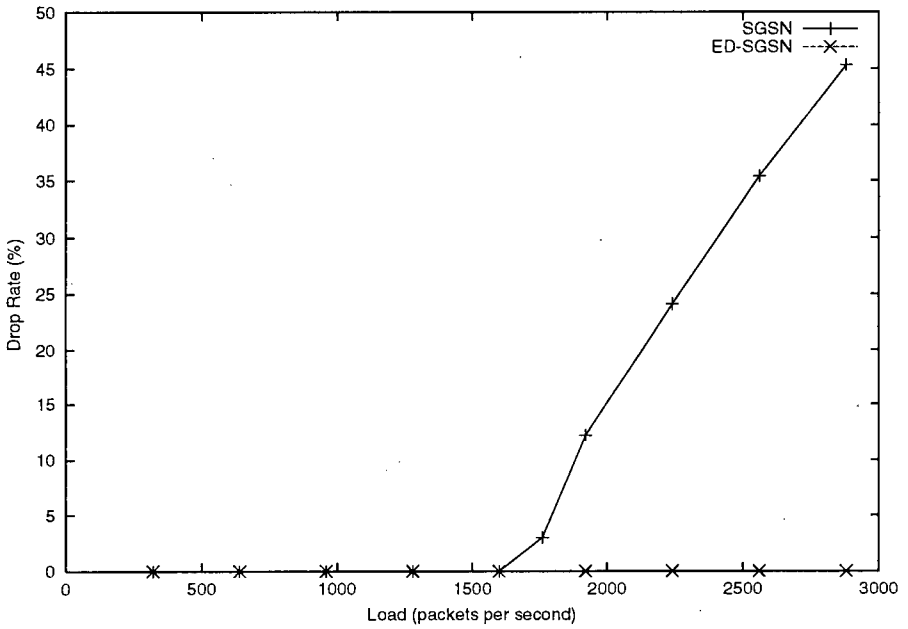


Figure 7.10: Drop Rate for High Precedence Traffic Class

This experiment shows how well the SGSN and ED-SGSN support precedence under various loads.

The cell we looked at serves five mobiles, all of which transmitting at 4000 bytes per second or ten packets per second. The cell has a capacity of 16000 bytes per second, which is not enough to support all five mobiles. One mobile belongs to the high-precedence class while the others belong to the regular precedence class. To make sure the individual mobile's flow control unit doesn't interfere, the allocated bandwidth for each user is set to 6000 bytes per second.

Figure 7.10 shows the drop rate for the high-precedence mobile user in the congested cell. For both the SGSN and ED-SGSN, the precedence mechanism in the flow mechanism properly maintains the high precedence service below the saturation point. Until it saturates, the SGSN drops none of the packets belonging to the high

precedence user. The support for different precedence levels fails on the SGSN as soon as the saturation point is reached. The drop rate of the high precedence user rises dramatically, reaching 45% at around 2900 packets per second. This number will probably keep growing if the load is increased.

Every high-precedence packet that the SGSN drops is dropped at the input queue by the operating system. The operating system controls which packet is to be dropped at the input queue and it does so without taking into account the precedence level of each packet. Service differentiation based on precedence disappears once the saturation point is reached on the SGSN.

The ED-SGSN can maintain a high precedence service perfectly, well past its saturation point. It is easy to see in Figure 7.10 that the drop rate stays at zero for every experiment run. Even at rates near 3000 packets per second, the ED-SGSN did not drop a single high precedence packet.

Chapter 8

Conclusion

In this thesis we have studied an SGSN from a system point of view. We have looked at the interaction between the operating system and the SGSN and analyzed how this interaction affects the performance of the SGSN. We have studied the overall performance of the system and we have looked at how well guaranteed services are supported.

Our study shows that an SGSN implemented as a user-level process will not support guaranteed services well when heavily loaded. We have looked at three Quality of Service(QoS) metrics: throughput, delay and precedence. In each case, the user-level SGSN failed to properly support services based on one of the metrics.

The overall throughput of a user-level SGSN is highly influenced by misbehaving traffic when overloaded. Isolation between the bandwidth allocated to each user is not maintained. Well-behaved users see their throughput affected by misbehaving users. We also show that a user-level SGSN will not be able to support a guaranteed low delay traffic class and that it does not support different precedence levels when overloaded.

All these problems are caused by the fact that there is currently no mecha-

nism available to an application to specify how incoming network traffic should be classified and queued by the operating system. The problem can be summarized in three main points:

- Packets are queued in a FIFO manner by the operating system ignoring the packets QoS requirements. This causes high priority packets to be queued behind lower priority traffic causing delays that are highly dependent on the system load.

- Once the input queue fills up, the operating system simply drops additional incoming packets regardless of their precedence.

- Packets are also processed through the SGSN protocol stack in the order they arrive at the SGSN, regardless of their QoS requirements and independently of the fact that their owner may have exceeded its allocated bandwidth.

We have built an SGSN where incoming traffic is classified and queued accordingly in interrupt context. Packets are queued on a per mobile basis on arrival. Because packets are identified on arrival, their precedence is immediately determined and the appropriate action can be taken. Instead of processing packets in the order they arrived, a packet scheduler decides which packet should be processed next. The scheduler takes into account the priority of each packet, the allocated bandwidth of each user and the status of the output queues. This allows the proper prioritization of packets, making sure that high-priority packets are not needlessly delayed.

Processing resources are allocated to each user according to its allocated bandwidth, making sure that misbehaving users do not consume an inappropriate amount of resources. This ensures that isolation between users is maintained even when the system is overloaded. Also, no resources are wasted on processing packets

through the SGSN protocol stack and then dropped because their output queue is full.

Results show that the Early-Demultiplexing SGSN(ED-SGSN) brings major improvements to all three QoS metrics identified earlier. The overall throughput of the system is maintained to a greater extent when faced with a large amount of exceeding traffic. Improvements of up to 42 % can be seen when the system is faced with large amounts of misbehaving traffic. Isolation between users is maintained even when the system is overloaded with large amounts of exceeding traffic.

The ED-SGSN also properly supports a low-delay real-time traffic class, providing only an increase of less than 300 microseconds between a no-load situation and a load of 1600 packets per second. The delay experienced at the regular SGSN varies by over 4 milliseconds for the same load interval. Precedence is also very well supported with no dropping of high precedence packets even well above the saturation point. This is in sharp contrast to the plain SGSN, which fails to support precedence when overloaded, even slightly.

Although early-demultiplexing is an idea that has been around for a while in the area of systems research, it has never been applied to GPRS. Our work shows that it is feasible to implement an early-demultiplexing mechanism on a SGSN even with the complexity of the protocol stack.

As far as we know, the idea of combining the use of early-demultiplexing with a feedback mechanism monitoring the status of the output queues, to avoid wasting processing resources, is novel. The usage of a packet scheduler for determining how packets should be processed throughout the rest of the system, in order to obtain proper resource allocation within the system, is also novel.

In terms of systems research, our work can be seen as a first step in establish-

ing the requirements of an operating system with support for Quality of Service in the I/O subsystem. Also, we believe our in-depth performance analysis of an SGSN from a system point of view could be used in communication and networking research to build a more accurate model of an SGSN that could be used in large-scale GPRS network studies.

Bibliography

[1] T.E. Anderson, B.N. Bershad, E.D. Lazowska, and H.M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–70, February 1992.

[2] D. A. Bandel. Taming the wild netfilter. *Linux Journal*, 89:64, 66–68, 70, 72, September 2001.

[3] G. Banga and J. C. Mogul. Scalable kernel performance for internet servers under realistic loads. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 1–12, Berkeley, USA, June 15–19 1998. USENIX Association.

[4] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. RFC 2475: An architecture for differentiated services, December 1998.

[5] E. Braden, L. Zhang, S. Berson, S. Herzog, and S. Jamin. RFC 2205: Resource ReSerVation Protocol (RSVP) — version 1 functional specification, September 1997.

[6] D. D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[7] P. Druschel and Q. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In USENIX, editor, *2nd Symposium*

on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA, pages 261–275, Berkeley, CA, USA, October 1996. USENIX.

[8] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Service Description; Stage 2 (GSM 03.60 version 6.1.1 Release 1997). *ETSI*, 1998.

[9] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Base Station System(BSS) - Serving GPRS Support Node (SGSN); BSS GPRS Protocol (BSSGP) (GSM 08.18 version 6.3.0 Release 1997). *ETSI*, 1999.

[10] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Base Station System(BSS) - Serving GPRS Support Node (SGSN) interface; Network Service (GSM 08.16 version 6.3.0 Release 1997). *ETSI*, 1999.

[11] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); GPRS Tunnelling Protocol(GTP) across the Gn and Gp interface; (GSM 09.60 version 7.0.0 Release 1998). *ETSI*, 1999.

[12] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Mobile Station - Serving GPRS Support Node (MS-SGSN); Logical Link Control (LLC) layer specification (GSM 04.64 version 6.3.0 Release 1997). *ETSI*, 1999.

[13] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Mobile Station (MS) - Serving GPRS Support Node

(SGSN); Subnetwork Dependent Convergence Protocol (SNDCP) (GSM 04.65 version 6.3.0 Release 1997). *ETSI*, 1999.

[14] ETSI. Digital cellular telecommunications system (Phase 2+); General Packet Radio Service(GPRS); Service Description; Stage 1 (GSM 02.60 version 6.3.1 Release 1997). *ETSI*, 2000.

[15] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, August 1999.

[16] B. Gallmeister. *POSIX.4: Programming for the Real World.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, January 1995.

[17] R. Gopalakrishnan and G. M. Parulkar. Efficient user-space protocol implementations with QoS guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4):374–388, August 1998.

[18] ITU-T. Recommendation V.42bis - data compression procedures for data circuits-terminating equipment (dce) using error correcting procedures. *Geneva, Switzerland*, February 1998.

[19] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, February 1990.

[20] V. Jacobson, K. Nichols, and K. Poduri. RFC 2598: An expedited forwarding phb, June 1999.

[21] C. Lee, K. Yoshida, C. Mercer, and R. Rajkumar. Predictable communication protocol processing in real-time mach. In *IEEE Real-Time Technology and*

Applications Symposium (RTAS '96), pages 220–229, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[22] C. Maltzahn, K. J. Richardson, and D. Grunwald. Performance issues of enterprise level web proxies. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, volume 25,1 of *Performance Evaluation Review*, pages 13–23, New York, June 15–18 1997. ACM Press.

[23] M. McKusick, K. Bostic, and M. Karels. The design and implementation of the 4.4BSD operating system. *Addison-Wesley*, May 1996.

[24] M. K. McKusick and M. J. Karels. Design of a general purpose memory allocator for the 4.3BSD UNIX kernel. In USENIX Association, editor, *USENIX Conference Proceedings, Summer, 1988. San Francisco*, pages 295–303, Berkeley, CA, USA, Summer 1988. USENIX.

[25] A. Mishra. Performance and architecture of sgsn and ggsn of general packet radio service(gprs). In *Global Telecommunications Conference, 2001. GLOBECOM '01. Volume 6.* IEEE, 2001.

[26] J. C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.

[27] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 153–167, Berkeley, CA, USA, October 1996. USENIX.

[28] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 headers, December 1998.

[29] J. K. Ousterhout. Why threads are A bad idea (for most purposes). Invited Talk at the 1996 USENIX Technical Conference, January 1996.

[30] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the 1999 USENIX Annual Technical Conference (USENIX-99)*, pages 199–212, Berkeley, CA, June 6–11 1999. USENIX Association.

[31] Q. Pang, A. Bigloo, V.C.M. Leung, and C. Scholefield. Service scheduling for general packet radio service classes. In *IEEE Wireless Communications and Networking Conference*, pages 1229–1233, 1999.

[32] J. Postel. RFC 768: User datagram protocol, August 1980.

[33] J. Postel. RFC 793: Transmission control protocol, September 1981.

[34] G. Priggouris, S. Hadjiefthymiades, and L. Merakos. Supporting IP QoS in the General Packet Radio Service. *IEEE Network*, pages 8–17, September 2000.

[35] M. Puuskari. Quality of service framework in gprs and evolution towards umts. In *3rd European Personal Mobile Communications Conference*, Paris, France, mar 1999.

[36] D. M. Ritchie. A stream input-output system. *BSTJ*, 63, 8:1897–1910, 1984.

[37] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Bell System Technical J.*, 57(6):1905, July-August 1978.

[38] D. Staehle, K. Leibnitz, and K. Tsipotis. QoS of internet access with GPRS. *The Fourth ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2001)*, 2001.

[39] R. Stevens. *UNIX network programming: Networking APIs: sockets and XTI*, volume 1. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, second edition, 1998.

[40] P. Stuckmann and F. Muller. GPRS radio network capacity and quality of service using fixed and on-demand channel allocation techniques. *Proc. Vehicular Technology Confernece (VTC spring 2000)*, May 2000.

[41] V. Sundaram, P. Shenoy, A. Chandra, J. Sahni, P. Goyal, and H. Vin. Application performance in the QLinux multimedia operating system. In *Proceedings of the 8th International ACM Conference on Multimedia (Multimedia-00)*, pages 127–136, N. Y., October 30–November 04 2000. ACM Press.

[42] F. Tataranni, S. Porcarelli, F. Di Giandomenico, A. Bondavalli, and L. Simoncini. Modeling and analysis of the behavior of gprs systems. In *Proceedings of the Sixth International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 5–12. IEEE, 2001.

[43] D. L. Tennenhouse. Layered Multiplexing Considered Harmful. In *Proceedings of the 1st International Workshop on High-Speed Networks*, May 1989.

[44] U. Vahalia. *UNIX Internals.* Prentice-Hall, Upper Saddle River, NJ 07458, USA, 1996.

[45] J. Yang, C. Tseng, and Cheng C. Dynamic scheduling framework on rlc/mac layer for general packet radio service. In *2001 International Conference on Distributed Computing Systems Workshop*, pages 441–447. IEEE, 2001.