MASTER'S THESIS

Modeling CPU usage of a GPRS protocol layer using Coloured Petri Nets

by
A.C. de Keijzer

Supervisor: dr. M.R.V. Chaudron
Eindhoven, January 2004
Abstract

This Master’s Thesis documents a research project about modelling CPU usage. The modelling language Coloured Petri Nets has been used for this. Coloured Petri Nets have been used for many years already to prove functional correctness of a system. This document describes a method how CP-nets can be used for modelling a non-functional property, CPU usage. Simulations of this model will show the expected CPU usage.

Since this is an exploratory research, it will not give a conclusion about which implementation is best or how much CPU is used by the modelled implementation. This document will evaluate what kind of information can be extracted from such a model and how this information may improve the development of a system.

The LLC layer of the GPRS protocol stack will be used as a case study. The model of the LLC layer has been used to show how the trade-off between memory and CPU can be quantified and how CPU usage of different implementations may be influenced differently by changes in the usage. The results of the two examples in this document illustrate how this method can be used in many different situations.
# Contents

1. Introduction .................................................................................................................. 7
   1.1 OSI reference model .................................................................................................. 8
2. The GPRS protocol stack ............................................................................................... 10
   2.1 General Packet Radio System (GPRS) ..................................................................... 10
   2.2 Network architecture ............................................................................................... 10
   2.3 GPRS protocol stack ............................................................................................... 11
   2.4 LLC layer ................................................................................................................ 12
   2.5 Logical Link Entity ................................................................................................... 15
      2.5.1 Relevant part of the interface ........................................................................... 15
      2.5.2 Used sequence numbers and variables ......................................................... 16
      2.5.3 Transmission of I frames .............................................................................. 16
      2.5.4 Receipt of I frames ......................................................................................... 17
3. The CPN model ............................................................................................................. 18
   3.1 Coloured Petri Nets ................................................................................................. 18
      3.1.1 Places, arcs and transitions ........................................................................... 18
      3.1.2 Colorsets ......................................................................................................... 19
      3.1.3 Concurrency and conflicts .............................................................................. 19
   3.2 Design/CPN tool ....................................................................................................... 19
      3.2.1 Functional language ....................................................................................... 19
      3.2.2 Colorset definitions ....................................................................................... 20
      3.2.3 Hierarchy ......................................................................................................... 20
      3.2.4 Fusion Places .................................................................................................. 21
      3.2.5 Code Region ................................................................................................... 22
      3.2.6 Statistical Variables ...................................................................................... 23
      3.2.7 Time ................................................................................................................ 23
   3.3 Modelling interfaces in a layered system .................................................................. 24
      3.3.1 Colorset definition of an interface .................................................................. 25
      3.3.2 Alternative 1: implementing a queue ............................................................ 25
      3.3.3 Alternative 2: using a mirror place ............................................................... 26
   3.4 Abstraction ................................................................................................................ 27
   3.5 Design/CPN model ................................................................................................... 28
      3.5.1 Top-level structure .......................................................................................... 29
      3.5.2 Definition of LLC-LLE, as a subsystem of the top-level structure .............. 32
      3.5.3 Definition of Uplink, as a subsystem of LLE ................................................. 33
      3.5.4 Definition of Send_Data, as a subsystem of Uplink ...................................... 34
      3.5.5 Definition of Receive_Ack, as a subsystem of Uplink .................................... 35
      3.5.6 Definition of Downlink, as a subsystem of LLE ............................................. 36
      3.5.7 Definition of Receive_Data, as a subsystem of Downlink .............................. 37
      3.5.8 Definition of Send_Ack, as a subsystem of Downlink .................................... 38
      3.5.9 Definition of RLC layer, as a subsystem of the top-level structure .............. 39
      3.5.10 Introduction of the concept of time ............................................................... 39
4. Estimation of CPU usage ............................................................................................... 42
   4.1 Simulation ................................................................................................................ 42
   4.2 Comparison between different implementations .................................................... 42
   4.3 Quantifying the results ............................................................................................ 43
      4.3.1 Counting CPU clock cycles ......................................................................... 44
      4.3.2 Counting basic actions .................................................................................. 44
   4.4 Storing the results during a simulation .................................................................... 44

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
4.4.1 Additional places ................................................................. 45
4.4.2 Extended tokens ................................................................. 45
4.4.3 Statistical variables ............................................................. 45
4.5 Obtaining accurate results ...................................................... 46
4.6 Implementation using statistical variables ................................. 48
  4.6.1 Used names for counters .................................................... 48
  4.6.2 Combination and storage of the results ............................... 49
  4.6.3 Comparing different scenarios .......................................... 49
  4.6.4 Implementation 1 & 2 ...................................................... 49
  4.6.5 Implementation 3 ............................................................. 50
  4.6.6 Implementation 4 ............................................................. 51
5. Results .................................................................................... 53
  5.1 Tables of results ................................................................. 53
  5.2 Comparison between implementation 1 and 2 ....................... 54
  5.3 Comparison between implementation 3 and 4 ....................... 55
6. Conclusion ............................................................................... 58
7. References ............................................................................... 60
Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3GPP</td>
<td>Third Generation Partnership Project</td>
</tr>
<tr>
<td>ABM</td>
<td>A-synchronous Balanced Mode</td>
</tr>
<tr>
<td>BSS</td>
<td>Base Station System</td>
</tr>
<tr>
<td>BSSGP</td>
<td>BSS GPRS Protocol</td>
</tr>
<tr>
<td>CPN</td>
<td>Coloured Petri Net</td>
</tr>
<tr>
<td>CP-net</td>
<td>Coloured Petri Net</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>GGSN</td>
<td>Gateway GPRS Support Node</td>
</tr>
<tr>
<td>GMM</td>
<td>GPRS Mobility Management</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio System</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communication</td>
</tr>
<tr>
<td>GSM RF</td>
<td>GSM Radio Frequency</td>
</tr>
<tr>
<td>I frame</td>
<td>Information frame</td>
</tr>
<tr>
<td>LLC</td>
<td>Logical Link Control</td>
</tr>
<tr>
<td>LLE</td>
<td>Logical Link Entity</td>
</tr>
<tr>
<td>MAC</td>
<td>Medium Access Control</td>
</tr>
<tr>
<td>MS</td>
<td>Mobile Station</td>
</tr>
<tr>
<td>OSI</td>
<td>Open System Interconnection</td>
</tr>
<tr>
<td>RLC</td>
<td>Radio Link Control</td>
</tr>
<tr>
<td>S frame</td>
<td>Acknowledgement frame</td>
</tr>
<tr>
<td>SGSN</td>
<td>Supporting GPRS Support Node</td>
</tr>
<tr>
<td>SMS</td>
<td>Short Message Service</td>
</tr>
<tr>
<td>SNDTCP</td>
<td>SubNetwork Dependent Convergence Protocol</td>
</tr>
<tr>
<td>TOM</td>
<td>Tunnelling Of Messages</td>
</tr>
</tbody>
</table>

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
1. INTRODUCTION

The research that is described in this document is a Master Thesis project and has been conducted as a part of the ITEA Robocop project. Robocop stands for Robust Open Component Based Software Architecture for Configurable Devices. It is a joint project of various European companies, both SME and large, together with private and public research institutes. The project is funded in part by the European ITEA program.

The aim of Robocop is to define an open, component-based partial architecture for the middleware layer in high-volume embedded appliances that enables robust and reliable operation, upgrading and extension, and component trading. The Robocop architecture is open in the sense of non-proprietary and in the sense of extensible.

The appliances targeted by Robocop are network gateways and consumer devices such as mobile phones and set-top boxes. The framework, however, aims to be equally applicable for high-volume professional devices such as PLCs and certain medical devices.

The Robocop project has been divided into four task-groups, each focusing on a part of the Robocop project. The research that is documented here is a part of the Resource Modelling group. This group used the modelling language Coloured Petri Nets with the tool Design/CPN. Therefore, this research has focused on the use of CP-nets.

The research took place at Nokia Research Centre in Helsinki, Finland. There were several good reasons to choose the GPRS protocol stack as a case study. First of all, GPRS is a new and large system. Results about non-functional properties can be used directly by Nokia. Furthermore, this system is being implemented by different parties, and will be used widely. GPRS is not just a simple and small system, and thus it should be possible to achieve similar results on other industrial systems as well.

The goals of this project are to find a way of modelling resource usage, in our case CPU, within a system, and to discover how this way of resource modelling can help to improve the development of new systems. How can this model be used to generate an estimate of CPU usage? What kind of results can be expected? Which conclusions can be drawn from these results? Which decisions can be taken, based on these conclusions? And how do these decisions improve the performance of the system or how can they provide economic benefits for the company?

The GPRS protocol stack is probably not known for most readers of this document. Therefore, the basic functionality of the layers of the GPRS protocol stack will be explained with help of the OSI reference model, of which a short description has been included below, in section 1.1. More details about GPRS and the specification of the LLC layer have been included in chapter 2. A Coloured Petri Net model of the LLC layer will be explained in chapter 3. Chapter 4 explains how this model has been extended to model CPU usage and how the estimates are generated by running simulations of the model. The results and evaluation are given in chapter 5. This is followed by a conclusion about the entire project in chapter 6.
1.1 OSI reference model

The Open System Interconnection (OSI) reference model describes how information from a software application in one computer moves through a network medium to a software application in another computer. The OSI reference model is a conceptual model composed of seven layers, each specifying particular network functions. The model has been developed by the International Organization for Standardization (ISO) in 1984, and it is now considered the primary architectural model for inter-computer communications.

The OSI model divides the tasks involved with moving information between networked computers into seven smaller, more manageable task groups. A task or group of tasks is then assigned to each of the seven OSI layers. Each layer is reasonably self-contained so that the tasks assigned to each layer can be implemented independently. This enables the implementation offered by one layer to be changed without adversely affecting the other layers. Figure 1 shows the seven layers of the Open System Interconnection (OSI) reference model within a network architecture.

Figure 1: Network architecture based on OSI reference model

The functions of the seven different layers will be explained briefly below.

7) Application layer:
The application layer provides access to the OSI environment. This layer provides the services required for an application running on one system to interact with applications running on other systems. The functions performed at this layer include establishing contact between processes and setting conditions of interaction.

6) Presentation layer:
The presentation layer is responsible for converting the information sent by the
application layers. The presentation layer is also responsible for maintaining the syntax and structure of the data.

5) Session layer:
The session layer establishes and maintains connections between applications and manages the dialog between them. The session layer determines which side is talking and which is listening at any given time, to ensure interaction proceeds in an orderly manner.

4) Transport layer:
The transport layer is responsible for the end-to-end communication between cooperating systems, regardless of their network characteristics. The transport layer delivers information from one application process to another and masks any failures of the underlying network services.

3) Network layer:
The network layer is the first layer to control communication between computers rather than processes. It is responsible for locating the device where a remote process is located. The network layer is also responsible for routing, addressing and flow control.

2) Data link layer:
The data link layer is responsible for transmitting data over a single link while carrying out error detection/correction, flow control, link sequencing and maintaining link integrity.

1) Physical layer:
The physical layer standardises how bits are moved from point to point over a physical medium. This includes specifying electrical or optical signal characteristics, connectors, encoding of digital signals and timing. It also includes the exchanges of control messages and handshaking procedures.
2. THE GPRS PROTOCOL STACK

This thesis is a case study that consists of a performance analysis on a part of the GPRS protocol stack. This chapter gives a general explanation about this new protocol. The need for this new protocol is explained in section 2.1. Section 2.2 shows the physical network architecture and section 2.3 will explain some more details about the protocol stack. In section 2.4, one layer of the protocol stack, the Logical Link Control (LLC) layer, will be explained in more detail. This layer is used for the CPN model, on which the estimation will be performed.

2.1 General Packet Radio System (GPRS)

During the last few years more and more data oriented services, like reading email and viewing web pages, have been developed for mobile phones. This changed the use of mobile stations from speech transmission towards transmission of data.

Data transfer consists of frequent communication of small amounts of data, alternated by infrequent communication of large amounts of data. So, the communication consists of irregular bursts of data. A circuit switched mobile service like GSM is not designed for this type of communication.

In circuit switched data transfer, a channel is opened for a certain period in which data can be transferred. Circuit switched data transfer is slow and it wastes radio bandwidth. Since a fixed amount of resources will be reserved for one user, other users cannot use radio resources before the connection is closed.

Packet switched data transfer is much more suitable for transmission of data. Radio resources will be reserved only when needed to transmit a chunk of data. A standardised service with the capability of packet switched data transfer is called GPRS.

GPRS allocates time slots only when the mobile station needs to transfer data. During a transmission, the number of time slots can be changed dynamically. Resources for uplink and downlink directions are reserved independently, according to the need. The network resources are released as soon as there is no longer need for data transfer. Different compression rates allow faster data transfer. All together, this results in short connection times and fast access to data services.

2.2 Network architecture

The physical GPRS network architecture has been constructed, in parallel, next to the existing GSM network. The GPRS network uses some of the existing physical GSM network elements as well. Some other elements are new, like the Serving GPRS Support Node (SGSN) and Gateway GPRS Support Node (GGSN). A picture of the GPRS network architecture is given in Figure 2.

The picture in Figure 2 gives a rough overview of the physical layout of the GPRS network; the most important elements are shown in this picture. Mobile Stations (MS) communicate via an air interface with the Base Station System (BSS), which is connected to Serving GPRS Support Node (SGSN). Different parts of the GPRS network are connected via the GPRS backbone. Via the Gateway GPRS Support Node (GGSN), the GPRS network is connected to the Internet.

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
2.3 GPRS protocol stack

An overall description of the GPRS protocol stack can be found in [5]. The protocol stack covers all the elements of the network architecture. The protocol stack is designed as a layered system. Different layers connect different network elements. Only the lowest layers have a physical connection between the elements that are covered by those layers. Higher layers build on the interface of the lower layer to implement more functionality. Only a part of the GPRS protocol stack, called the LLC layer, will be used later as a basis for the CPN model. Therefore, in the following section more attention will be given to the LLC layer.

Figure 3 shows the part of the stack that contains the LLC layer. The following physical elements play a role in this part of the system: MS (Mobile Station), BSS (Base Station System) and SGSN (Serving GPRS Support Node). How these elements are connected in the GPRS network is shown in Figure 2. The terms Um and Gb are names of the air interface that is used for the communication. Both are a GPRS interface that is located between the respective elements. The LLC layer is defined for both the MS and SGSN. This layer specifies how communication from MS to SGSN, and vice versa, is realised by using the functionality of the underlying layers.
**Figure 3: Protocol Layering in GPRS**

In order to get an idea of the functionality of each layer we will identify the position of each layer within the OSI reference model. The OSI model is often used as a reference model and it is explained in several books and web pages. A short description of this OSI model is given in chapter 1.

Figure 3 shows how, in GPRS, the GSM Radio Frequency is used as the lowest (Physical) layer to provide communication between MS and BSS; at this level, a stream of bits is communicated.

The layers MAC (Medium Access Control), RLC (Radio Link Control), as well as Network Service and BSSGP (BSS GPRS Protocol), are all part of the second OSI layer, called Data Link layer. The unit of communication is a frame. A frame is a structured series of bits, containing different data fields of variable size. Some kind of delimiter is used to distinguish between consecutive frames.

The LLC (Logical Link Control) is also considered a part of OSI layer 2 since most of its functions are related to typical Data Link layer tasks. Only the GPRS ciphering procedure, which is performed in the LLC as well, is uncommon for OSI layer 2. The LLC layer communicates frames, like common for layer 2 of the OSI reference model.

The other layers in Figure 3, TOM, GMM, SNDC and SMS, are all part of OSI layer 3 (Network). At this level, packets are communicated.

The following section will explain the LLC layer in more detail. Section 2.4 will also explain a little about the layers on top of the LLC (TOM, GMM, SNDCP and SMS) and the layers below (RLC and BSSGP).

**2.4 LLC layer**

The Logical Link Control (LLC) layer protocol is used for packet data transfer between MS and SGSN. This protocol is the lowest GPRS protocol that is independent of the used air interface protocols. It is very important to make the GPRS core network design as independent as possible from the air interface, since this enables the introduction of alternative GPRS radio solutions with minimal changes. The technical specification of the
LLC layer can be found in [8]. Many details, that are irrelevant for understanding the rest of this document, have been left out of the description of the LLC and LLE (section 2.5). For a good understanding of the LLC layer, or development of a new model or implementation, the technical specification should be used.

LLC provides services to the network layer protocols that GPRS uses, especially to the SNDCP (SubNetwork Dependent Convergence Protocol), SMS and GMM (GPRS Mobility Management). The points, where the LLC layer provides services to these layer 3 protocols, are called Service Access Points.

GPRS Mobility Management (GMM) uses the services of the LLC layer to transfer messages between the MS and the SGSN. GMM performs different management tasks, while the Mobile Station may move from one to another location.

The Short Message Service (SMS) uses the services of the LLC layer to transfer short messages between the MS and the SGSN.

The SubNetwork Dependent Convergence Protocol (SNDCP) uses the services of the LLC layer to transfer user data between the MS and the SGSN.

Tunnelling of Messages (TOM) is a generic protocol layer used for the exchange of so-called TOM Protocol Envelopes between the MS and the SGSN. These messages can be used to exchange any kind of information between different elements within the GPRS protocol stack.

The LLC layer uses services of the Radio Link Control (RLC) on the MS side; on the SGSN side it uses services of the BSS GPRS Protocol (BSSGP). The technical specifications of RLC and BSSGP can be found in [7] and [8]. For this document, it is sufficient to know that both RLC and BSSGP provide a simple interface to send unacknowledged or acknowledged variable-sized frames.

A schematic picture of this layer is shown in Figure 4.
Figure 4: Functional model of the LLC layer

The LLC layer provides a highly reliable logical link between the MS and the SGSN, independent of the underlying radio interface. It supports peer-to-peer data transfers of variable length messages, both acknowledged and unacknowledged. These messages are encapsulated in information frames, called L1 frames. Each frame starts with a header, which contains different fields that supply supervisory information. The body of an information frame contains the user data that needs to be transferred. Different MSs need to be able to use the same radio resources; and thus each frame must uniquely identify the MS sending or receiving the information. Transfers with different service criteria are allowed by the LLC, such that high-priority data transfers may take precedence over other transfers of the same MS. Furthermore, both user data and user identity confidentiality shall be provided by encryption of the frames.

Figure 4 shows how the LLC layer can be divided in 3 functionally different elements: Logical Link Management, Logical Link Entity and Multiplex Procedure.
The Logical Link Management manages parameter initialisation and performs error processing. Logical Link Management performs these management tasks for all the different Logical Link Entities. This way, GPRS Mobility Management can easily initialise all LLEs and process any error from one of the LLEs.

The Multiplex Procedure will perform frame ciphering to protect user data and user information confidentiality. Furthermore, it will insert the Frame Check Sequence field, which is used to detect communication errors. Finally, it merges and distributes all communications between the RLC or BSSGP layer and one of the LLEs.

The Logical Link Entity controls the information flow of individual connections. Each LLE provides both unacknowledged and acknowledged information transfers. For acknowledged information transfer, an LLE will first make a connection in so-called Asynchronous Bi-directional Mode (ABM). An LLE controls the flow in ABM operation and detects frame errors. Different LLEs may have different service criteria; this way, the LLC provides channels with a different priority. More details about the behaviour of an LLE will be given in the next section.

2.5 Logical Link Entity

Besides flow control in ABM operation and frame error detection, the Logical Link entity provides both unacknowledged and acknowledged information transfer. The tasks of the LLE can therefore be divided in two parts: establishing or maintaining a connection and transferring data. The establishment of a connection consists of the communication of several different commands and replies. This is not the main task for the LLE.

The main task of the LLE is the acknowledged transfer of user data. The LLE also supports the transfer of unacknowledged data, but this does not require any special processing. Acknowledged data transfer requires, besides buffering and transmission of frames, sending acknowledgements, re-transmitting lost frames and buffering received frames. The information frames are often referred to as I frames; the term S frames will be used for acknowledgement frames.

Since only acknowledged information transfer is necessary to understand the constructed CPN model, only this part of the LLE will be explained in some more detail. Section 3.4 will explain why only this part of the LLC layer has been modelled.

2.5.1 Relevant part of the interface

Within 3GPP, the interfaces between layers are defined as service primitives. Service primitives represent, in an abstract way, the logical exchange of information and control between the LLC layer and adjacent layers. It is up to each individual partner of the 3GPP to decide how to implement the interface between the layers of their GPRS protocol stack.

Layer 3 can use the primitive LL_DATA_REQ to request the transfer of data to the peer. The data to be transferred is contained in one of the parameter. When the peer has received the data, this data will be indicated to layer 3 of the peer, in FIFO order, by a LL_DATA_IND primitive. The correct transfer of data will be confirmed to layer 3 at the sender side by the primitive LL_DATA_CNF.

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
The LLE uses a simple interface to layer RLC or BSSGP. This interface consists of the primitives GRR_DATA_REQ and GRR_DATA_IND. The RLC or BSSGP layer can use the GRR_DATA_REQ primitive to request the transfer of a frame. The receipt of a frame is indicated to the LLC layer by the GRR_DATA_IND primitive.

Other parts of the interfaces are not used in our CPN model and will not be explained here.

2.5.2 Used sequence numbers and variables

Sequence numbers are used by the LLE for FIFO delivery of frames. The following sequence numbers and variable are used for this purpose.

- Send state variable V(S): this variable denotes the sequence number of the next in-sequence I frame to be transmitted. The value of V(S) is assigned to the N(S) of each new frame to be transmitted. After this assignment, V(S) is incremented by 1.

- Acknowledge state variable V(A): V(A)-1 identifies the N(S) of the last in-sequence acknowledged I frame.

- Send sequence number N(S): each I frame contains the send sequence number N(S).

- Receive state variable V(R): each LLE peer has an associated receive state variable V(R), which denotes the sequence number of the next in-sequence I frame expected to be received.

- Receive sequence number N(R): Each S frame contains N(R), the expected send sequence number of the next in-sequence received I frame. When such a frame is transmitted, the value of N(R) is set equal to V(R). N(R) indicates that the LLE peer has successfully received all I frames numbered up to and including N(R)-1.

- SACK bitmap R(n): Besides an N(R), S frames can also contain a SACK bitmap. This bitmap indicates which frames, with a sequence number higher than N(R), have been received by the LLE peer. The value of R(n) equals 1 if and only if I frame number N(R)+n has been received.

2.5.3 Transmission of I frames

The term "transmission of an I frame" refers to the delivery of an I frame by the LLC layer to the RLC or BSSGP layer.

Layer 3 can request a LLE to transfer a frame to the peer LLE, by means of the command LL_DATA_REQ.

A LLE may transmit only a limited amount of frames, without the receipt of their acknowledgements. This is called the transmit window. Each frame within the transmit window is either:

- not yet transmitted: the frame has not yet been transmitted;
- transmitted: the frame has been (re-) transmitted, but the LLE does not know if the frame has been received in the peer LLE;

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
- marked for retransmission: the LLE has decided to retransmit the I frame; or
- acknowledged: the frame has been acknowledged by the peer LLE;

I frames shall be transmitted in ascending N(S) order. When some I frames are marked for retransmission, the frame with the lowest N(S) shall be retransmitted first. This is used by the peer LLE to detect lost frames.

An acknowledgement can be requested explicitly, by setting the acknowledgement bit of this frame to 1. This will be done for example when the last information frame is transmitted. Based on the information, this acknowledgement contains, the LLC layer can decide which frames have been received by the peer and which frames require retransmission. This second set of frames will be marked for retransmission. Successful transfer of frames will be confirmed by a LL_DATA_CNF command for all frames that are acknowledged by one of the acknowledgement frames.

2.5.4 Receipt of I frames

The LLE receives information frames from the RLC or BSSGP layer. If this frame is in-sequence, then the information field will be passed on to layer 3 using the LL_DATA_IND primitive. If the acknowledgement bit of this frame is set to 1, then the LLE will respond with an acknowledgement frame to its peer.

If an I frame is not in-sequence, then this frame will be buffered. This frame will be indicated to layer 3 as soon as all frames, with a lower sequence number, have been received. Duplicate I frames will be ignored.

Whenever an LLE receives a frame with the acknowledgement bit set to 1, it shall transmit an acknowledgement frame as soon as possible.

The receiving LLE shall use the knowledge of the (re-) transmission strategy of its peer LLE to detect sequence errors. If the LLE receives an I frame with a higher N(S) than the N(S) of the previously received I frame, and there are frames missing between these two N(S) values, then the LLE shall assume that the missing I frames have been lost. If the LLE receives an I frame with a lower N(S) than the N(S) of the previously received I frame, it can assume that its peer LLE has started retransmission due to the reception of an acknowledgement. Whenever the receiving LLE detects the loss of an I frame, it shall transmit an acknowledgement frame as soon as possible.

An acknowledgement frame consists of two important data fields. The first field contains the value N(R), which indicates the sequence number of the expected (in-sequence) I frame. The second data field contains the SACK bitmap. This bitmap indicates which frames, with a sequence number higher than N(R), have been received already.
3. THE CPN MODEL

In this project, a Coloured Petri Net (CPN) model of the LLC layer has been constructed. This model will be simulated for obtaining performance data. The syntax and semantics of CP-nets will be assumed as knowledge of the reader and will be explained only briefly in section 3.1. A formal definition of CP-nets can be found in [1]. A lot of information about CP-nets can also be found on the Internet. A good place to start is Petri Nets World [2], which provides a variety of online services for the international Petri Nets community.

Section 3.2 will explain which tool is used for both modelling and simulation. Some things about layered systems in general are explained in section 3.3. A few alternatives for defining the interfaces are evaluated. Section 3.4 describes which part of the LLC layer contains the key function of this layer, and thus which part has been modelled by a CP-net. In section 3.5, the CPN model will be described; this section clarifies the design decisions only briefly.

3.1 Coloured Petri Nets

The development of Coloured Petri Nets is the result of a desire to develop a new modelling language that is both theoretically well founded and versatile enough to support the development of complex industrial projects. For this purpose, the strength of Petri nets has been combined with the strength of programming languages. Programming languages provide primitives for the definition of data types, and Petri nets provide theoretically well-founded primitives for synchronisation of concurrent processes.

Several tools for the construction and analysis of CPN models have been developed. This has helped the industry to use this modelling language in their projects. In the mean time, the industry has provided useful feedback that helped developers to improve and extend Coloured Petri Nets. Some of these extensions have been accepted widely, like extensions to hierarchical and timed CP-nets. These two extensions will be explained briefly in sections 3.2.3 and 3.2.7.

3.1.1 Places, arcs and transitions

The basic elements of Coloured Petri Nets are places, arcs and transitions. The two differences with classic Petri nets are the colorsets of places and inscriptions of the arc, because CP-nets support different data types.

Each place can contain multiple tokens. A transition can move tokens from one place to another, and thus model a state change of a system. The transition consumes a token from each place that is connected with an incoming arc. These places are also called input places. A token is produced into each (output) place that is connected with an outgoing arc.

![Figure 5: CPN - Place, arc, transition and tokens](image-url)
3.1.2 Colorsets

The term colorset is used in CP-nets for data types. Each place has a colorset, which defines the data type of each token in that place. Possible data types are integers, boolean or compound data types.

Transitions can generate tokens with different data values. Arc inscriptions describe the kind of tokens that are consumed or produced. Arc inscriptions are expressions that specify input-input and input-output relations. Arc inscriptions may only use free variables or values of input tokens. A transition can also define a separate firing condition, an expression that may use all values of input tokens. If there is no set of tokens that satisfies the condition (the transition is not enabled), then this transition will not fire.

The total state of the model, called the marking of the net, is defined by the number of tokens in each place and the value of each token.

Each CP-net can be rewritten to a classic Petri net, although infinite colorsets may result in an infinite classic Petri net. The advantage of a Coloured Petri net is that a system can be described at a more abstract level.

3.1.3 Concurrency and conflicts

Different transitions can share the same input place. One token in that place may enable both transitions. Only one transition may consume this token and fire, which disables the other transition. This is called a conflict.

In implementations of CP-nets, only one transition will fire at a time. The execution of this step is atomic, because otherwise the net may move to an inconsistent state. The order in which transitions will fire is non-deterministic. Transitions that are in a conflict have equal chance to fire.

The order in which transitions will fire is selected non-deterministically. Therefore, two parts of a CP-net that are not directly connected, will behave as if they are executed independently. This property is very useful when analysing a distributed system.

3.2 Design/CPN tool

Design/CPN [3] is a graphical tool that supports construction, simulation and functional analysis of CPN models. The simulation aspect of the tool will be very useful for obtaining performance data. Section 4.1 will explain why a simulation is the best solution for our analysis of the non-functional properties of the system.

A few concepts and terms of Design/CPN will be explained. Most parts of the model will be explained in common CPN terms. Sometimes, however, some specifics for Design/CPN are used. Each specific construction can be rewritten to a classic Petri Net.

3.2.1 Functional language

Design/CPN uses the CPN ML language for arc inscriptions and definitions of colorsets. This language is based on Standard ML (SML), a well-known functional programming language. More information on the functional language ML can be found in [4]. Design/CPN
has introduced some keywords for certain CPN specifics like colorset definitions; other parts, like the definition of functions are based on the ML syntax.

3.2.2 Colorset definitions

In CP-nets, the type of a token is defined by its colorset. Colorsets are usually defined separately from the model. In Design/CPN, colorsets are defined in a Global or Temporary Declaration Node. Colorsets may consist of basic types, like integer, Boolean and string values, and compound types, like tuples, records and lists; even recursive type definitions are allowed.

The colorset of a place is indicated by a label that contains the name of the colorset. A second label may contain an initial set of tokens for that place.

The Declaration Nodes may also contain global functions, constants and variables. These functions and variable names can be used everywhere in the model, like e.g. on the arcs.

3.2.3 Hierarchy

Many tools for modelling CP-nets allow some kind of hierarchy. This increases readability of large systems. A part of the model can be represented by one transition. The specifics of this part are modelled by a separate CP-net on a new page. Each page reveals some of the details of the entire CPN model. This hierarchical process can be repeated several times.

A sub-system is connected to the rest of the model via special places, called Port Places. A Port Place is created for each place that is connected to the hierarchical transition. In Design/CPN, Port Places can be recognized by a small square with a "P", located top-left of the place. A second square indicates in which directions this place is connected to the rest of the system.

Hierarchy allows a compact, global overview of the system, in which components can be divided in sub-components. Each definition of a sub-component will reveal more details about its implementation.

An example of how this hierarchy works is shown below in Figure 6 and Figure 7. In this small example, clients need to take a numbered ticket from a machine. The clients will be helped at the counter as soon as their number is next. The details of how clients are helped at the counter have been left out. The details of how the clients take their ticket is shown on the sub-page in Figure 7. In Figure 6, this hierarchical relation is indicated by the square with "HS", right of the transition Pick_A_Nr_First.

![Figure 6: CPN – Hierarchy example](image-url)
3.2.4 Fusion Places

A new concept in Design/CPN is called Fusion Places. Fusion Places are normal places that are marked to be part of a Fusion Set. Each token that is added to, or removed from, a Fusion Place will be added to, or removed from, all places in the same Fusion Set.

This is the same principle that is used for Port Places. Places in different parts of the model are identified to be the same place, with respect the tokens that are in that place.

Different kinds of Fusion Places are allowed in Design/CPN: global, page and instance. By using the first type, places anywhere in the model can be unified. The last type of Fusion Place only has the scope of that page. This can be useful if one place on a page has a lot of connecting arcs to different locations on the same page; this place can be duplicated thus reducing the length of its connected arcs, while semantically it remains one place.

A CP-net with Fusion Places can be translated to a normal CP-net by defining one common place on the highest hierarchical level of its scope. All former Fusion Places can be identified with that place, by connecting them through the definition of Port Places. The result will be a hierarchical CP-net, which can be translated to a classic CP-net.

Details of this translation are illustrated in the next example. This example models a small village that consists of 4 households, two factories and a power supplier, supplying them with power via a power grid. Figure 8 shows how this example is modelled using a common place at the highest level. Figure 9 shows some details of how the power from the power grid is consumed in a factory and in a household. The power grid is connected to the factory by means of Port Places.

Figure 7: CPN – Details of hierarchical transition Pick_A_Nr_First

Figure 8: CPN – Port place example (Top level)
Figure 9: CPN – Port place example (details of Factory and Household)

A semantically identical model using Fusion Places is shown in Figure 10 and Figure 11. Since we are using Fusion Places, the central place Power_Grid is no longer necessary in Figure 10. Details of the new models of factory and household are shown in Figure 11. The place Power_Grid in Figure 11 is marked with a "FG"-label, which means that this place is a Global Fusion Place. The model of power_supplier is changed in a similar way: the place Power_Grid is no longer a Port Place, but a Global Fusion Place. Since the place Power_Grid is now marked as a Global Fusion Place, this place is identified with each place, in all diagrams of the system, that is named the same. Therefore, each factory, household and power_supplier is connected to one and the same place Power_Grid.

Figure 10: CPN – Fusion example (Top level)

Figure 11: CPN – Fusion example (Factory details)

3.2.5 Code Region

Some transitions may perform very complex calculations to get the results for the outgoing arcs. Sometimes the same result is needed for several different arcs. For this purpose, Design/CPN has added a so-called Code Region to transitions. This Code Region allows you to associate a complex calculation with a transition and its outgoing arcs. For simple transitions, this Code Region will usually be empty.

Besides complex calculations and local function declarations, a Code Region may contain global function calls. These can be used to e.g. update charts or Statistical Variables (section 3.2.6). When the transition fires, this code will be executed exactly once.
3.2.6 Statistical Variables

Statistical Variables are introduced in Design/CPN to allow the user to keep track of certain statistics easily. The variables are declared in a Global or Temporary Declaration Node and have a global scope. After initialisation, they can be updated and evaluated from any Code Region or Arc. Statistical Variables can only be of integer or real type. For the ease of use, several functions have been pre-declared like average and count. A small example of how Statistical Variables are used is shown in Figure 12.

```plaintext
Val isv_Example = USV.CreateInt ();  (* Declaration *)
CPN'IUSV.Init  (isv_Example);       (* Initialisation *)
CPN'IUSV.Upd   (isv_Example, 2);    (* Update, add value 2 *)
CPN'IUSV.Upd   (isv_Example, 3);    (* Update, add value 3 *)
CPN'IUSV.Avg   (isv_Example);       (* Average, which is 2.5 *)
CPN'IUSV.Init  (isv_Example);       (* (Re-) Initialisation *)
CPN'IUSV.Count (isv_Example);       (* Count, which is again 0 *)
```

Figure 12: CPN – Use of Statistical Variables

Statistical Variables are an addition of Design/CPN, which is not common to CP-nets. Just like Fusion Places, however, a Design/CPN model using these Statistical Variables can be translated directly to a standard CP-net. First, a new colorset needs to be defined. Each Statistical Variable becomes a token in a globally accessible place. Figure 13 shows how this, and the use of the pre-defined functions, works in a standard CP-net.

![Diagram](image-url)

Color Int_Stat_Var = list of Int;
var int_val: Int_Stat_Var;
var int_val: Int;
fun isv_init: Int_Stat_Var =
 (* create an empty list *);
fun isv_upd(int_stat_var, new_value) =
 (* add new_value to the list of values and return this *);
fun isv_avg(int_stat_var): Real =
 (* calculate the average of this list of integers *);
fun isv_cnt(int_stat_var): Int =
 (* return the length of the list of values *);

Figure 13: CPN – Statistical Variables in a standard CP-net

3.2.7 Time

Time is an extension of CPN that has been implemented in many CPN tools. In Design/CPN, the concept of time has been implemented as well. How time influences the
dynamics of a CPN model, and how time can be defined in a model with Design/CPN, will be explained in this section.

Each token gets a timestamp. This timestamp determines at what time the token may be used by a transition. The simulator maintains a global clock. A token may not be used for a transition before the clock has reached the value of the token's timestamp. The simulator will increase the clock value only when all transitions are blocked. After incrementing the value of the clock, tokens with a timestamp that equals the clock-value may now enable a transition.

A transition that models an action that takes some time can add a larger timestamp to the tokens it produces. The timestamp is specified on the arc inscription by "@+n", in which n is an integer value. This timestamp is relative to the time at which the transition occurs. A "@+2" timestamp means that the token gets a timestamp 2 units higher than the current clock value. In a CPN-net, this implies that the results of an action are not available for other transitions until 2 time units have passed. Even though a transition adds a timestamp to the output tokens, this transition itself may fire several times before the clock value is incremented.

The syntax and semantics of time in a CPN-net are illustrated by a small elevator example, shown in Figure 14. Opening or closing the doors takes one second. The doors will only start to close 5 seconds after they are opening. The elevator can change floor only when the doors are totally closed. Moving one floor up or down always takes 3 seconds. These 3 seconds are modelled by the timestamp "@+3" of the doors-token. During this time the doors cannot be opened, because the token in the place Door_Closed is not available for 3 time units.

**Figure 14: CPN – Elevator example**

The non-deterministic properties of a CPN-net are preserved with the introduction of time. Transitions will fire non-deterministically as long as possible. Only as soon as all transitions are blocked, the clock of the simulator will be incremented. This new clock value may enable multiple transitions.

### 3.3 Modelling interfaces in a layered system

This section will give two alternative solutions for modelling interfaces in CP-nets. No assumptions about the rest of the system have been made. The advantages and disadvantages, given in this section, have been used to select the best way to construct the CPN model.

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
3.3.1 Colorset definition of an interface

An interface usually consists of several commands. Commands can be modeled by tokens. The invocation of a command can be modeled by the transfer of a token from the caller to the callee. One component can now invoke a command at another component by producing a token in the right input place of that other component. The interface, which is the connecting part between two components, is modeled by places that define that interface and connect the different parts of the CP-net.

For each command, a different place could be used. However, if we want different commands of one interface to be put in a single place (e.g. in a queue), then we need one single colorset definition for the entire interface. Depending on the tool you are using, this may be done in various ways. The following examples show how this can be defined in Design/CPN.

The names of the commands can be encoded in the colorset definition of the interface. Assuming that our interface (Intf1) consists of two commands (CmdA and CmdB), a colorset Intf1 can be defined as follows:

```
Color Intf1 = with CmdA | CmdB; (* enumeration of two constants *)
```

Many commands, however, may also contain some parameters. We need to define a colorset for each different command, such that it can contain all parameters. These colorsets must be combined to a new colorset that unifies all commands. The following piece of code defines new colorsets for the commands and redefines Intf1:

```
Color col_CmdA = int; (* CmdA: one integer parameter *)
Color col_CmdB = product int * int; (* CmdB: two integer parameters *)
Color Intf1 = union cmd_A: col_CmdA + cmd_B: col_CmdB;
```

3.3.2 Alternative 1: implementing a queue

Commands, given through an interface, should be processed in sequential order. There are two possible ways to maintain the order of the commands: either store the commands in a FIFO queue, or allow only one command at a time. This second alternative will be explained in section 3.3.3.

In Design/CPN, a queue can be implemented by a list. The following example shows how this is done. If initially the queue is empty, the initial marking should be the empty list. In this simple example, both interfaces (up and down) have the same type.
Figure 15: Example – Queue

The advantage of this alternative is that a component or layer can send commands through the interface at any time. All commands will be buffered until the other component is ready to receive. Advanced implementations could even use the number of buffered commands for certain decisions.

3.3.3 Alternative 2: using a mirror place

A second way to maintain the order of commands is synchronisation by allowing only one command at a time. This means that a new command can only be send after the previous has been processed. Since there is no way in a CP-net to check emptiness of a place, we use a so-called mirror place. A token in this place will indicate emptiness of the other place.

The example in Figure 16 (left) shows how such a mirror place can be implemented. From the left to the right, the places have the following function: sending commands from layer 1 to layer 2; mirror place for the first place; sending commands from layer 2 to layer 1; mirror place for the third place.

An interface can even be split up in different places. Each command may use a separate place. However, only one mirror place can be used for all the commands of this interface. Figure 16 (right), shows how the communication interface from layer 2 to layer 1 is split up into two places, one for commands of type CmdA and one for commands of type CmdB. The third place on the right is a common mirror place. A token in this place implies that both up_A and up_B are empty. A token (command) can only be placed in one of these two places if both places are empty. Therefore, the interface will, at any time, contain at most one command from layer 2 to layer 1.
Figure 16: Example – Mirror Place
The advantage of this solution is that a component can wait for the other component to be ready to receive and then send an up-to-date command or reply. This way, both push and pull communication can be implemented.

Furthermore, an interface can be split up in different subsets, which can increase the readability. This is shown in the example on the right. Different places can be used for each command or subset of commands. The only disadvantages of splitting up an interface are:

- Changing the interface may require changes on different locations in the model. This can be a problem if the interface is often changed and the used tool does not support such changes easily.
- The interface definition will take a large part of the page. This is merely a trade-off since readability can increase significantly if unrelated commands are stored in different places. On a sub page that implements certain functionality, only the places with related commands will be shown.

3.4 Abstraction
The technical specification of the LLC layer [6] has been used as a basis for the CPN model. This specification contains all necessary details for the protocol to function properly. Modelling all details of this layer would result in a very big model. Most of these details are not necessary for the CPU usage estimation. The key function of the LLC layer has been modelled to serve to predict CPU usage. This results in a carefully constructed model at a certain abstract level with just enough details for the estimates.

The LLC layer consists of three different types of components: LLM (Logical Link Management), LLE (Logical Link Entity) and the Multiplexer. The LLM distributes some management tasks to the different LLEs, while the Multiplexer basically synchronises and ciphers the communication with the layer below. Each LLE provides a logical link connection via which user data can be transferred. Therefore, the LLE is the most important component for our estimates.
Only one LLE will be implemented in the model, since each LLE has the same functionality. One LLE is therefore sufficient to model the LLC layer's key function. Interaction between the different LLEs only influences the external environment in a way that more frames are transferred at the same time. Therefore, the physical communication medium may be busy more often and frames will be received a bit later. This is, just like any external influence, out of our scope.

The tasks of the LLE can functionally be divided in two parts: establishing or maintaining a connection and transferring data. The establishment of a connection consists of the communication of several different commands and replies. This is not the main task for the LLE. Therefore, it is much easier to assume that the connection has been initialised and remains open.

The main task of the LLE is the acknowledged transfer of user data. An LLE also supports the transfer of unacknowledged data, which does not require any special processing. Acknowledged data transfer requires, besides buffering and transmission of frames, sending acknowledgements, re-transmitting lost frames and buffering received frames. This key function of the LLC layer will be modelled in Design/CPN to create resource estimates.

3.5 Design/CPN model

This section starts explaining the CPN model of the LLC layer of the GPRS protocol stack. Chapter 4 will continue by explaining the necessary extensions for the resource estimates.

The explanation of the CPN model will not elaborate on different alternatives to the used solution. Furthermore, the model is not optimal in several aspects. The tool Design/CPN has often caused problems, either critical or just annoying, after changing or extending the model. Therefore, some things are left as they were, even though a better solution was close, and spare parameters have been added to color sets for likely future need.

Because the process of debugging functions in Design/CPN was very difficult, the use of functions has been avoided whenever possible. For example direct access to elements in a buffer could have been implemented by a few functions; in our model we have created a small CP-net that does not require these functions. However, the resulting CP-net is more complex than an equivalent net that uses functions.

In the following subsections the CPN model will be explained top-down. Each subsection will show a part of the model, followed by some explanation. The overview in Figure 17 shows which part of the hierarchical model will be explained in which section.
3.5.1 Top-level structure

The top-level structure of our CPN model, in Figure 18, shows the model of two identical mobile stations. The one on the left will be used during the simulations to send data to the one on the right. The same MS could be used both to send and to receive, but by using two
MSs, the simulation becomes both more realistic and easier to understand. The rounded rectangle top-most left contains a comment with a name for this page.

The centre shows the transition that models the LLC layer, both on the left and on the right. The implementation of the LLC will be given in the next section. Since the implementation for both MSs is identical, this model will be shown only once.

The two interfaces towards layer 3 are shown on the top. Only one place is used for each interface, since the used part of the interface was changing a lot in the beginning. Since the LLC layer has been modelled with only one LLE, the interfaces to the other LLEs and the interface to the GMM have been left out.

Both mobile stations are connected through the RLC layer. This is modelled by the transitions on the bottom. In real life, a Mobile Station will be connected to the SGSN via a Base Station and the BSSGP layer. This is not important for the simulation since the estimates of CPU usage are calculated separately, both for sending data from the first MS and for receiving data at the second MS. This implies that we will only have estimates for CPU usage of the Mobile Station part of the LLC layer. The specifications for both sides of the LLC layer are identical except for some names of interface. Furthermore, the CPU usage is a more limiting factor for a Mobile Station than it is for a SGSN.

All the interfaces have been implemented using mirror places, which have been explained in section 3.3. One reason for this is that the LLC layer needs to decide which frame to (re-)transmit, depending on the last acknowledgement frame it received.
Related colorset definitions:

Color TMessageList = list TMessage;
Color TMessage = Integer; (* stores size of message (not used) *)
(* TMessage models the data that layer 3 wants to transfer *)

Color L3_LLE_interface = TLL_DATA_REQ; (* LLC input from layer 3 *)
Color TLL_DATA_REQ = TMessage; (* Request to transfer message (uplink) *)

Color LLE_L3_interface = union (* LLC output to layer 3 *)
  LL_DATA_IND: TLL_DATA_IND +
  LL_DATA_CNF: TLL_DATA_CNF;
Color TLL_DATA_IND = TMessage; (* Indicates receipt of message (downl) *)
Color TLL_DATA_CNF = Integer; (* Confirms receipt of message (uplink) *)

Color LLE_GRR_interface = TGRR_DATA_REQ; (* LLC output to RLC *)
Color TGRR_DATA_REQ = TLLPDU; (* Request to transfer I or S frame *)

Color GRR_LLE_interface = TGRR_DATA_IND; (* LLC input from RLC *)
Color TGRR_DATA_IND = TLLPDU; (* Indicates receipt of I or S frame *)

(* TLLPDU models a frame that is transferred via the RLC layer, to and *)
(* from a Mobile Station. In our model only I frames (information) and *)
(* S frames (supervisory/acknowledgement) are transferred. The *)
(* information frames contain the data from layer 3 that need to be *)
(* transferred. Other fields are copied from the specification and *)
(* will be explained when used. *)

Color TLLPDU = record
data: TMessage * (* for Iframe, the data from layer 3 *)
  frametype: TLLPDUTYPE * (* all frames, type of frame *)
  MS: Integer * (* debug info, sender *)
  A: Boolean * (* for Iframe, acknowledgement bit *)
  NS: Integer * (* for Iframe, sequence number *)
  NR: Integer * (* for Sframe, expected seq. number *)
  SACK: TackInfoList * (* for Sframe, acknowledgement bitmap *)
  SType: TSType; (* (not used), type of Sframe *)
Color TLLPDUTYPE = with Iframe | Sframe;
Color TackInfoList = list Boolean;
Color TSType = (* not used *);
3.5.2 Definition of LLC-LLE, as a subsystem of the top-level structure

Figure 19: CPN – LLE

In our model, the LLC layer is implemented by exactly one LLE. This LLE, shown in Figure 19, can be functionally divided into two parts: uplink and downlink. In a uni-directional communication of data: uplink is used by the sender; the receiver uses downlink to receive the data. Recalling the top-level structure in Figure 18, the LLC layer of the MS on the left will only use the definition of uplink; the LLC layer of the MS on the right will only use the downlink part of this definition.

The four places at the top of Figure 19 are the input and output places to layer 3. The four places at the bottom are the input and output places to the RLC layer. The two small squares, with an “HS” inside, right of Uplink and Downlink indicate that both transitions are hierarchical transitions. Their details will be explained in one of the following sections.

The uplink will be explained further in section 3.5.3; for downlink, see section 3.5.6. In the middle, there is a place called MS_Id. This place will be initialised with a unique identifier for both mobile stations that are modelled. This initialisation is done by the transition Get_MS_Id and the Global Fusion Place Unique_MS, which are shown on the right. Uplink and downlink use the unique identifier in MS_Id only for statistics and debugging. Since the place MS_Id is connected to uplink and downlink with an input and an output arc, this place will show up on both pages as an I/O Port Place.
3.5.3 Definition of Uplink, as a subsystem of LLE

Figure 20: CPN – Uplink

The uplink side is divided in two parts. The part on the left (section 3.5.4) will handle data requests from layer 3, transmit and retransmit data frames. The part on the right (section 3.5.5) will receive and process acknowledgements from the RLC layer.

The two places Buffers_Info and Send_Buffer are specific for uplink and have not been defined at a higher hierarchical level. These places contain information about the buffered frames. Send_Data will put each data frame, received from layer 3, in this buffer. Each frame will be marked with a status:

- marked for transmission; the frame has not been sent yet
- marked as sent; the frame has been (re-) transmitted, but the LLE does not know if the frame has been received in the peer LLE
- marked for retransmission; the frame has not been received by the peer LLE and needs to be retransmitted
- marked as acknowledged; the peer LLE has received and acknowledged this frame.

The place Buffers_Info maintains a counter for the number of frames per status and one counter for the total. The place Send_Buffer contains the buffer of frames with their status. This buffer has been modelled by two lists, of which the concatenation forms the complete buffer. At which point the buffer is split up will be explained in the next section (3.5.4).
3.5.4 Definition of Send_Data, as a subsystem of Uplink

![Diagram of Send_Data subsystem]

**Figure 21: CPN – Send_Data**

The part of the model, shown in Figure 21, is responsible for transmitting and retransmitting data frames.

At the top-left, DATA_REQ, handles data requests from layer 3. This data will be stored in an information frame. These frames will be saved in the send buffer, marked for transmission. These frames have a field named NS, which should contain a sequence number. The value of the place VS is copied to this field, after which its value is incremented.

The three transitions Skip Frame, Send_I_frame and Retransmit_I_frame will send the first frame that is marked for (re-) transmission as soon as the RLC layer is ready to transmit a
new frame. Skip_Frame will search the buffer for the next frame that needs to be (re-) transmitted. Send_I_frame or Retransmit_I_frame will respectively transmit or retransmit this frame. This has been implemented by splitting up the Send_Buffer into two lists in such a way that the first part of the buffer only contains frames that are either marked as Sent or Acknowledged. The transition Skip_Frame will move frames that are marked as Sent or Acknowledged to the first part of the buffer, until it finds the first frame that needs to be (re-) transmitted. Either Send_I_frame or Retransmit_I_frame sends a command to the RLC layer to (re-) transmit this frame, marks it as Sent and moves it to the first part of the buffer.

The last transition is called Time-out. Delivery of frames via the RLC layer is not completely reliable or it may take too long. Therefore, a timer is set for sent frames. If the LLE does not receive any information about the receipt of that frame in time, it will resend this frame. The lowest indexed frame, which is marked as Sent, will be marked for retransmission by this transition. Section 3.5.10 shows the final sub-model of this transition, which has been changed a little in the final model.

3.5.5 Definition of Receive_Ack, as a subsystem of Uplink

![Diagram of Uplink-Receive](image)

**Figure 22: CPN – Receive_Ack**

This part of the Uplink receives and processes so-called S frames. An S frame contains supervisory information about frames that have been received by the peer. The construction and exact contents of an S frame will be explained in section 3.5.8. Based on the information of the S frame, the LLE can determine for a range of frames which ones have been received and which ones are lost. Each lost frame will be marked for retransmission. Frames that are acknowledged by the S frame will be marked as acknowledged. This has been implemented by the transition Process_ACK.
The transition DATA_CNF will confirm successful delivery of frames by sending a LL_DATA_CNF to layer 3. These commands will be sent to layer 3 in FIFO order, as soon as new frames have been marked as acknowledged.

3.5.6 Definition of Downlink, as a subsystem of LLE

![Diagram of Downlink subsystem]

**Figure 23: CPN – Downlink**

The Downlink side can be divided into two parts. One part, called Receive_Data (section 3.5.7), will receive I frames and send them to layer 3 with the LL_DATA_IND-command. The other part (Send_Ack, section 3.5.8) will send S frames to the sender, containing information about the received I frames.

The place Buffer_Info, in the middle of Figure 23, contains different types of information about the received frames. It stores information about which frames have been received and which have already been indicated to layer 3. Furthermore, it contains a flag that indicates whether an acknowledgement frame needs to be sent. The exact details about how these values are stored and updated are not important.
3.5.7 Definition of Receive_Data, as a subsystem of Downlink

![Diagram showing the flow of data through Receive_Data](image)

**Figure 24: CPN – Receive_Data**

Figure 24 shows how the I frames are received from the RLC layer. At the bottom, the transition Receive_I_frame handles these incoming frames. All received I frames will be stored in Receive_Buffer. The transition on top will send those frames to layer 3, via an LL_DATA_IND, in the same order as their sequence number.

Duplicate I frames will be ignored by the transition Receive_I_frame. Receive_I_frame will update the information in Buffer_Info whenever a new I frame arrives. If the LLE detects an error in the sequence of received frames, or if the sender requests it explicitly by setting the A-bit of the I frame to true, then a flag will be set. This flag indicates to Send_Ack that an acknowledgement frame needs to be sent. Furthermore, the Buffer_Info will contain enough information, about which frames have been received, to construct the acknowledgement frame.
3.5.8 Definition of Send_Ack, as a subsystem of Downlink

Figure 25: CPN – Send_Ack

This transition will only occur when an acknowledgement is needed. This is indicated by a flag in Buffer_Info. After the S frame has been sent, this flag will be set to false again. In the model this is shown by the input and output arc inscriptions to the place Buffers_Info, which check and modify some values of Buffers_Info. In order to be able to distinguish between input and output arc inscription, the input inscription will be written above the output inscription, whenever two arcs are visually overlapping.

The S frame is constructed by assigning values to the fields NR and SACK. The first field will get the sequence number of the first frame that has not been received yet. The second field will indicate of each frame with a higher sequence number, whether or not it has been received.
3.5.9 Definition of RLC layer, as a subsystem of the top-level structure

![Diagram of RLC layer](image)

**Figure 26: CPN – RLC layer**

The last important part of the top-level structure (section 3.5.1), which has not been shown in detail, is the RLC layer. The CP-net that we show in Figure 26 is not an abstract model of the specification of the RLC layer. Our model only simulates an open peer-to-peer connection via the RLC layer. The property that data may be lost during transmission is modelled by the transition Select_New_Random, which will produce a random value. Depending on this value, either Send_To_MS will successfully transmit the frame to the other MS, or Frame_Lost will consume the transmitted frame. The initial token (1°10) in Percentage_Lost configures the system for a 10% data loss. The place RLC_Msg_Hist is a global fusion place, which is used for debugging and statistics.

In order to get more realistic simulations, the model of the RLC layer has been changed a little. These changes will be explained in the next section.

3.5.10 Introduction of the concept of time

In a CP-net, the order in which different transitions fire is non-deterministic. This property is very useful when you try to prove functional correctness. All possible execution sequences can be analysed.

For a performance estimates, however, it is more useful to simulate execution sequences that are likely to occur in a real-life system. In our example, there is not much use in analysing the performance of the system in the case where many frames are sent to the peer before the peer has any chance to send an acknowledgement, since this is very unlikely to happen in an actual system. Another execution sequence that is highly unlikely to happen in a real-life situation is one in which a timeout always occurs before an acknowledgement can be received. Of course, it is quite unlikely that the non-deterministic continues to select the time-out transition for a long time. The only problem is that each enabled transition has an equal chance to fire during the simulations. These chances
cannot be influenced. As a result of this, the transition Time_out fired much too often to resemble a real life situation.

Design/CPN allows you to insert a concept of time in your CPN model. This time concept allows you to influence the order in which transitions occur. The mechanism how the timing works in Design/CPN has been explained already in section 3.2.7.

Often, it is rather clear where to add time to the model. However, the influence of those changes on the behaviour of the model is not always that clear. Every timed transition limits the amount of parallelism in the model. This can be necessary to rule out undesirable behaviour, but this more restricted model may disallow some real-life situations. Running a simulation to get CPU estimates may therefore give a wrong prediction about the performance of the real system, because only a small part of the possible behaviour is modelled by the CP-net. Therefore, it is important to take a close look at which execution sequences are still possible and which sequences are no longer allowed by the model.

We will continue this section with an explanation about how our CPN model has been extended with time. This extension should still allow likely real-life situations, but rule out the very unlikely situations. This will result in a more accurate estimate of CPU usage of the real system.

A new implementation of the transition Time_Out (LLE-Uplink-Send_Data) is shown in Figure 27. Some concept of time has been added here, because otherwise this time-out will occur too often, which would ruin our estimates. Connecting the transition to the dummy place Timeout and adding some time to it ensures that this transition will not occur randomly. If this is the only location where time is inserted in the model, then this transition will only occur when all other transitions are blocked. In a real life situation, the time-out may also occur when a frame is very delayed; in our CPN model, that is just modelled as a frame that is lost.

Figure 27: CPN – Time_Out, with time

The second part of the model that performs undesirable behaviour is the RLC layer in which the frames are communicated between the Mobile Stations. The uplink side may send all frames, before the peer even has a chance to respond with an acknowledgement. We want to prevent this to happen, since it will not happen in real life. The amount of CPU used for these unrealistic runs cannot be used to draw any conclusions about the CPU usage of an operational implementation.
The undesirable behaviour can be prevented easily by adding time in the RLC layer. A new model of the RLC layer is given below in Figure 28. The transmission of a frame will now take two time units, while a next frame may only be transmitted after one, two or three time units. This way, the uplink can send up to four frames before it receives an acknowledgement that is transmitted by the peer right after the receipt of the first frame.

Figure 28: CPN – RLC layer, with time

During the simulation, time will be incremented more often. Hence the transition Time_Out needs a higher timestamp. Its value will be set to 100 and will be reset after each transmission of an I frame. This way, the transition Time_Out will, again, only occur when all other transitions are blocked.
4. ESTIMATION OF CPU USAGE

This chapter describes how estimates for CPU usage can be generated and how the results can be interpreted. Section 4.1 will first explain why a simulation of the CP-net is preferred above other means to analyse the performance. Section 4.2 shows the differences between an absolute estimate and a relative comparison. The latter one has been used for this case study. Section 4.3 explains how the results can be quantified, while the way the results are stored is explained in section 4.4. Accuracy of the results is discussed in section 4.5. Section 4.6 shows how the measurements have been implemented in the different versions of the model. The results of the simulations are given in chapter 5.

4.1 Simulation

There are two well-known methods to analyse a CP-net: simulation and state-space-exploration. A simulation can give a quick idea of how the system behaves, while state-space-exploration identifies all reachable states. To prove functional correctness of a system, the latter method is very useful. Both methods can be used to generate an estimate of CPU usage.

In order to be able to use the method of state-space-exploration for analysis of CPU usage, it is necessary to store the estimate of CPU usage in a place in the CPN model. The constructed occurrence graph will then show which values can be reached for this place storing the CPU usage. It gives an exact upper and lower bound for the CPU usage of a scenario. However, this method does not show the likelihood of these extremes. There is no way to see, from the occurrence graph, the amount of CPU that will be used on average.

A simulation is much more suitable for an analysis of non-functional properties like CPU usage. Due to the non-determinism of the model, the measured CPU usage may change for different runs of the same scenario. Less likely executions of the system are less likely to occur during a simulation and different executions that result in the same amount of CPU usage will show with this method as well. A couple of simulations give a quick overview of the average, the bounds and the distribution of your estimates. More information about possible CPU usage can be obtained by running the simulation several times more.

Generating estimates of CPU usage by simulation requires only some small additions to the model. At certain times during the simulation, values that are necessary to calculate the CPU usage need to be copied to a separate place. Depending on the used tools, these additions can be kept completely separate from the rest of the model.

4.2 Comparison between different implementations

Once we have a model, several extensions can be made to generate estimates of CPU usage. Depending on the level of details in the CPN model, those estimates can be either coarse-grained or fine-grained. The results can be used to predict the CPU usage of the system that is modelled by this CP-net.

The estimates can become very interesting when comparing two different possible implementations. This allows a programmer to evaluate the non-functional properties of two alternative implementations. Two different models, representing both implementations, are.
needed. These models should have a comparable level of detail, such that the results can easily be compared. Based on the estimates, the best alternative can be chosen without implementing the system and testing it in a real-life environment.

The performance of a system can largely depend on its environment or its use. Simulations may indicate that one implementation performs very well on certain scenarios, while it performs much worse on other scenarios, compared to the second implementation. Depending on the expected use of the system, one implementation will be preferred above the other.

If two different implementations of the same system are compared, it is not always necessary to measure all CPU usage in the model. A large part of both implementations may be exactly the same and the rest of the implementation does not influence the CPU usage of that part. In this case, estimates can be generated for the differences in the models only. These estimates will show how the CPU usage of both implementations differs. The relevance of this difference depends largely on the total amount of CPU that is used; this includes CPU usage of the rest of the layer, which has been left out, and CPU usage of the other GPRS layers and other applications running on the same processor.

Our CPN model is based on the specification of the LLC layer of the GPRS protocol stack. This specification still allows some small variations in implementation. Our model has been changed both on the uplink and on the downlink side. Both changes are independent, allowed by the specification and model a different implementation.

The difference between the first and the second implementation variant is at the uplink side. The first implementation uses an additional variable to keep track of frames that should be retransmitted next. The second implementation will not use this additional variable and thus save memory usage. Of course, this second implementation requires more CPU because it needs to search for the next frame to be retransmitted.

The differences between the other two implementations are only at the downlink side. Implementation 3 updates an acknowledgement frame whenever new information has been received. This frame will be transmitted as soon as an acknowledgement is required. Implementation 4 will construct the acknowledgement frame only when such a frame needs to be sent.

Due to time constrains of the project, more complex changes in implementation have not been analysed. However, the conclusions that can be drawn from the current estimates give a fair indication of the kind of conclusions that can be drawn, using this method of resource prediction. The models of these four different implementations will be explained in more detail in section 4.6.

4.3 Quantifying the results

In our case study we are estimating CPU usage in a part of the GPRS protocol stack. CPU usage will be measured in CPU clock cycles. The amount of time that the processor needs for the communication of a certain amount of data can be calculated by dividing the estimates by the number of clock cycles per second. The total time that is needed for the transfer of this amount of data is also influenced by the physical communication times and by other processes running on the same processor.
4.3.1 Counting CPU clock cycles

The simplest way to get an estimate of CPU usage is to have one aggregate counter. Every time an action is executed, this counter is incremented with a value that represents the amount of CPU cycles needed for that action.

More details can be added to these results by using several counters. Each counter counts the number of clock cycles needed for a category of actions. In our example, a distinction between CPU needed for the uplink could be separated from CPU needed for downlink. The uplink counter could again be divided in a counter for all actions related to encryption and first transmission and a counter for all retransmission related CPU usage.

4.3.2 Counting basic actions

Instead of counting the number of CPU clock cycles directly, it is possible as well to count the number of executions of individual actions. Multiplying these numbers by the number of clock cycles needed for such an action, we get an estimate of the required number of CPU clock cycles.

The advantage of this approach is that knowledge about the amount of CPU usage for different actions is not needed in advance. The counted actions can be complex calculations. Depending on the platform, these actions may take different amounts of clock cycles. So, the same results can be used to draw conclusions, which may be different for different platforms.

Furthermore, different platform specifications may allow different implementations of these complex calculations. More efficient algorithms can be used if the platform has more available memory or a different instruction set. The results can even be used to analyse the effects of increasing available memory on a platform.

Though it is easiest to use one counter for each different action, different actions can share a counter if their CPU usage is more or less the same. In order to analyse the difference in CPU usage of e.g. uplink and downlink activities, even more than one counter can be used for identical actions at different locations of the model.

Since the number of clock cycles required for each action was still unknown during this project, the second method of estimating CPU usage will be used in our CPN model. One counter will be used for each action that is included in the estimate of CPU usage.

4.4 Storing the results during a simulation

A simulation consists of several steps. During these steps, all measurements should be stored somehow, until the simulation ends. In Design/CPN there are three different ways to store the values of your measurement during a simulation:

- Create additional CPN places
- Add values to tokens
- Use statistical variables
All three methods will be explained below. Different methods may be combined to get better results. Some advantages and disadvantages of the different methods will be shown as well.

Statistical variables have been used in our model to store the actual estimates. Section 4.5 will give some details about how this method has been applied and at which locations in the model values for the measurements will be updated.

4.4.1 Additional places

Additional places can easily be added to a CPN model. Depending on the colorset of that place, a total sum or a list containing a history can be stored. For example, the place RLC_Msg_Hist in Figure 28 contains a list of string that shows in which order the frames have been transmitted. By declaring this place on a higher hierarchical level, different sub-pages can update a stored measurement. In Design/CPN, the concept of fusion places can also be applied, which is a short hand for the previous change. Declaring the place RLC_Msg_Hist in Figure 28 as a global fusion place will show the history of frames in both directions. If the place is declared as a normal place, then the transmission history for both directions will be stored separately.

This method is quite clear and easy. All kinds of values and results can be stored on different hierarchical levels. With this method, it should be possible to keep the changes small and local to the location where you perform the measurements. The editor may help by supporting different colours, which creates a visual distinction from the functional model.

4.4.2 Extended tokens

In CP-Nets, tokens may have different values. Besides the values the tokens need for the functional correctness, values that are used for storing measurements can be added to some tokens. First, of course, the colorset of this token needs to be extended. After that, all transitions, which change or use the value of the related token, probably need some changes to preserve the value of the measurements. Finally, one or more transitions that influence the measurements need to update those values of the token.

In our case, the token for information and confirmation frames can easily be used for storing measurements. Frames are sent back and forth. Each action that the LLC layer in our CPN model performs is related to a frame. Therefore, all CPU usage can be added to a frame.

The advantage of an approach like this is that all CPU usage related to one frame is saved in a natural way. Different parts of the model, which perform an action related to that frame, have access to the stored value, without the need to connect all these transitions to one single place. A problem that arises here, in our case study, is the fact that some frames may get lost. The CPU measurements of these frames need to be stored somewhere else.

4.4.3 Statistical variables

Statistical variables are an addition of Design/CPN, which is not common to CP-nets. However, any Design/CPN model that uses these statistical variables can be translated directly to a standard CP-net. A statistical variable is globally accessible and maintains a
history of integer or real values. Several pre-defined functions simplify the process of e.g. adding a value to the history or calculating the average.

The biggest advantage of using statistical variables is that no big changes to the model are required: no changes in colorset definitions and no additional places. Only a simple call to the update function of the correct statistical variable needs to be added to the code region of a transition. These statements can safely be added or removed from the model without requiring changes anywhere else or affecting the appearance or behaviour of the model.

4.5 Obtaining accurate results

Due to some non-deterministic behaviour, a simulation will not always give the same results. Different frames may get lost, or the RLC layer may use less or more time for the communication. This results in a different sequence of frames that are communicated.

We define a scenario to be a certain number of messages to be communicated between the two MSs. The simulations need to be run several times to get more reliable measurements of a scenario. The maximum and standard deviation could also be very interesting values to analyse for several runs of a scenario.

The Design/CPN simulator has options to go back to the initial state and start the simulation again. The problem is that you need to store all the results of your measurements somewhere. Figure 29 shows an extension to our CPN model. This extension will run one scenario for several times and collect the results of each run separately.
Figure 29: CPN – Run multiple scenarios

The top of Figure 29 shows a check that a confirmation has been sent to layer 3 for each message that needed to be transmitted. This condition will only hold when all other transitions have stopped firing. Now, all statistics need to be collected and stored, which is implemented in the code region of Collect_Statistics. The next step is to bring both MSs back in their initial states. When both MSs are back in their initial state, there will be a token in the place Start. The transition Restart_Scenarios will reset statistical variables, increment the Scenario_Count and copy the list of messages to LLC_Left_Input. This will restart the same scenario. Once Scenario_Count equals the number of scenarios that should be run, this process stops.
4.6 Implementation using statistical variables

This section will explain how estimates of CPU usage can be extracted from the CPN model. The first section explains which names have been used for the statistical variables and what those variables will count. The following two sections explain how the estimates of different runs of a scenario are combined and how the results of different types of scenarios can be compared. The last three sections explain four different implementations and provide parts of the model that have been changed to model each difference in implementation. Each model gives an estimate of CPU usage of one of the possible implementations. The reason why these different models are created has already been explained in section 4.2.

4.6.1 Used names for counters

The following variable names have been used to store data about the simulation: ScenarioCount, MessageCount, MessageSend, MessageResend, MessageAck, CpuTimeout, CpuReadAEIt, CpuUpdAck and CpuCreateAck. The first two variables count the progress of the simulation. The next four variables will count certain events. The last three variables are used for counting actions related to CPU usage. These variables apply to different types of implementations, of which the differences will be explained in more detail in sections 4.6.4, 4.6.5 and 4.6.6. Each variable will now briefly be explained.

ScenarioCount: As explained in section 4.3, each simulation will execute an identical scenario several times, which will result in more accurate measurements. This variable will count the number of scenarios that have been run during the simulation. This is the only variable that contains a total sum; all other variables will contain an average for all scenarios.

MessageCount: This variable contains the number of different messages that are transferred from one side to the other.

MessageSend: The number of times a message is sent for the first time is counted with this variable. In effect, this will thus always contain the same number as MessageCount.

MessageResend: In this variable, the number of times a frame is retransmitted is counted.

MessageAck: This variable counts the number of acknowledgement frames sent from the downlink side, back to the sender.

CpuTimeout: The variable CpuTimeout counts the number of times a timeout occurs. This variable is therefore an indication of the amount of CPU that is needed to handle timeouts.

CpuReadAEIt: This variable is used in implementations one and two. It counts the number of times an element of the send buffer is indexed. When the uplink wants to send a frame, it will use a linear search to find the next frame to be (re-) transmitted.

CpuUpdAck: Every time an information frame is received, the information needed for an acknowledgement frame will be updated. At the moment an acknowledgement frame will be sent to the sender, this information can simply be copied. The amount of CPU usage depends on the size of data that needs to be updated. This variable will therefore count the size of the acknowledgement frame every time it needs to be updated.
CpuCreateAck: In the fourth implementation, an acknowledgement frame is only constructed when it is needed. Therefore the receive-buffer needs to be read completely to see which frames have or have not been received. This action will take more time when the acknowledgement frame contains more info about the information frames. Therefore, this variable will count the number of frames that are, either positively or negatively, acknowledged by any of the acknowledgement frames.

4.6.2 Combination and storage of the results

Two sets of variables are declared and used in our model to store the measurements. The first set is used for the measurement during each run of a scenario. At the end of each run, these variables contain the measured CPU usage for a complete scenario. At the beginning of each run of a scenario, these variables need to be reset.

The second set of statistical variables will be used to accumulate the measurements of all runs. At the end of each run, the data that has been accumulated in the first set will be added to this second set of variables. Most values will be divided by the number of messages before they are stored; the reason for this will be explained in the following section. When the simulation ends, the second set of statistical variables contains the measurements of all runs. The average CPU usage per scenario can be determined by using the average function on these variables.

4.6.3 Comparing different scenarios

In practice, the LLC will get different quantities of messages that need to be transferred at approximately the same time. Therefore, it would be interesting to see how the performance of the different implementations changes for different number of messages. This would show how stable the results are, as well. If CPU usage changes a lot for small changes in the number of messages, then the results of the measurements should be used carefully. A small change in the use of the LLC layer would already lead to a big change in CPU usage.

The absolute amount of CPU usage will increase for each additional message to be transferred. If we divide each variable by the number of messages, we get an indication of how efficient each message is transferred. Different scenarios can now easily be compared with respect to how efficient the messages are communicated.

The two variables ScenarioCount and MessageCount will not be divided by any value, since they count the total number of runs of the scenario and the number of messages per scenario respectively.

4.6.4 Implementation 1 & 2

The difference between implementations 1 and 2 is at the uplink side. When a confirmation frame arrives, Implementation 1 will store additional information about which frame needs to be retransmitted next. The second implementation does not store additional information and will just start looking from the beginning for the first frame to retransmit. At the cost of higher memory usage, the first implementation will reduce CPU usage because fewer lookups are required.
For both implementations, the number of array-element lookups is measured. Section 3.5.4 has shown the part of the CPN model that sends frames. The three transitions Skip_Frame, Send_I_Frame and Retransmit_I_Frame all perform an array-element lookup. Figure 30, which contains the detailed model for Send_I_Frame, shows how the number of these lookups is counted. The last line of the Code Region shows an update of the Statistical Variable CpuReadAEIt. A similar Code Region is added to the details of the other two transitions that perform these lookups, Skip_Frame and Retransmit_I_Frame. Applying the Count function to this variable, at the end of the execution of a scenario, will give the total number executions of one of these three transitions, and thus the total number of lookups.

Figure 30: CPN – Measurement of the execution of an array element lookup

The difference between the two implementations is located in the part of the model, where acknowledgements are processed by the uplink side; this part of the model has been explained earlier in section 3.5.5. In the first implementation, the transition Process_Ack, which receives acknowledgement frames and updates the buffer-information, will set an indicator to the first frame to retransmit. In the second implementation, the indicator will be set to the first frame in the buffer. Therefore, the second implementation will start its search, for frames to be retransmitted, at the first element in the buffer.

4.6.5 Implementation 3

The difference between implementation 3 and 4 is at the downlink side. The downlink receives information frames and sometimes needs to reply by sending an acknowledgement frame. Implementation 3 will construct and update the acknowledgement frame whenever it receives a new information frame. The amount of CPU required for this update depends on how big the resulting acknowledgement frame has to be. Figure 31 shows how, upon receipt of an I frame, the statistical variable is updated according to the size of the constructed acknowledgement frame.
4.6.6 Implementation 4

This implementation has only a small difference compared to implementation 3. Instead of updating an acknowledgement frame whenever a new information frame arrives, this implementation will only construct an S frame the moment it needs to be sent. A acknowledgement frame needs to be sent if it is requested explicitly by the sender or if an error in the transmission sequence has been detected.

The construction of a complete S frame will take more time than the small updates, but this action needs to be performed less often. The amount of CPU that is used for the construction of this acknowledgement frame depends on the range of information frames that are acknowledged by this S frame. Figure 32 shows the implementation in Design/CPN, where the CpuCreateAck variable is updated.
Figure 32: CPN – Measurement of the construction of an acknowledgement frame
5. RESULTS

This chapter describes the results of our simulations. The previous chapter has explained how estimates of CPU usage are generated through simulations. The result of these simulations will be given in section 5.1. In sections 5.2 and 5.3, the four different implementations will be compared in pairs, based on these results. These comparisons will not be followed by a conclusion about which implementation is best; this has never been a goal of the research project. The goal of this project is to discover whether it is possible to construct a model such that CPU usage of an implementation can be estimated and be compared to that of other implementations. The results will show that it is possible to generate these estimates and how these estimates can be used to select a best implementation.

5.1 Tables of results

The results of the simulations are shown below in two tables. Table 1 contains results for each implementation, shown in the columns that are labelled from I to IV. A scenario of 14 messages has been repeated for 50 times to get the estimates for CPU usage. Section 4.6 has explained how these results have been extracted from the model. The exact meaning of the values of each variable in this table has been explained already in sections 4.6.1 and 4.6.3. A short summary of this description is given in the last column of the table. If no value has been measured for a variable, then this is denoted by a “-”; values that are not relevant for the comparison of the two implementations are shown between brackets.

Because of the exploratory character of this research, we will not worry about reliability intervals. In order to still get a fair idea about the reliability of the documented values, some simulations have been repeated with a different start value for the random generator. It showed that the results of a second run differ less than 10% from the documented values.

<table>
<thead>
<tr>
<th>Variable</th>
<th>I</th>
<th>II</th>
<th>III</th>
<th>IV</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScenarioCount</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>Executions of scenario</td>
</tr>
<tr>
<td>MessageCount</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>Messages per scenario</td>
</tr>
<tr>
<td>MessageSend</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
<td>Send actions per message</td>
</tr>
<tr>
<td>MessageResend</td>
<td>0.112</td>
<td>0.136</td>
<td>0.117</td>
<td>0.128</td>
<td>Resend actions per message</td>
</tr>
<tr>
<td>MessageAck</td>
<td>0.256</td>
<td>0.253</td>
<td>0.255</td>
<td>0.294</td>
<td>Acknowledgements per message</td>
</tr>
<tr>
<td>CpuTimeout</td>
<td>0.020</td>
<td>0.033</td>
<td>0.027</td>
<td>0.023</td>
<td>Time out occurrences per message</td>
</tr>
<tr>
<td>CpuReadAElt</td>
<td>1.577</td>
<td>2.672</td>
<td>(1.588)</td>
<td>(1.632)</td>
<td>Buffer lookup actions per message</td>
</tr>
<tr>
<td>CpuUpdAck</td>
<td>-</td>
<td>-</td>
<td>1.857</td>
<td>0</td>
<td>Size of updates to ack-frame per msg</td>
</tr>
<tr>
<td>CpuCreateAck</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0.631</td>
<td>Size of constructed ack-frame per msg</td>
</tr>
</tbody>
</table>

Table 1: Results (1)

For each implementation, the simulation has been repeated for different scenarios. These scenarios consist of 1, 3, 14 and 30 messages respectively. Each scenario has been run for 50 times and the average results are shown in Table 2. For each implementation, only

---

1 Number of times the scenario has been run, to get more reliable results.
2 Number of messages that need to be sent per scenario.
3 Number of executions of related action, per message.
4 Number of executions of related action, multiplied by the size of the acknowledgement frame, per message.
one variable is important for our evaluation. Therefore, the results for all the other variables have been left out in Table 2.

<table>
<thead>
<tr>
<th>Impl</th>
<th>Variable</th>
<th>Nr of messages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>CpuReadAElt(^5)</td>
<td>1.20</td>
</tr>
<tr>
<td>II</td>
<td>CpuReadAElt(^5)</td>
<td>2.18</td>
</tr>
<tr>
<td>III</td>
<td>CpuUpdAck(^4)</td>
<td>1.18</td>
</tr>
<tr>
<td>IV</td>
<td>CpuCreateAck(^4)</td>
<td>1.17</td>
</tr>
</tbody>
</table>

Table 2: Results (2)

5.2 Comparison between implementation 1 and 2

The differences between implementations 1 and 2 are explained in section 4.6.4. The second implementation saves memory usage, but has a higher CPU usage. Section 4.6.4 also contains information on how the CPU usage is modelled in both CP-nets. The results of the simulations are shown in Table 3. The results in Table 3 are copied from Table 1 in section 5.1.

The relative increase of CPU usage, with respect to the first implementation, varies only a little for different scenarios. These percentages are shown in Table 3. Of course, CPU usage of the complete system will increase by much less than 70%. These percentages are only related to this single action; all other actions like encrypting data will still require the same amount of CPU. The percentages given in Table 3 are given to illustrate how CPU usage increases for different scenarios: the increase in CPU usage is getting smaller when more messages are communicated at the same time.

Figure 33 shows a chart of the values of CpuReadAElt for both implementations, as they are given in Table 3.

<table>
<thead>
<tr>
<th>Nr of messages</th>
<th>1</th>
<th>3</th>
<th>14</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpuReadAElt(^5) (Impl I)</td>
<td>1.20</td>
<td>1.69</td>
<td>1.58</td>
<td>1.64</td>
</tr>
<tr>
<td>CpuReadAElt(^5) (Impl II)</td>
<td>2.18</td>
<td>2.92</td>
<td>2.67</td>
<td>2.61</td>
</tr>
<tr>
<td>Ratio CPU usage, I : II</td>
<td>55%</td>
<td>58%</td>
<td>59%</td>
<td>63%</td>
</tr>
</tbody>
</table>

Table 3: Results implementations 1 and 2

\(^5\) Number of executions of this action, per message.

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets
Figure 33: Chart – Estimates for implementation 1 and 2

A choice between these two implementations is based on a trade-off between memory and CPU usage. The exact increase of CPU usage can be calculated by combining the data in Table 3 with knowledge about common communication traffic and the number CPU clock cycles needed for an array element lookup on the platform of choice. Based on the differences in CPU usage and memory usage, a preference for one of the two implementations can be motivated by numbers.

Once CPU usage of the other parts of the LLC layer and all other layers is known, the figures will show which implementations are feasible on a selected platform. The other way around, minimal hardware requirements can be extracted from the estimates of resource usage.

Without knowledge of total CPU and memory usage, a cheapest solution can already be selected. The differences in CPU and memory usage can be multiplied by the price of CPU and memory respectively. On a platform like a mobile phone, this comparison is not completely fair. Memory and CPU is selected from a small set of components. A small increase in memory usage may require the use of a much bigger chip. Still the cheapest combination can be selected.

Some other factors may influence the choice of components as well, like for example power consumption. A choice for a faster CPU instead of a bigger memory chip could allow some other applications, like e.g. a more advanced game, to be run on the mobile phone as well. In this example, the investment of a faster CPU can be earned back much easier than a bigger memory chip.

5.3 Comparison between implementation 3 and 4

The difference between implementation 3 and 4 is explained in sections 4.6.5 and 4.6.6. These sections also contain information on how our estimates can be obtained from both CP-nets.

It is quite hard to give a first estimate about the efficiency of both operations that have been measured. Some real measurements on a test platform should provide the necessary data. Once these data have been collected, the figures of Table 4 can be used. Again, the
combination of the counted actions in our CPN model and the actual CPU usage for these actions give a clear indication of the most efficient solution.

Figure 34 shows a chart of the values of both variables that are given in Table 4. Note that two different actions are plotted in this graph. Though a break-even point probably exists, implementation 4 is not the most efficient for each scenario that has been plotted. The graph does show how different scenarios have a different influence on the CPU usage of both implementations.

<table>
<thead>
<tr>
<th>Nr of messages</th>
<th>1</th>
<th>3</th>
<th>14</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>CpuUpdAck⁶ (Impl III)</td>
<td>1.18</td>
<td>1.27</td>
<td>1.86</td>
<td>2.83</td>
</tr>
<tr>
<td>CpuCreateAck⁶ (Impl IV)</td>
<td>1.17</td>
<td>0.89</td>
<td>0.63</td>
<td>0.51</td>
</tr>
<tr>
<td>Ratio counters, III : IV</td>
<td>101%</td>
<td>142%</td>
<td>295%</td>
<td>555%</td>
</tr>
</tbody>
</table>

Table 4: Results implementations 3 and 4

Figure 34: Chart – Estimates for implementation 3 and 4

Table 4 shows a big difference in the amount of CPU usage that can be saved for different scenarios. This means that the actual use of the LLC layer is of a big influence on which implementation performs best.

Even though the exact amount of CPU usage is unknown, implementation 3 is probably using more CPU than implementation 4. During a transmission of a series of frames, not all frames need to be acknowledged independently in the bitmap. The successful receipt of a series of frames can be acknowledged by sending the sequence number of the last frame. Implementation 4 needs to do at most the same amount of work as implementation 3. On the other hand, implementation 3 does not need to retrieve any buffered frame from the memory.

Implementation 3 probably uses more CPU, but this implementation may still be preferred. Implementation 3 performs a part of the total CPU usage upon receipt of each frame, and thus spreads CPU usage over different moments of time. Implementation 4 requires all its CPU power at the moment when an acknowledgement frame needs to be sent. The

---

⁶ Number of executions of this action, multiplied by the size of the acknowledgement frame, per message.
response time for this implementation is a little longer, since the entire acknowledgement frame needs to be constructed.

The results in Table 4 show that the CPU usage of a system is dependent on the usage of that system. During our simulations we kept the reliability and speed of underlying layer constant; changing the model of the RLC layer in between simulations would have influenced the results as well. Running the simulation for different use cases and with different models for the environment, this method can be used to analyse the relationship between CPU usage and the environment and use of a system for each different implementation.

This method proves to be useful for selecting an implementation that performs best on a selected set of scenarios. Different criteria can be used to make this selection. An implementation that uses little CPU when executing some common scenarios will provide the user with a system that performs very well most of the time. Power consumption is likely to be low as well, since less CPU is used during normal use of the system. An implementation that uses relatively little CPU for more extreme scenarios results in a system with a steady performance. Thus, even if a less common scenario is executed, the user will not notice a big change in performance of the system.
6. CONCLUSION

This document describes a method for modelling CPU usage of a system using Coloured Petri Nets. The estimates generated from this model can provide useful information about the system. The estimates of this document will not be used for conclusions about which implementation is better. These estimates are only used to show that such estimates can be generated and to illustrate how the CPU usage of two implementations can be compared. The possible (financial) advantages of this knowledge, however, will not always outweigh the costs to generate these estimates. Unless Coloured Petri Nets are already used to model the system, this method should not be integrated in every development cycle, but only be applied on a project basis. And even if a CPN model has been constructed already, that model may contain too many details to generate CPU estimates easily.

Modelling CPU usage can provide significant benefits in three kinds of situations. Firstly, it can be useful for the analysis of a running system. Secondly, it can be very useful for the design of a new system. It can provide useful insights in the expected CPU usage. Finally, the method can be used to discover the better of two possible implementations. The following three paragraphs will discuss what useful information the method developed can provide, for each of these three situations. A project manager should decide whether this information provides sufficient benefits to the project and reserve a part of the project's resources for this CPU modelling.

The implementation of a system may have been completed already. Its CPU usage, however, can deviate from the expected usage. An estimate of the CPU usage can be generated from a model of the current implementation with the method that is discussed in this document. This method will identify how often each part of the system is executed. This will show of which function the number of executions deviates from expected. It can also show how much CPU usage will change if this function is implemented more efficiently.

The described method can be very useful for a feasibility study, as well. The CPU estimates, generated from an abstract model of the entire or only the core part of the new system, will show whether the implementation meets the hardware specifications. The other way around, the CPU estimates can also be used to determine hardware requirements of the new system.

The third kind of situation is one in which two or more possible implementations are under consideration. Possibly, one of these two is an existing implementation, while a new implementation is considered because e.g. a more enhanced encryption algorithm is available or because a higher throughput is desired. In this case, an analysis of the change in CPU usage of the new implementation is necessary to see whether the CPU savings are worth reprogramming, the new implementation fits the hardware specifications, or whether the higher throughput outweighs the increase in CPU usage.

Two alternative implementation designs can be compared with each other. The described method gives more information than just which implementation has the lowest CPU usage. The method can be used quantify the trade-off between memory and CPU, for two implementations that are optimised for memory and CPU respectively after the difference in memory usage has been determined. CPU estimates can be generated for different scenarios. For different implementations, CPU usage may be influenced differently by...
these scenarios. This method will quantify how much better or worse one implementation performs, compared to the other, on each scenario or set of scenarios.

The method does not generate a total sum of all CPU usage, but it counts the number of executions of actions, which are various pieces of code. These numbers are generated through simulations. There are a few advantages to the counting of actions in comparison to counting the number of CPU clock cycles directly. All simulations can be executed without prior knowledge about the CPU usage of each action; those values can be generated in parallel by another department. A different value can be used for each platform, without the need for new simulations. Depending on the platform, a different implementation may be chosen, based on the same simulations. Another advantage is that the counted actions can be of arbitrary level of detail. Therefore, this method can be used for both high-level specifications and small detailed implementations. If CPU usage of an action changes, a more efficient subroutine or component becomes available, then the effects on the CPU usage can be evaluated directly. This may change the choice of implementation, without investing any additional research.

Concluding, this method can provide a lot of useful information about the CPU usage of a system, both during the design and during the re-design of that system. It can be used to check feasibility or to quantify the trade-off between e.g. memory and CPU usage. Developers can base their choice of implementation on the results of the model, and motivate their choice towards managers by presenting the estimated benefits for the company.
7. REFERENCES


[5] 3GPP TS 03.64: "Digital cellular telecommunications system (Phase 2+); Overall description of the General Packet Radio Service (GPRS) Radio interface; Stage 2".

[6] 3GPP TS 04.64: "Digital cellular telecommunications system (Phase 2+); General Packet Radio Service (GPRS); Logical Link Control (LLC)".

[7] 3GPP TS 08.64: "GPRS support node – BSS protocol (SGSN_BSS) interface; BSSGP layer 3".

[8] 3GPP TS 04.60: "General Packet Radio Service (GPRS); Mobile Station (MS) – Base Station System (BSS) interface; Radio Link Control/ Medium Access Control (RLC/MAC) protocol".

Modelling CPU usage of a GPRS protocol layer using Coloured Petri Nets