An Overload Protection Mechanism in Radio Base Station (ORBS) Analysis, Simulation, and Solutions

H A B I B A L I V I R J I

Master of Science Thesis

Stockholm, Sweden 2005

IMIT/LECS-2005-84

An Overload Protection Mechanism in Radio Base Station (ORBS) Analysis, Simulation, and Solution

By:

Habib Ali Virji

(habibvirji@yahoo.com

November 2005-11-22

Industry Advisor at Ericsson Radio AB: Tony Malmstrom

Supervisor in KTH: Vladamir Vlassov

Department of Microelectronics and Information Technology

Royal Institute of Technology, Stockholm, Sweden.

Acknowledgement

I would like to express my gratitude to Mr Tony Malmström for giving me the opportunity to do this thesis work in Ericsson Radio Systems AB, Kista, Sweden.

I am thankful to Mr Olle Rosendahl for his feedback and explanation of concepts. I am grateful to Ludomir Rewo, Fredrick Lunell, Daniel Tekle, and Vitali Fridland for their support in this thesis work.

I would also like to thank my mentor Dr Ahmed Hemani for his views and help during the thesis work. I am thankful to Mr. Vladimir Vlassov for supervising me in this thesis and making it acceptable to KTH standards.

Abstract

In wireless communication, Radio Base Station (RBS) establishes a physical link over an air interface with the Mobile Equipment (ME). The functionality provided by the RBS is either related to the traffic generated between RBS and ME or execution of activities related to Operations and Maintenance (O&M).

In RBS resources management and process scheduling on processor is done through Real Time Operating System (RTOS). In Ericsson's RBS it uses OSE as its RTOS that uses pre-emptive scheduling to schedule processes on processor. RoseRT is used as an IDE, which also provides virtual machine for executing automatic generated code and developer code, which is usually written in C++.

The workload created by RBS tasks sometime exceed more than the processor capacity and lead to a system restart, such situation is known as an overload situation. To control system resources and processor utilization under situation of overload, an overload protection mechanism is implemented in RBS.

Overload protection main aim is to allow continuation of services for already connected users and prevent RBS from restarting. The main focus of this thesis is to study RBS present overload protection mechanism and suggest other possible ways for effectively implementing overload protection mechanism.

To understand the working of overload protection, a load simulator is built that imaginarily represents load created by traffic application and works along with the present overload control implementation. The simulation program is implemented in C++ with Rational RoseRT as its IDE. The reason for developing simulation program is that overload protection works in a very complex environment and to understand it there is a need to observe overload protection working along with real system. Then based on the understanding of the overload protection different ways of controlling overload protection and measuring processor utilization are discussed.

Abbreviations:

3G – Third generation wireless communication system

3GPP – 3rd Generation Partnership Project

AMPS – Advanced Mobile Phone System

ATM – Asynchronous Transfer Mode

BC – Base Control

BCNM – Base Control Node Manager

BS – Base Station

BTS – Base Transceiver System

CDMA – Code Division Multiple Access

CN – Core Network

CPI – Clock Per Instruction

FDD – Frequency Division Duplex

FDMA – Frequency Division multiple Access

FIFO – First In First Out

GGSN – Gateway GPRS Support Node

GMSC – Gateway Mobile Switching Centre

GPRS – General Packet Radio Service

GSM – Global System for Mobile Communication

HLR – Home Location Registry

IDE – Integrated Development Environment

IP – Internet Protocol

ITU.T – International Telecommunication Union

LRS – Logical Resource System

ME – Mobile Equipment

MMS- Multimedia Message Service

MSC – Mobile Switching Centre

NBAP – NodeB Application Protocol

NMT – Nordic Mobile Telephone

NodeB – UMTS RBS

O&M – Operations and Management

OOAD – Object Oriented Analysis and Design

OS – Operating System

OSE - Operating system embedded

PB - Power balancing function of RBS

PSTN – Public Switched Telephony Network

QoS – Quality of Service

RBS – Radio Base Station

RLS- Radio Link Set-up function

RNC – Radio Network Controller

RoseRT - Rational Rose Real Time

RTOS – Real Time Operating System

RTS – Runtime Service Library

SGSN – Service GPRS Support Node

TA – Traffic Application

TACS – Total Access Communication System

TDMA – Time Division Multiple Access

TDM – Time Division Multiplexing

TDD – Time Division Duplex

UMTS – Universal Mobile Telephone System

UTRAN – Universal Terrestrial Radio Access Network

USIM – Universal Subscriber Identity Module

VLR – Visiting Location Registry

WAP – Wireless Application Protocol

WCDMA – Wideband Code Division multiple access

Table of Contents

1. Introduction

1.1 Background

Wireless communication in its short span has become one of the major forms of the communication. From its primary function of carrying voice over an air interface it has now moved on to support rich multimedia applications and Internet facilities

In the wireless communication the only form of communication over an air interface is between RBS and ME. Rest of the communication is over the high-speed landline network. Thus the connectivity of the mobile user to the wireless network is mostly depended on functioning of RBS. RBS functionality is either related to the traffic control function or either related to the operations and management (O&M) function.

RBS functionality is implemented in both hardware and software. The execution of traffic application is dependent on scheduling done by both RTOS OSE and IDE RoseRT. The tasks functionality implemented in software has specific time stringent requirements. If the tasks present in the system are more than the load RBS can handle it will lead to an invocation of overload situation. Overload situation has to be avoided, as one of the primary goals of 3G is to support no drop of calls and allowing continuity of services for users.

To control overload situation, RBS has an implementation of overload protection functionality. Overload protection mechanism aim is to facilitate continuous communication of service for already connected users even under heavy workload. An activity that leads to heavy workload like Radio Link Set-up (RLS) is deactivated till the processor load is reduced to the satisfactory level.

This thesis tries to study the present overload protection mechanism and study the effective ways of implementing overload protection mechanism.

1.2 Purpose

The main goal of this thesis is to check implementation of present overload protection mechanism in RBS and to:

- Understand the complex environment under which RBS is developed
- Develop load simulator tool that represents load created by traffic application and runs along with overload protection processes.

- Different means to measure processor utilization and other ways to ensure proper working of overload protection.
- Understand the working of OSE and RoseRT environment and flow of message between these two environments.

1.3 Thesis Outline

Wireless communication: Part 2 gives brief view of wireless communication overview and its generations, WCDMA architecture and features, RBS architecture and functionality.

RBS as an Embedded System: Part 3 describes about role of RTOS in RBS and covers implementation scheduling done by RTOS OSE and IDE RoseRT.

Present overload protection: Part 4 presents the present overload protection mechanism implementation, its needs, and its principles.

Simulator Implementation: Part 5 contains the details about the simulation program developed for an overload protection and results observed from the simulation.

Solution and Analysis: Part 6 presents solutions for overload protection and other ways of implementing overload protection.

Conclusion: Part 7 summaries the work done

2. Wireless communication

2.1 Wireless Generations

Wireless communication usage goes back to a century with its usage by Guglielmo Marconi for sending wireless telegraph. There has been three wireless generations (see figure 2.1) with latest generation being 3G whose services are being provided by service providers like AT&T, Cingular, T-Mobile, NTT [46], but still the second generation and the technology developed during second and third generation is still dominant in the market.

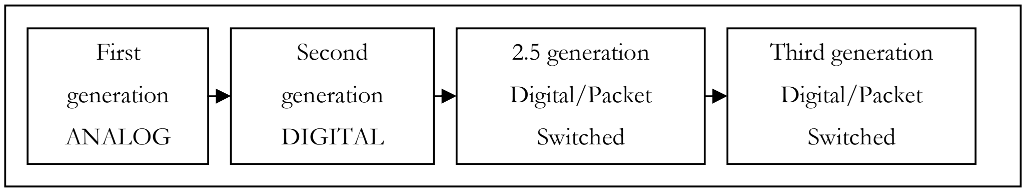
First generation ANALOG
Second generation DIGITAL
2.5 generation Digital/Packet Switched
Third generation Digital/Packet Switched

Figure 2.1: This picture shows the transition growth of wireless communication.

***First Generation**:* It made use of analogue circuit switched network. It was developed with an aim to free people from using fixed lines telephone and gives users mobility in their telephone usage. The military people primarily used this set of wireless communication. It got evolved into commercial field and was launched as Advance Mobile Phone System (AMPS) in USA, Nordic Mobile Telephone (NMT) in Scandinavia, and Total Access Communication System (TACS) in UK.

***Second Generation**:* This generation marked a change from analog communication system to digital communication system. It provided higher network capacity and better security compared to the previous generation. The standards that prevailed in second generations are Global Systems for Mobile Communication (GSM) and Code Division Multiple Access (CDMA). CDMA is the name of the technology as well as transmission technique that works on the principle of spread spectrum. GSM uses Time Division Multiple Access (TDMA) transmission techniques that allocate user different time slots on a given frequency.

***2.5 Generation**:* The long gap between second and third generation and high cost involved in upgrading of network to 3G lead to 2.5 wireless generation. This generation marked beginning of packet switched data elements such as Internet through Wireless Application Protocol (WAP) and Multimedia Messaging Service (MMS). This was provided through General Packet Radio Service (GPRS), which was the first wireless technology to make use of packet switched network.

***Third Generation**:* It supports packet switched data elements like Internet and multimedia over wireless communication at higher bit rate. It makes use of Wideband Code Division Multiple Access (WCDMA) as its radio transmission technology over an air interface. It is compatible with second-generation technologies like GSM.

2.2 Wireless Communication Overview

In wireless communication the area that service provider aims to provide is subdivided into cells and each cell is handled by the Radio Base Station (RBS). As shown in figure 2.2, wireless communication comprises of communication between the RBS and Mobile Equipment (ME) over an air interface, rest of the communication between RBS and Radio Network Controller (RNC), and then between RNC and Mobile Switching Centre (MSC) are on high-speed landlines network.

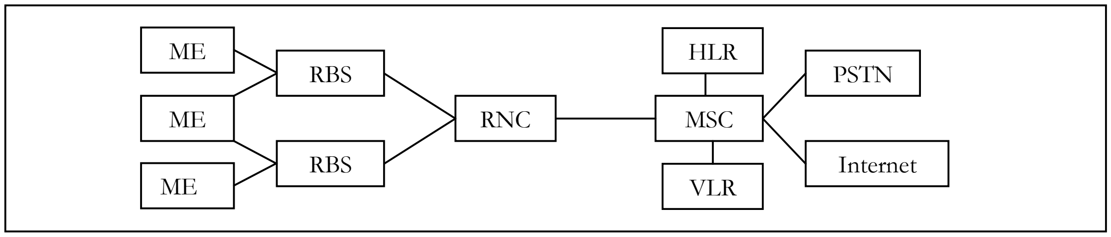

Figure 2.2: Wireless communication overview

ME move from cell to cell and make use of principal of handover and frequency reuse to connect to RBS. RNC function is to handle handover and radio channels between different RBS. Mobile Switching Centre (MSC) connects the mobile network to the Public Switched Telephone Network (PSTN). MSC contains Home Location Registry (HLR), which includes information about the subscriber and the location of the ME and Visitor Location Registry (VLR), which contains dynamic information and copies from HLR for ME currently in the area covered by a RBS.

Besides handling of each cell, RBS carries out the functionality of the radio multiplexing. Multiplexing over an air interface is required to enable multiple ME to connect to the RBS and allow efficient usage of bandwidth. Multiplexing can be achieved in terrestrial network through

Time Division Multiple Access (TDMA), or through Code Division Multiple Access (CDMA), or through Frequency Division Multiple Access (FDMA).

2.3 UMTS / WCDMA

Third generation of wireless communication is based on Universal Mobile Telecommunications System (UMTS) standard and uses WCDMA as its air interface. UMTS is a set of open specification that is backed up by 3rd generation partnership project (3GPP), 3GPP has its representative bodies from Europe, Japan, China, Korea, and USA, whose aim is to set a truly global standard for wireless communication and lead to a seamless global roaming for mobile users.

The major change in wireless generation has been the way radio transmission is multiplexed over an air interface. The multiplexing technique used for transmission decides how the radio spectrum can be divided into channels and how these channels separate different users of the system. Channel allocation can be done either based on some scheduling mechanism or randomly chosen. If randomly chosen then user cannot be for sure whether access to the network will be contention free but if scheduled then the channel allocation is fixed and resources are allotted dynamically based on the user requirements.

UMTS air interface is based on WCDMA, which is an extension of the CDMA multiplexing methodology. CDMA technology is based on the principle of spread spectrum, where allocated bandwidth is much higher than the requirements of the user. It is based on scheduled channel allocation. All the users use the same carrier frequency and the codes are used for distinguishing the different users and channels. Each user in the mobile network can only correlate to the message that has the user code and all other messages looks like noise due to the correlation technique in ME.

In WCDMA, the user information bits spread over a wide frequency bandwidth is obtained by multiplying the user data rate with a spreading code of sequence. There are two basic modes of operations in WCDMA are Frequency Division Duplex (FDD) and Time Division Duplex (TDD). In FDD separate carrier frequency are used for downlink and uplink activities. TDD uses one carrier frequency for sharing frequency between uplink and downlink.

WCDMA supports data speed from 384 kbps till 2 Mbps and provides a bandwidth of 5 MHz for each channel carrier. [38] It supports transmission from multiple users simultaneously and uses variable spreading to support multi-code connections. The signal that is transmitted from the RBS or ME is subject to reflections, diffraction, and attenuations. This is known as multi-path propagation where signal can take different directions to get to the end user and it can be either RBS or ME.

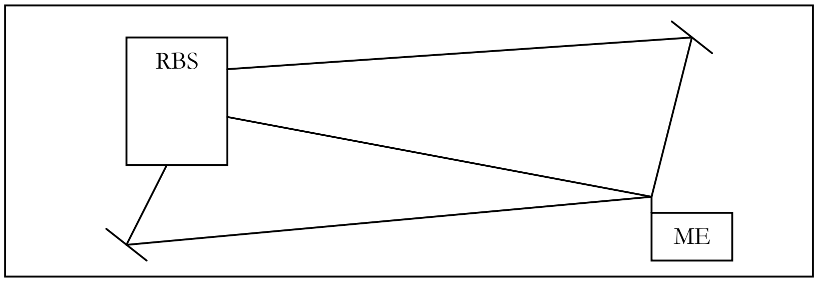

Figure 2.3: Showing multi-path propagation in WCDMA network

In figure 2.3 it shows that ME or RBS receives multiple signals due to multi-path propagation, it has to properly scrutinize received signals and select only one of the signals from received signal and rest has to be discarded.

Architecture: UMTS network is an evolution from GSM, second-generation technology, and GPRS technology, 2.5-generation technology. The architecture of UMTS as shown in figure 2.4, the network bears a strong resemblance to these technologies. It includes of two main parts:

1. UTRAN (UMTS Terrestrial Radio Access Network)
2. Core Network (CN)

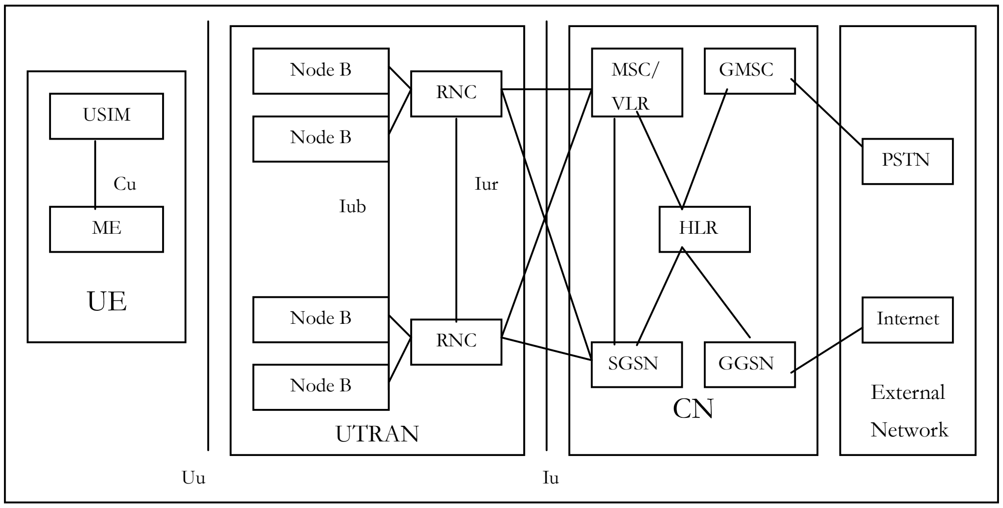

Figure 2.4: UMTS Architecture [6]

UTRAN comprises of components that handles the air interface for managing connections with multiple ME. The two main components of the UTRAN are RBS and RNC. (Note: In UMTS, RBS is referred to as Node B but for sake of uniformity in this thesis RBS is used through out the documentation). Core network is responsible for switching and routing of calls over PSTN or Internet based on the nature of traffic. It includes of MSC, HLR, VLR, GGSN, SGSN, and GMSC. All the interfaces used in UMTS standards are defined to allow usage of different equipment from different manufactures.

UTRAN: It includes of one or more radio network systems (RNS). RNS comprises of RNC, several RBS, and UE.

Radio Network Controller (RNC): RNC is responsible for the control of radio resource of UTRAN. RNC interfaces with CN via lu interface, via lub to control RBS, and via lur interface between RNC for soft handover. It plays a pivotal role in performing of activities like:

- Power Control (PC)
- Handover Control (HC)
- Admission Control (AC)
- Load Control (LC)
- Packet Scheduling (PS)

Radio Base Station (RBS): It is similar to GSM's Base Station (BS) or Base Transceiver Station (BTS). RBS is the physical unit for handling radio transmission and reception from various cells. RBS performs the air interface processing, which includes channel coding, interleaving, rate adaptation, and spreading. The connection with UE is made via Uu interface, which is actually WDMA air interface used for communication between RBS and ME. It is also responsible for providing soft handover and inner closed loop power control for ME.

User Equipment (UE): UE comprises of Universal Subscriber Identity Module (USIM) and Mobile Equipment (ME). ME is the device used by user for communicating with RBS over an air interface. USIM is the smart card technology used for holding user identity and personal information.

Core Network: It is responsible for handling both circuit switched and packet switched traffic. The main entities that play a pivotal role in core network are:

HLR: A database that holds all the necessary information about ME like its subscription information, location of ME registered at that time for enabling charges, and routing of calls over MSC or SGSN.

MSC: It is a link that connects the wireless network and fixed network. MSC performs all necessary functions in order to handle the circuit switched and packet switched traffic to and from ME. A ME roaming in a certain MSC are handled by VLR in charge of that area.

Gateway MSC (GMSC): It is a switch that handles traffic of circuit switched connections between ME. At this point UMTS network is connected to the external circuit switch network like PSTN.

Serving GPRS Support Node (SGSN): This node is used for handling packet switched services like multimedia or Internet.

Gateway GPRS Support Node (GGSN): The support node has the same functionality for handling the packet traffic as the GMSC does for the circuit domain.

Capabilities: The major change in UMTS architecture is the multiplexing technique used by an air interface for radio transmission. There have been changes in RNC and RBS to support new requirements but the air interface marks a major change in this network.

Some of the main capabilities of the UMTS network are [38]:

- Capability to support packet switch and circuit switch data at higher rate – 144 Kbps for mobility traffic, 384 Kbps for pedestrian traffic, and 2 Mb/Sec for indoor traffic.
- Supports interoperability through defined standard interfaces.
- Common billing and user profiles, standardization of user profiles, call details, and information exchange between service providers. This feature is a result of 3GPP that provided a common platform for different service providers to come in contact with each other more easily and make contracts for exchanging information about the user moving from one region to another.
- Support for packet switched data like multimedia services, Internet, and mail facilities.
- Fixed and variable bit traffic to support user's on demand bandwidth requirements.
- Asymmetric data rates in uplink and downlink.

- Intersystem handover support between GSM and WCDMA

2.4 Radio Base Station

It is a physical unit used for handling radio transmission and reception with cells. RBS is referred to as Node B in WCDMA/UMTS Terrestrial Radio Access Network (UTRAN). RBS is connected to both RNC and ME to carryout its functionality. In figure 2.5 shows how RBS looks physically.

Figure 2. 5: An indoor RBS [19]

Functions: RBS functionality can be divided into two parts

1. Traffic related functions that deal with communication with RNC and ME for handling of cells, common channels, dedicated channels, and ATM links.
2. O&M functions are used to set the system into an operational state, handling of equipment malfunction, and monitoring the performance of RBS.

RBS receives its input from RNC over lub interface. The lub interface is divided into physical layer, ATM adaptation layer, and network layer for frame handling. The topmost layer is Node B Application Protocol (NBAP), which handles different air interface channels. NBAP functions are divided into dedicated procedures each terminating in separate logical ports in RBS.

The communication between RNC and RBS is controlled by BCNM (Base Station Control Node Manager). BCNM handles reduction of processor workload in overload protection mechanism. The start and stop of overload protection is done through this control unit in RBS. To reduce workload the parameters used for RLS (Radio Link Setup) over NBAP are changed. This parameter change will block RLS function till workload is reduced in the system.

All these functionalities run on Ericsson's cello platform, which acts like a middleware providing facilities of database and Real Time Operating System (RTOS). The cello platform is used in development of switching network nodes such as simple ATM switches, Radio Base Stations (RBS), or Radio Network Controllers (RNC). It provides a robust real time distributed telecom control system which supports ATM, TDM, or IP transport. The nodes that uses cello can run between 1.5 Mbit/s – 155 Mbit/s. (See Appendix B for more details on cello)

RBS implements following functionality to support radio traffic:

1. Platform Independent
2. Radio Transport Functions
3. Synchronization functions
4. Bearer functions
5. Traffic control functions
6. Configuration management function
7. Fault management function
8. Performance management function
9. User Interface function
10. Infrastructure function

Node Architecture: As shown in figure 2.6, RBS node comprises of user plane functions to implement transport, base-band, radio, and antenna near parts functions and control plane functions that provide functions related to the traffic and O&M activities.

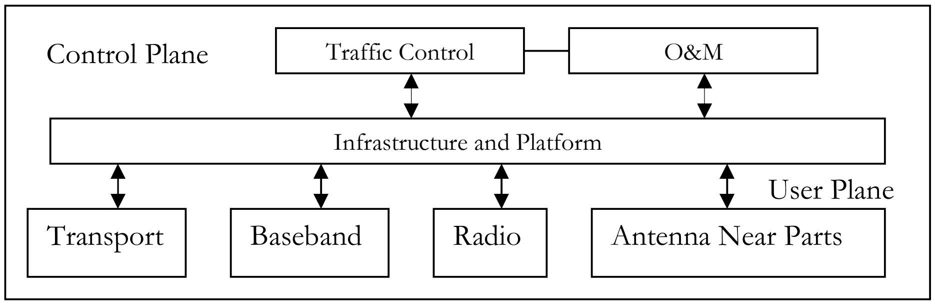

Figure 2.6: RBS Node Architecture

(Note: All these functionalities and this node architecture form the basis for the simulation model that will be developed to check overload protection mechanism.)

As RBS node architecture is three layered as this eases in development of such a complex application of RBS. Each layer implementation is carried out separately allowing layered approach to be carried out effectively:

1. Infrastructure and Platform: This layer comprises of cello platform and application layer. Functionality implemented in application layer runs on cello platform. Cello platform runs independently from application layer and allows possibility to change processor and other hardware parts without changing implementation in application layer.
2. Control Plane:
    a. Traffic Control: In this layer, inputs supplied by RNC and RBS internal functions are all handled. It includes of four sub layers:
        i. Hardware layer handles details of a specific board.
        ii. Equipment layer hides details of specific board functionality.
        iii. Logical resource layer provides logical resources such as ATM links, channels, cell carriers, etc. It transforms functions into operations on devices.
        iv. Traffic service layer is used to handle NBAP procedures. It receives request from RNC and utilizes above layer to perform its functionalities. Through this layer nodes functionality is controlled.

    b. O&M View: This view eases in configuration of system and allows seeing log files through GUI interface. It tries to segregate MO (Managed Object) from RO (Resource Object). MO represents alarm/event generating objects and logs about notification of subscription objects. RO handles low levels details of MO.
3. User Plane: This layer deals with functionality implemented in hardware.

Base Station Control (BC) implements traffic service layer in RBS and takes part in all control functions. It uses logical resources (LRS) of RBS to fulfil its functionalities. BC performs following functionalities.

1. Common Procedures:

- Common transport channel set-up/reconfigure/delete
- Common measurement
- Cell set-up/reconfiguration/deletion
- Resource status indication
- System information update
- Radio link set-up/release
- Reset

2. Dedicated Procedures:

- Radio Link Addition/Deletion
- Radio Link Configuration
- DL power control
- Dedicated measurement
- Compressed mode
- Radio Link Failure/Restore

3. NBAP message trace support.

Overload protection control functionality is implemented in BC. When overload protection signal is received, BC stops NBAP parameters related to RLS till workload is reduced. In rest of the documentation mostly traffic service layer is discussed as BC is part of it. In next section we will see how functionality implemented in RBS uses OSE as its RTOS and uses RoseRT as its IDE and for its runtime environment.

3. RBS Implementation

The traffic service layer functionality of RBS is very complex. It has certain functions that include being reactive to Uu interface through which it communicates with ME and with lub interface for communication with the RNC.

All these functionalities are implemented like in real time embedded systems. Radio Base Station (RBS) is a soft real time embedded system, it follows same rule of a soft real time system. In soft real time system, missing of deadline does not lead to a major catastrophe but leads to provision of bad QoS to the user. The important characteristic of real time system is that implementation has to not only comply with logical correctness but also need execution of events on time.

Timing plays a very critical part in the real time system and any miss of target will lead to the user dissatisfaction. Tasks can be performed on time if the proper scheduling of tasks is done. In most real time systems time management, resource allocation, and scheduling are provided by Real-Time Operating System (RTOS). RTOS scheduling can be static or dynamic, pre-emptive or non pre-emptive, and fixed priority based or random based.

Scheduling is done through schedulers and dispatchers. Schedulers are used to arrange running of a task and plan when a task is going to start, for doing this it generates a table that is used by the dispatcher. The dispatcher executes the task present in the tables generated by scheduler and the timer controls task invocation.

In RBS development, the important roles are played by RTOS and application runtime environment. In section 3.1, a brief overview of processes and how they are scheduled in OSE is covered. In section 3.2 it covers RoseRT general overview and runtime mechanism of RoseRT Runtime Service Library (RTS).

3.1 Role of Real Time Operating System (RTOS) in RBS

An Operating System (OS) is a computer program that is responsible for managing all the tasks in the systems like memory allocation, job scheduling, interrupt handling, and distribution of input and output. [45] The requirements of OS in an embedded system differ from general OS requirements that embedded system OS has stringent timing requirements.

RBS needs RTOS to provide basic set of functionalities that can be used by any other processes in the system. The RTOS in RBS manages resource management, portability, fault management, scheduling, and debugging. RTOS scheduling policy defines how particular process will behave during execution.

RTOS has to effectively handle the timing requirements. Hard real time systems are time bound while soft real time systems are priority based. Scheduling plays a pivotal role as the processor have multiple programs running simultaneously, and based on scheduling it knows what task needs to be executed. In soft real time system, scheduling can be either fixed priority based where priorities are assigned before hand or dynamic priority based where priority are assigned at runtime on executing parameters like deadlines.

Scheduling algorithm is used to ensure critical timing constraints are met for reaching deadlines. [43] The burst time, which is the time taken up by the process for execution on processor and the I/O wait, has to comply with the system requirements on the maximum time the task can take to make system meet its real time requirements.

OSE: Operating System Embedded (OSE) is a product of ENEA and is used in Ericsson's RBS. OSE is a dynamic, fault tolerant and distributed real time operating system. It is designed to meet real time demands as well as to meet requirements of high availability, reliability, and safety. OSE comprises of two types of kernels; one designed to run on target environment referred as hard kernel and other to run on host machine called soft kernel.

The main functionalities of OSE are [42]:

1. Resource allocation
2. CPU allocation
3. Event priority
4. Communication
5. Timing
6. Error handling

OSE comprises of processes (function with a context) and signals (information carriers). Processes are of five types in OSE: Interrupt process occurs on event of hardware interrupt or software event like a signal. Timer interrupt processes are a special case of interrupt processes called in response to changes in system timer. Prioritised processes are written as infinite loops and runs till the higher priority process is ready to execute. Background processes runs in strict time-sharing environment. Phantom processes do not have any programmable code and is used for communication with processes across target boundaries.

As shown in figure 3.1, OSE processes are considered to be in either one of these three states waiting, running, or ready state.

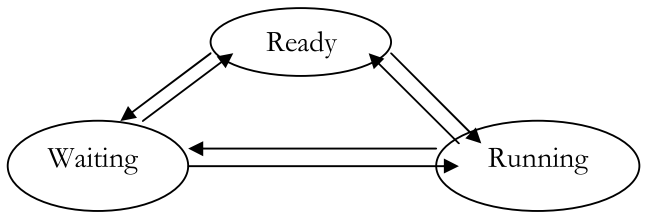

Figure 3.1: Process handling in OSE [19, 23]

If a process is in waiting state then process waits for some event to occur. It does not require any system resources till process is in running state. The running process is the one that has the highest priority among any ready process and can be pre-empted only by the process having higher priority than the priority of running process. This is the special feature of OSE that enables scheduling based on process priorities. In ready state, process wait until all processes with higher priority have finished their execution or have entered into the waiting state. The two processes when they share same priority and they both are in ready state, OSE uses round robin scheme to schedule tasks.

OSE scheduling is based on pre-emptive priority scheduling, where low priority process running can be pre-empted by high priority process that is in ready state. Processes are scheduled based on their priority assigned; possible number of priorities is 32 in OSE. The scheduling of OSE plays a pivotal role in providing traffic service layer functionality. During an overload situation how overload protection implementation functionality is scheduled and how resources of the system are handled are all dependent on scheduling policy.

3.2 Rational Rose

Rational Rose is a visual design tool developed by Rational Software to facilitate object oriented analysis and design (OOAD). This tool is made with collective effort of three pioneers in UML, Booch, Jacobson, and Rumbaugh. [20] RoseRT allows creation of visual model based on UML to simplify, increase efficiency, and make software less error prone through out software development.

Rational software used in development of real-time system is Rational Rose Real Time (RoseRT). It is a tool used in modelling and implementing of reactive systems, complex, event driven, and concurrent systems such as RBS. The advantage of using RoseRT is that modelling eases verifying design process and allows solving the problem at a higher abstraction level. The model driven approach provides rich information structure that eases in understanding behaviour of the system.

The model driven approach comprises of various components that can be used in modelling the system and these are: classes, components, relationships, objects, operations, modules, processes, and processor. These components can be used either in physical model, logical model, static model, or in dynamic model.

The RBS developed using RoseRT is a collection of capsules that can exchange asynchronous messages. The execution time of the traffic application is spent in executing developer written code; RoseRT generated code, and RoseRT runtime systems (RTS). [20] The execution of code is done on a single thread or on multiple threads. The threads that are executed have their own controller that comprises of main loop, has a duty of picking up message from the message queue, and delivering it to the destination capsules. Each controller has five different message queues for each priority

The behavioural specification of the RoseRT is specified through state-chart diagrams. The different components used in state-chart diagrams showed in figure 3.2 are [14]:

- Capsules: defines the possible state the process can be in. Code is executed when state is entered and allows transactions to take place.
- Transitions: the action that can be performed by the process under a certain state
- Ports: they provide a means to receive signals from other capsules
- Protocols: defines message syntax and semantics based on which communications between two ports take place.

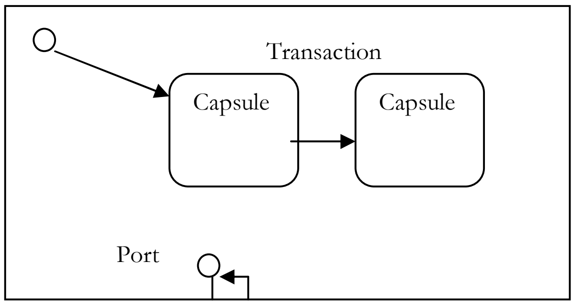

Figure 3.2: Different state-chart diagrams parts of rational rose

State diagrams define how objects should react to an external stimulus. These objects can be either active object or passive object. Passive object does not have their thread of control and can react only when invoked by method calls from external callers. Passive object are very useful where resources are scarce in the system. Passive classes provide improved memory usage and better performance.

Active classes are useful but have high performance and memory overhead associated with them. The active classes have their own thread of control and communicate with other capsules using asynchronous messages. The communication between capsules is done through ports. Ports are used for two-way communications and use signals for communications. If two ports are wired they can communicate based on interface defined using protocols.

RTS implements a virtual machine that allows executing a model developed in RoseRT. The capsule of state-chart diagrams is scheduled by the scheduling mechanism of controller in RoseRT. The scheduling involves dispatching of messages to appropriate capsules and when message is received by appropriate capsule related transitions are executed. Next message is dispatched only when task submitted to the transition is completed.

RTS runs on OSE, OSE is responsible for scheduling controller and controller schedules traffic application functions in RoseRT environment. Controller is like a process in OSE and is scheduled by OSE based on pre-emptive scheduling. The capsules in RoseRT belonging to same logical thread can efficiently utilize processor as they can run concurrently and are scheduled by controller based on First in First out (FIFO) principle.

Ericsson's uses in-house controller to make it run-able on cello platform. When inter-capsule message is sent between capsules a synchronization message is sent to the OSE message queue of that controller to make it schedulable by OSE.

To receive message from OSE the message has to be registered in RoseRT using REGISTER_OSE_SIGNAL. When message is received from OSE, it is translated to RoseRT signal and memory for the OSE message is freed up. Messages from OSE message queue are handled at lower priority and are sent to appropriate capsules only when queues in RoseRT are free.

The scheduling of capsule in RoseRT requires first scheduling by OSE and after that RoseRT controller schedules capsules based on FIFO. The scheduling of overload protection will require using of both OSE and RoseRT controller to invoke some action to control overload situation.

4. Overload Protection - Need and Principles

RBS functionality execution is dependent on how OSE schedules controller and how RoseRT controller schedules traffic application functionality. The development methodology of RBS implementation has to meet functional requirements like Radio Link Set-Up and Power Balancing functions, and non-functional requirements issues related to QoS, meeting deadline, reliability, availability, and security.

RBS to provide QoS requirements has to efficiently handle an overload situation where system has more jobs than it can handle. To safeguard from overload situation, overload protection is implemented. Overload protection is a measure in RBS to ensure the availability of the network connection for already connected users and ensure reliability of the services offered by the system. Overload Protection has to ensure reduction of the throughput, handling of QoS at acceptable levels, avoid system from restart, and providing continuity of services for already connected users.

As per ITU.T overload protection should secure time constraints and reduce throughput by rejecting excessive amount of tasks already present in the system. [35,28] So there are two functionalities of overload protection. First functionality is related to the time constraints, as exceeding time limit will lead to missing deadline and providing service after deadline is of no use in real time system. Second functionality of reducing throughput requires giving preferences to some tasks. This requires a kind of traffic differentiation, which is a way of distinguishing activities belonging to which function of RBS, when activities are permitted into the system so that when overload situation arises it can reduce throughput.

Overload is directly related to the amount of workload the processor can handle. Processor workload is defined as the mixture of program and operating system commands that are submitted on the machine for execution. [40] The reason for system getting overloaded could be due to processor having more processes than it can handle. As message queue in RoseRT comprises of both intra-capsule, inter-capsule, and OSE messages it can sometime lead to message queue overflow.

The proper implementation of overload protection will lead to reduction of resources that are used up in overload situation and avoid system from restart. The present implementation provides details about how to control network resources under overload situation, details about controlling the admission control of the overload protection, and techniques for making signal received till NBAP handler to stop RLS function to reduce workload.

The next section elaborates the need for overload protection and is followed by a section that presents principles of overload protection. The last section covers the present implementation of overload protection mechanism.

4.1 Need for Overload Protection

It is important to fulfil QoS requirements to safeguard system from overload situation as it leads to performance degradation, drops in the calls of already connected customers, and above all the time taken to restart and start working again.

Traffic application comprises of 20 functions and each function has sub-functions to handle the air traffic. Some of the most prominent functions are RLS, PB, RLR, etc. When situation of overload arise the activities that put workload on the system, like RLS, are stopped for a period till processor workload is reduced.

In embedded systems, worst-case scenarios are used to evaluate performance of the system. As these scenarios helps us in giving information about the maximum performance and behaviour of the RBS in situation such as overload situation. The main parameters that are captured for RBS are execution time, intensities, and amount of workload it creates on the processor when it is executing.

Following are the worst-case scenarios for some of the main functionalities of traffic application in RBS:

Functionality name	*Execution time*	*Intensity per second*	*CPU load*
Radio link setup (RLS)	5.5	61	33.3%
Power balancing (PB)	.056	286	16%
Radio link release (RLR)	2.0	61	12.1%

It has been noted the processor performance of nearly 100% is observed when all 20 functionalities of traffic application are running in worst-case scenarios.

To ensure the better QoS under overload situation, these measures have been proposed in the system:

- Software optimisation of the control plane activities
- Faster CPU that involves moving from PowerPC 750 to PowerPC GX. It can increase performance by nearly twice.
- Cache increase will lead to reduction in CPU – memory communication
- Multi-processor system where each increase in the processor leads to performance increase by approximately 80% and each processor will handle these tasks separately encode/decoder, and AAL2 connections.

The solutions presented above impose a new cost in the development of the system and changes in hardware. It is not a viable solution as companies to be competitive try to reduce cost of development and release product early in the market to capture the market. But the changes suggested require considerable development effort and subsequent delay in the release of the product.

There is very less probability of worst-scenarios will happen and system will be under such heavy overload situation. Making such huge changes will not be that high regarding instead an overload protection mechanism properly implemented can solve issues pertaining to an overload situation and ensure that calls for already connected customers remains connected and new connection are stopped till workload is reduced.

4.2 Overload Protection Principles

To implement overload protection it requires considerably attention that the means used for overload protection does not itself become a load that will take the execution time of the CPU or consume resources of the system.

Several principles are followed in order to carry out overload protection principles.

1. The proper means to measure processor utilization of CPU workload is needed. If utilization is more than 70% it should invoke some operation to stop time-consuming activities in the system.

2. To apply edge control as early as possible in order to avoid late rejection problem. When the 50% processor utilization is noticed in set time interval it should lead to 10% reduction in throughput of maximum throughput allowed.
3. The users connected have higher preferences compared to new connections establishment. New connections lead to high processor utilization and are stopped till workload is reduced. This rule requires some form of traffic differentiation to implement this functionality.
4. For continuous working of RBS, it should be ensured that system is available under overload situation and does not lead to restart.

Above principles forms the cornerstone on which overload protection mechanism is designed. All the principles are not fully present in the present system. First and second principles are lacking and there is no traffic differentiation done. Because of no traffic differentiation messages enter into the system and are executed in RoseRT and only when there is communication with RNC over NBAP some appropriate action is taken.

The techniques followed should help in reducing overload in the system, avoid system restart, and avoid internal buffer overflow. The possibility of checking the system when overloaded through measurement of OSE queue length is ruled out because of cello platform nature.

4.3 Present Overload Protection:

Overload protection present implementation is invoked when timeout signal is received by higher-level priority process when it is waiting for signal from a lower-priority process. It follows the principle of starvation theory to enable its working. According to the starvation theory a low priority process never gets access to the processor due to higher effective process access to the processor.

As shown in figure 4.1, the overload protection mechanism is implemented through two processes running along with the main traffic application. These overload protection processes starts when RBS starts and run continuously like daemons in background and ensure system working under overload situation by invoking overload mechanism.

First process that plays pivotal role in overload protection is Load Control Server that runs at the highest priority (In OSE the process having priority 11 is higher compared to process having process priority 12). The responsibilities of this process are as follows:

- To send signal to Load Control Response Server process and wait to receive back response from the process. The message sending involves starting timer to keep track of time it took to receive back signal.
- If it receives back the signal before the completion of the maximum time set (like 100ms) it will wait for elapse of time interval to send back signal to the Load Control Response Server.
- If the signal sent reply is not received in time interval set, it sends signal to traffic application controller to activate overload protection, that is to stop radio link set-up till workload is reduced. If it receives signal after lapse of set time from Load Control Response Server it omits that signal.

The second process is traffic application running at middle level priority under RoseRT virtual machine. The overload protection signal expected from Load Control Server process is registered in this application. The signal when received appropriate action is present in Base Station Node Manager (BCNM) section of RBS. The action includes taking steps to stop RLS Connection. The workload is produced by this process and sometime leads system to an overload situation and causes lower priority process to starve. In overload situation Load Control Response Server process will starve.

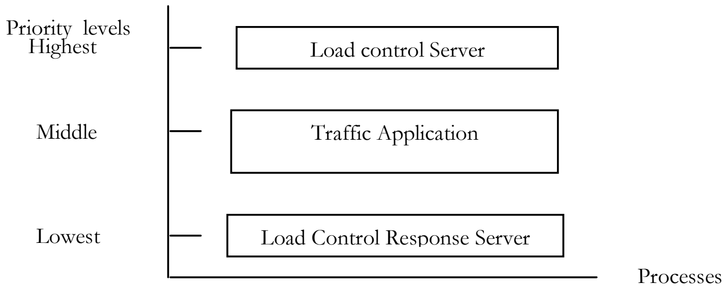

Figure 4.1: Showing different processes and their priority in overload protection implementation

The third process along is Load Control Response Server process. The responsibility of this process includes receiving signal from Load Control Server and responding back to it.

To cease overload protection, Load Control Server sends signal after lapse of 100 ms to Load Control Response Server. If Load Control Server receives back signal in set-time period it will call for the cease of overload protection by sending message again to BCNM. All the activities that are carried out during overload protection are recorded in cello database (see appendix B for more information about cello) for O&M purpose. The information stored in cello includes the time overload protection was active, number of users rejected, and other relevant details.

Overload protection needs to meet its principles as discussed in previous section for avoiding system restart and services to the connected users. To ensure overload protection proper working, processor utilization has to be present in numerical values and implementation should include scheduling of both OSE and RoseRT controller.

In next section we will see how simulation is used for checking above conjecture. The topic covered includes steps taken in development of the simulation program, analysis of the results, and verification and validation of the model.

5. Simulation Implementation and Results

The present overload protection mechanism needs to be checked way to see whether it meets the purpose it is designed for. During start of the thesis it was reported that the result produced by overload protection mechanism under situation of overload varies. Sometimes its works and in some situations it does not work.

The fluctuation requires some analysis of the system in order to find out the reason for its failure. There are various ways through which system analysis can be done. It is very difficult to analytically verify model such as overload protection as it involves two queues and also lacks semantics of how messages are handled in RoseRT making the task difficult.

The other possible way of understanding system behaviour is through Simulation. Simulation is a means through which part of the system working is implemented to achieve desired results. Simulation is extensively used nowadays for understanding system behaviour and is one of best ways of developing software. The major implication with simulation is that it is time consuming, and simulate only some part of the system is implemented, but complete system behaviour cannot be understood. [22]

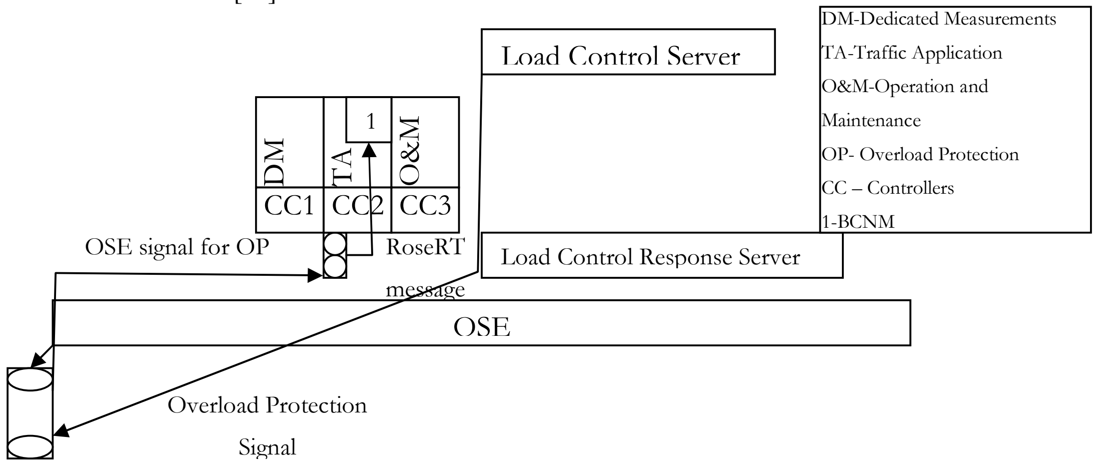

Figure 5.1 showing flow of overload protection signal from OSE to RoseRT

The working of overload protection in theory involves sending message to the BCNM capsule to turn off parameter of NBAP handler when situation of overload is found. In a working system it involves flow of messages as presented in Figure 5.1, message is first sent from Load Control Server to OSE message queue, then message from OSE message queue is only retrieved by the controller when there are no more a messages left in RoseRT message queue, there is no preference given by RoseRT for messages which are of high priority in OSE. This leads to some kind of unpredictability in the system.

The other problem is that the OSE message queue had problem of buffer overflow and also RoseRT message queue has queue utilization of 100% [14]. So under situation of overload it has been found that only 25% of the resources are available in the system [1,2,3]. To understand system working under some such situation requires simulation program to see possibilities that are leading to unpredictability in the system.

The simulation program aim is to develop a model that allows checking present overload protection mechanism in order to investigate it's working under situation of overload. The need for investigating is to understand whether under overload situation does the message dispatched from OSE processes reaches destined capsules, does message get received under RoseRT, and understand how pre-emptive scheduling works for OSE.

To simulation of the program requires following some steps in order to get understanding of the system. If we divide it into small steps there are 12 stages that simulation program has to undergo but it can be divided into 4 steps:

1. Definition of problems and objectives
2. Model building and data collection
3. Simulation experiment and analysis
4. Documentation and implementation of the results

There are various ways through which simulation can be carried out. One can make use of Simulink® to model the system in it and see how queues work, whether overload protection signal is received or not can be checked. But the problem is that the working of system is not known if pre-emptive scheduling is modelled in Simulink it cannot be guaranteed for certain that behaviour produced by the system matches with target environment.

So for this purpose target environment is selected to be same as the development environment use of developing application layer software of RBS. As through simulation it will check working of pre-emptive scheduling of the OSE, how controller handles messages delivered from OSE, and mechanism followed by controller for sending messages between capsules. All this can be observed only when the simulation model is developed with the current system and not developed based on just conceptual model.

This simulation requires getting out the behaviour of the system through simulator and does not involve just modelling. The simulator program will replicate that part of the system that is related to overload protection by creating a right kind of environment in the system that in someway is similar to traffic application.

The program developed is called simulator, and will involve simulating behaviour of the traffic application, overload protection based on the input parameters of the current system. To make simulator runs it will also involves configuration in OSE to make this simulation program run-able.

To develop this simulator we follow steps in developing simulation model. There are basically four steps followed in modelling and all steps undertaken in developing simulator are covered in these steps:

5.1 Problem definition and formulation

The simulation program first step is to get clear why the simulation program is being developed. There are some sub steps in this step and will be covered in this section:

1. Problem formulation:
2. Set of objectives:
3. Overall project plan:

***Problem formulation**:* To study overload protection mechanism of present RBS in order to see the reason for getting unpredictable behaviour of the overload protection mechanism, to study how effective is processor utilization measurement function, and see whether capsule in RoseRT receives message from the overload protection process.

***Set of objectives**:* The task carried out by RBS can be divided into TA, DM, and O&M. One of the functionality of TA it provides is overload protection that allows system to continues its operation even under overload situation by reducing throughput and stopping new connections. The objective of the simulation model is to:

1. To check whether overload protection complies with ITU.T recommendation

2. To check whether overload protection meets its purpose under situation of overload
3. To check how effective is the method
4. To check the possible reasons of unpredictable behaviour.
5. To check processor utilization measurement to check overload is a good means to check utilization or not.
6. To check whether NBAP handler parameter is set off the time message is sent by OSE or takes it time.

Limitations: The simulation program cannot check whether setting of NBAP parameter off is effective way of reducing workload. It does reduce workload but to determine by how much percentage is not possible to calculate through simulation program. The recommendation by ITU.T is considered in this case that stopping new connection does reduce throughput and workload in the system.

Overall project plan: The model should try to use RoseRT and OSE to get current working scenario of the system under situation of overload. Simulating traffic application with certain workload that can lead the system to overload situation is a better solution compared to running the real application. Simulation program tries to use service time and inter arrival time of the current system but the workload created will be done in a way that it leads system to overload situation.

5.2 Model building and data collection

This stage in the simulation program design is where the base of the simulation program on which the implementation will be carried out comprises of following steps:

1. Model conceptualisation
2. Data collection
3. Model Translation
4. Verification
5. Validation

Model conceptualisation: The main idea in developing the simulation program is to enable the scenario where two OSE processes run at different priorities and one process representing traffic application runs with heavy workload in between these processes. The simulator to be built will be simulating traffic application. The model should provide enough means in studying the working of overload protection mechanism and message flow between two message queues.

The workload in the simulation needs to match with the workload generated by RBS to get correct analysis of the working of the system, the frequency i.e. the number of time function takes place in 1 minute or in terms of queuing theory an inter-arrival time of the task, and the duration i.e. the time taken for executing task, in queuing theory it is usually referred by service time of the task.

The simulation model to be developed has only certain activities that represent traffic application and not all the functionalities of RBS. The functionality represented through simulation program does not carry out any functionality of the system does, but only creates workload for the period system performs its work.

The input parameters of RBS are receiving of radio signal for some request. What can be considered is the maximum amount of time a particular can take place. The time is usually called worst-case scenario time period of traffic application. The workload is created for that particular time period which exceeds the set time interval period. If the period exceeds then some appropriate action is taken.

Selection of proper workload is needed to get better understanding of overload protection mechanism implementation. Some of the ways of creating workload is through running real application, or by using some standard benchmark, or through implementation of a kernel. Workload is chosen depending on the application for which it is tested.

For the simulation program, kernel provides an appropriate workload for the overload protection, because this methodology is based on isolating of individual features of the machine to explain the difference in the performance of the real program. [40] Other workload requires lot of development activities to be undertaken. To limit our scope to just test overload protection mechanism of RBS, kernel workload mechanism is best suitable for checking reaction of overload protection when system is under heavy overloaded.

To simulate a workload on processor, program is divided into three capsules. One capsule is referred as a driver capsule; it will be responsible for sending messages between other two capsules. The signal is exchanged between two processes in asynchronous way. Before the communication between two capsules start two processes in OSE are instantiated. Then the communication happens between these capsules, and workload created by capsule tries to make processor busy in capsules and create starvation situation for Load Control Response Server process.

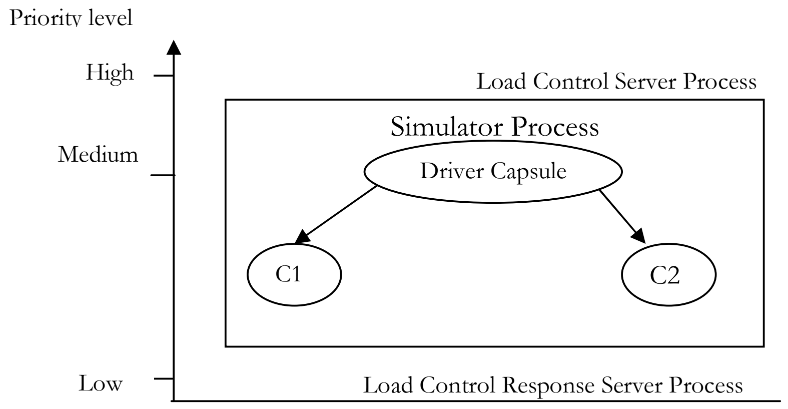
Priority level
High
Medium
Low
Load Control Server Process
Simulator Process
Driver Capsule
C1
C2
Load Control Response Server Process

Figure 5.2: Present working mechanism of overload protection [35]

Figure 5.2 represents different processes running at different priorities levels in simulation program. C1 and C2 represent capsules representing Radio Link Setup and Power Balancing functionality. One capsule represents radio link set-up and other represents power balancing. The driver capsule contains normal loop that executes for a period of time.

The two processes are primarily responsible for controlling overload protection mechanism runs in OSE at two different priorities. Highest priority processes sends signals and wait for response from lowest priority process. Lowest priority process gets time to execute when processes running at priority above are not using processor.

One of the processes running in OSE runs at higher priority, thus it should be able to detect overload situation and send message to driver capsule. To decrease workload driver capsule will stop sending message till cease message is sent by higher priority process. The simulation program will be developed in RoseRT and will work in OSE RTOS. Rational rose environment and soft target (Solaris /simcello) environment is used through out the development lifecycle of implementation.

Data Collection: The data for the simulation program has to match with the real system to get some results. Input analysis is one of the core area on which system will give desired output.

The simulation model tries to create workload to invoke overload protection mechanism to see its working. The workload has to follow some norms of the system in order to get correct output. To simulate behaviour of the current system we can add few about number of times event takes place and time it takes for processing an event. For that period we can create workload and make system busy.

The data collection stage comprises of four sub stages and they are:

1. Find input data
2. Probability distribution to represent the input process
3. Based on distribution find parameters
4. Test the parameters selected

To get raw information directly from the system is a difficult task and was not available. As the primary purpose is not to completely represent the system working it is required to represent it partially. The available data was the service time and arrival time was the available data.

The data that was available had already undergone above given stages, as probability distribution of data was reported to be Poisson distribution. $((e^{-\alpha} \alpha^{x})/x!)$. As frequency of raw data is not known and sample data used for finding this distribution parameters could not be tested for goodness of fit using chi-square $(\sum_{i=1}^{K}(O_i-E_i)^2/E_i^2)$.

Following this distribution of the input data given in form of functionality, execution process take, intensity per second, and how much load is observed.

Functionality name	*Execution time*	*Intensity per second*	*CPU load*
Radio link setup (RLS)	5.5	61	33.3%
Power balancing (PB)	.056	286	16%
Radio link release (RLR)	2.0	61	12.1%

***Model Translation**:* The model was prepared in order to check system working through simulator. This model tries to follow software architecture procedure followed in the real system.

To match with the system requirements the simulator capsules in RoseRT were divided into four parts as shown in Figure 5.3.

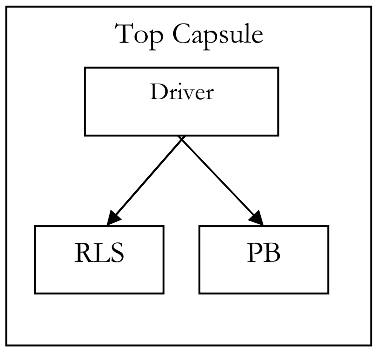

Figure 5.3: Structure of simulation program developed under RoseRT

1. Top Capsule: The main capsule of the program from where the whole program begins. It encompasses of all the capsules used in the simulation program.
2. Driver: This capsule is the main coordinating capsule of the whole program. This capsule is responsible for starting Load Control Server and Load Control Response Server as OSE processes; it is also responsible for sending messages to RLS and PB capsule in asynchronous way.
3. RLS: This capsule represents radio link setup function. The service time of this capsule is 5.5 m/sec, which is based on the worst-case scenario timing of actual RBS.
4. PB: This capsule represents the power balancing function of RBS and runs for 0.2 m/sec in the simulation program.

The development part was carried out in RoseRT environment. The various configurations were made to enable running of application in OSE like configuring osemain.con, softose.con, heap.con, and build.spec to enable application run-able in OSE. All these files play a pivotal role in execution of program on soft target (Solaris) environment and target environment (RBS).

The RLS and PB capsules had code defined in their transition and are invoked on receipt of message from driver capsule, followed by messages that are transferred through ports, and syntax for messages sent though protocols. There are two possibilities of message passing between capsules. It can be done either through send or invoke functions. Invoke is used when message passing is asynchronous and send is used when message passing is synchronous.

The driver capsule sends signal to RLS and PB capsules. PB capsules and RLS capsules both have a service time based on worst case scenarios time intervals presented in section 4.1. The inter-arrival time is changed in order to create high workload. For the message received from the driver capsule, message is sent after completion of the activity in the capsule. Driver capsule randomly selects one of the capsules as in traffic application.

The driver capsule has functionality to start the OSE processes. First the Load Control Response Server starts and then Load Control Server. The Load Control Server process sends the message to Load Control Response Server process and waits for the response based on the supervision time. If time elapses it sends overload protection signal to the driver capsule. The capsule activates overload protection mechanism by stopping workload created by RLS capsule and waits till the Load Control Server process does not send the signal back for ceasing overload protection.

Validate: A model is validated in order to check whether right model is build. Model should be able to demonstrate that it matches with real system for achieving its purpose.

The inference if made from the simulation program should give correct results. In this model to get behaviour of the system it is simulated based on the service time and inter-arrival time similar to the target system. The validity that can be produced for this model is structural validity as model developed models behaviour similar to the produced by subpart of the real system.

In comparison to simulator, real system also has task divided into capsules and maximum time they can take is based on worst case scenarios time. This is also present in simulator as task are divided into capsule and capsule communicates through messages with each other, after receiving message capsule performs some task to create workload in the system which is similar to maximum time task can take in worst case scenario situation. In real system time is spent in executing code and communicating with other physical entities. Instead of this in simulator capsules create high workload to represent similar workload as real application.

When task arrives in the real system, OSE schedules specific controller and in simulator model it happens when task arrives in the system it accepts task and then capsule allocates task to specific capsule though it cannot be done in OSE but it is done in RoseRT though it not similar to how real application works but as it involves moving from one capsule and this involves moving from one thread to another and this is similar in real application.

The results received from the model shows the behaviour of model is similar to what occurs in that real system. At start OSE processes representing overload protection are first instantiated from RoseRT model and then control passes to application running in RoseRT like it happens in real system, and load created by simulator keeps system busy.

But as testing of simulation program was done on Solaris which is used for development purpose and SoftOSE is used, which differs from HardOSE which is the one used in real system. Some behaviour differed like control could not pass from RoseRT environment to OSE as control can pass only from RoseRT to OSE when there is system call. To facilitate this system call was used in overload protection process and it worked. But system call is not used in overload protection method in real system but as it uses prioritized process and it creates interrupt unlike SoftOSE where this behaviour is lacking.

So it shows that simulator represents functionality similar to how overload protection works with real traffic application. Simulator purpose might be just limited to create workload but it does in a way that it matches in functionality of real application.

Verification: It tries to address whether model build is right and has an ability to demonstrate correctness according to specification, rules, formalizations, and constraints.

Verification is done through transforming model from one form to another with sufficient accuracy. This is verified through pair wise consistency with OSE and RoseRT model. The model purpose is basically to create workload based on some parameters and if you change parameters workload produced will differ. The activity runs for fixed number of times as possible in worst case scenario and if you change that parameters activity will run for amount of time that does not matches with traffic application.

The model working along with overload protection process is a simple simulator whose parameters that can be changed are number of times they can run, a workload that they create, and how much time do they spent on their task.

So it is verified that model can be transformed into other forms with change of above parameters.

5.3 Simulation experiment and analysis

This is where the model developed is run, tested to check whether it meets its purpose, analysis of the produced results, whether simulation achieved its purpose, and how many runs will be required to check desired system property from the simulation program. This stage comprises of following steps:

1. Production runs and analysis
2. More runs

Production runs and analysis: The testing for the simulation program was done exclusively on the SoftOSE platform provided by the Ericsson. As previously stated OSE comes under two versions one is soft target based and other is hard target based. The major difference between these two platforms is that in soft target there is no provision for pre-emption except for system calls. So the results produced from the targets vary a lot, but as testing on hard target requires lot of configuration and requires lot of resources, it was exempted.

The problem found in soft target was that during execution after initialisation of the client and server, RoseRT environment executes without pre-emption and after completion of the activities in RoseRT, Load Control Server and Load Control Response Server communicates. To solve this issue system call was added in overload protection process in real application to pre-empt and code was added to ensure that OSE processes start before RoseRT as sometime OSE processes are not even started when RoseRT starts executing. This was solved by adding delay before real application creating necessary interrupt needed for OSE process to start before simulator.

Another issue was that the loop was not putting efficient load on the processor. To put more load on the processor an array multiplication with a bigger array size was used. It differed a bit with service time but created enough workload to check working of overload protection.

The results observed were that first OSE process start then followed by traffic application and control does not leave RoseRT till all the activities are completed this happens when workload is high, if workload is reduced then at set interval period the control passes from real application. This behaviour shows that if activity is high in the system, simulation program does not leave RoseRT till workload is reduced. Action has to be taken either in RoseRT or OSE as coordination between two does not seem to work well during overload situation.

So based on the results that were achieved and understanding of the behaviour of RoseRT and OSE it is clear that if the signal is sent by Load Control Server, it will not be received as the signal will be delivered to the OSE message queue and will only be extracted from the RoseRT message queue only when no messages are left under situation of overload.

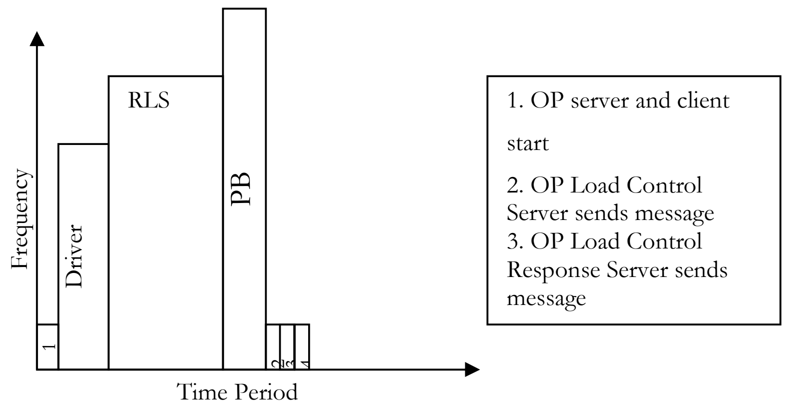

Figure 5.4: Showing the time period and frequency of various capsules collected through traces

The Figure 5.4 shows what the results were obtained from the simulation program. First overload protection starts and then it is followed by the Driver capsule, then by RLS, and then by PB, after that the OP starts.

So the results of the simulation clearly shows that the OSE co-ordination with RoseRT because of multiple queues, the way controller works, and workload present in the system it is difficult to send signal from OSE to RoseRT under system that has only 25% of resources available under overload situation.

More runs: Every simulation program has its purpose. In stochastic models running of models gives different results. As the purpose of simulation program is satisfied by the results obtained so far, they are not carried further and investigated.

5.4 Documentation and implementation of the results

In this stage the results of the system designed are documented and what action needed for correcting the implementation issues is carried out.

As the results expected out of the system was evident it was decided not to carry on the simulation process further. During start of thesis working of RoseRT and OSE was not that clear but by mid of thesis it was quite evident how it works and thus need for simulating program main purpose was lost.

Simulation results showed that the overload protection results are achieved when load is low but does not work when load is high. These results are similar to what results were observed regarding overload protection but change was done in overload protection code to create system call explicitly compared to real system where pre-emption happens implicitly. This aid was given to me by one of the employee in Ericsson which proved to be very valuable tip and helped in achieving satisfactory results.

6. Solutions

The present overload protection mechanism needs refinements to meet the ITU.T standard for overload protection. This part presents different ways of controlling overload protection through means of scheduling, RoseRT controller, and through other ways of implementing overload protection. Processor utilization measurement is covered in one section, as it produces a value based on which overload protection mechanism can take action.

The overload situation is directly related to the workload in the system, if workload is high then overload situation will arise. The workload in UMTS RBS compared to previous generation RBS differs due to increase in services it offers. The factors that can lead to overload situation previously presented in part 4 are there but the additional workload are caused by complex services added like intra-handover facility and handling of both circuit switched and packet switched data leads to increase in workload. .

In overload situation, overload protection helps a lot in reducing workload by provision of resources in a control way, as under overload situation only *25% of system resources are available* [1,2,3] and scheduling algorithms such as RMS can guarantee only if *the processor utilization is less than 69.5%* [44] and if utilization exceeds deadlines will be missed. Avoiding overload situation is very critical in providing QoS to the users but it is not justifiable with the high cost that will incur from the implementation proposal presented in section 4.1 as chances of worst-case scenario happening are very rare.

Overload protection mechanism can provide a good solution for controlling overload situation. But overload protection does not only have to provide mechanism for controlling overload protection, it has also to provide some mechanism for controlling higher utilization. The various situations which overload protection has to deal with are:

1. Normal situation: This situation is when processor utilization is less than 50%. DM, TA, and O&M activities run together in this situation, there is no action taken by overload protection mechanism in this situation.
2. High utilization situation: It has processor utilization between 50% and 70%. Overload protection takes some action in this situation to reduce workload in the system. In this situation both DM and TA can carry on their task. As per ITU.T recommendation

throughput by 10% should be reduced in this situation, action overload protection mechanism can take is to block O&M activities to reduce workload by 10%.

3. Overload situation: This situation arises when utilization has gone above 70% of the processor utilization. According to ITU.T recommendation for this step is to stop new link setup function until workload is reduced. The action overload protection can take in this situation is to stop most time consuming activities like RLS till workload is reduced.

The action described to be undertaken by overload protection is based on my observations about system that which activities can help in reducing overload. The main parameters that are considered are the deadline of the activity; inter arrival time of the activity, service time of the activity, priority, and importance of activity in the system.

The last two situations are where actual overload protection action takes place for controlling throughput. The important issue pertaining to these situations is **how** and **when** to take action. When to take action depends on the numerical value attained from processor utilization measurement and how to take action depends on the methodology followed for reducing throughput through overload protection mechanism or through use of scheduling.

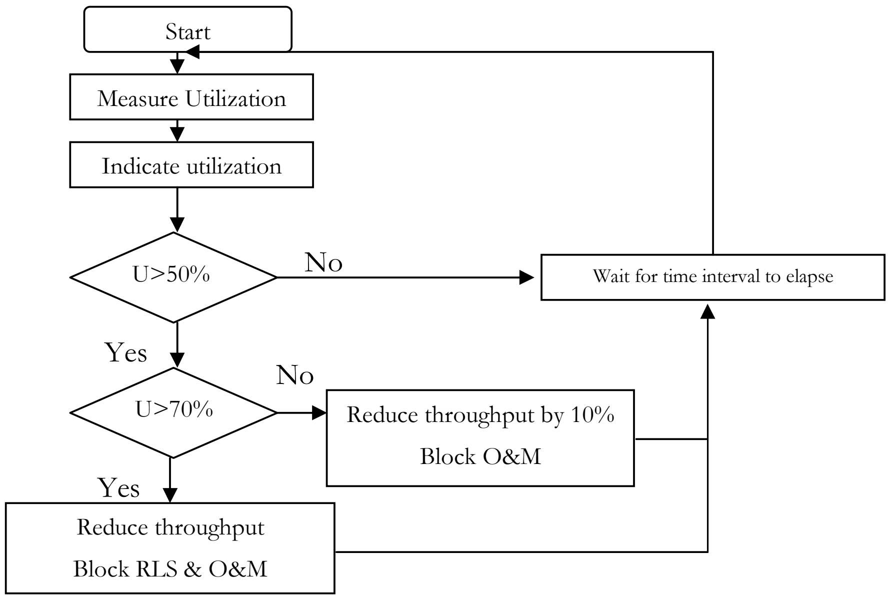

Figure 6.1: Flowchart of action taken by overload protection mechanism

The steps that can be undertaken by overload protection are showed through flowchart diagram in figure 6.1. The steps in flowchart allow overload protection mechanism to control overload situation in a controlled way. To summarize steps undertaken in implementing overload protection are:

1. Some means through which processor utilization can be measured.
2. Indicate of load on continuous basis in form of numerical value to decide whether it is higher utilization or overload situation.
3. If utilization is more than 50%, reduce throughput by 10% by blocking O&M activities.
4. If utilization is found to be more than 70%, stop RLS activities till workload is reduced in the system.

Above steps are undertaken in implementing overload protection mechanism when overload condition arises. After detection of processor utilization, action taken by overload protection mechanism is to either control throughput or stop new radio links or take no action.

The major step in deciding how to control overload situation is through processor utilization measurement and this can be done through counting the number of jobs that are offered to the system in comparison to how many processes it actually executes, measuring processor ticks, event driven measurements, sampling, or through CPI net execution time.

The most important thing is that after getting utilization numerical value, overload protection methodology to take action after overload situation detection has to comply with the system working. As starvation theory is good means of measurement but lacks precision and will be appropriate only if the scheduler for activities in the system would have been OSE but as there is an added scheduling by RoseRT.

So the implementation of overload protection has to be looked from the perspective of how RoseRT controller and OSE work together. Solutions suggested for overload protection method is based on this parameter.

The various solutions presented for controlling overload protection are:

1. Different ways of measuring processor utilization are presented and implementation details of the implementation based on the system working.
2. Using RoseRT controller for providing scheduling to control overload situation, which is better than present FIFO scheduling and in some implementation, it can be used for doing traffic differentiation in overload situation.

3. Overload protection can be controlled using scheduling algorithm for controlling overload situation by reducing workload to enable activities to meet their deadline in the system.
4. Other different mechanisms that can be used for implementing overload protection mechanism.
5. At last solution for controlling buffer overflow problem due to synchronization message.

The next five sections elaborate above given steps undertaken in study of overload protection mechanism.

6.1 Processor Utilization Measurement

The working of overload protection is completely related to how precisely processor utilization is measured. The present processor utilization measurement mechanism used in overload protection is based on the use of timeout and utilization of processor is thus not convertible to some numerical figure. It cannot detect whether processor utilization is 50% or 70 % from the timeout mechanism used in present implementation.

Various general means of measuring processor utilization are covered in this section. Associated with processor utilization measurement is how much workload system has at a point of time. Processor utilization can help in performance analysis of the system and can be used for attracting service providers by showing processor utilization in numerical figures.

Measurement of utilization can be either based on coarse-grained measurement or fine-grained measurement. Means opted for measurement depends on the precision required. The processor utilization that we are going to consider in this section is:

1. Measuring number of ticks that the processor takes in execution of task
2. Execution time of a single process, event driven approach
3. Execution time of processes for a period of time, this method is called sampling.
4. Measuring clocks cycles per instruction (CPI) that processor takes in execution of the process.

The most precise measurement is obtained through fine-grained measurement, which can be done only in hardware but requires an extra tool for its measurement. Coarse-grained measurements are done in software and are not that precise as fine-grained measurements. The measurements done can be either for a single process or for overall workload in the system.

One of the means of measuring utilization is through execution time, which is a coarse-grained technique. The execution time is measured from the time the activity started till the activity is completed. It will include time taken for I/O activities, memory access time, and time taken by OS for executing a task.

Execution time can be measured by creating workload through synthetic program (considers some application operations and their frequency), toy benchmark, kernel where some part of application is executed, or through execution of real application. The workload that is appropriate for checking processor utilization for overload protection has to be done through execution of real program. As processor utilization measurement has to be undertaken every set time interval and this can be done only when real application is running.

As overload protection action has to be taken based on some set time interval, the results from fine grained for such interval will be quite complex and is of not much help, so coarse- grained based processor utilization techniques are presented below.

1. Measuring processor ticks: The ticks are measured when processor is idle and if processor is busy it will report processor is busy. [8] The processor ticks that are measured require access to one of the processor pins to measure processor ticks. In OSE get_ticks can be used since system start and get_systime can be used for number of ticks since system start and the number of microseconds since the last tick.

2. Event driven approach: To measure processor utilization is done through event driven approach, where the time event started and time at which it stops is recorded. [11] Then based on the difference how much time system was busy is calculated. It does not differentiate between time taken for I/O, memory, and OS.

   The code outline presented below shows how processor ticks can be used for measuring processor utilization for a single function. When processor starts the time from the start of the system is noted down and on completion of call then again the ticks are stored. Difference between these two values gives how much time process took in executing the process.

```
// Event driven approach in OSE for measuring processor
utilization for one event
#include “ose.h”
OS_PROCESS (my_process)
{
   OSTICK tick1, tick2, store;
   for(;;){
         tick1=get_ticks(); // number of ticks before start of
   the process
         f1();
         tick2=get_ticks(); //number of ticks after end of the
   process
         store=(tick2-tick1); //processor utilization
      }
   }
```

3. Sampling: Processor load measurement is done through sampling, where processor utilization is measured by the time taken by NULL processes which runs at lower priority for set time interval. The difference between set time interval and NULL process gives processor utilization. The result tells about how much processor utilization was observed for the period. The sampling based approach can be used in OSE through use of timer-interrupt process.

   Implementation details of how sampling can be used for measuring processor performance are covered in code below. Code illustrates that first timer interrupt process is created that invokes every 100ms and creates an interrupt. On its invocation it checks for signal from NULL process. Signal specifies how much processor utilization is taken by NULL process. Based on set time interval and idle time we calculate total time the processor utilization takes by calculating processor busy time from the interval period.

```
//Sampling based approach:
#include “ose.h”
extern OSENTRYPOINT new_process;
OS_PROCESS(my_process)
{
   PROCESS proc_;
   for(;;){
```

```
   int number_of_tasks, utilization;
   int time_period;
   proc_=create_process(OS_TI_PROC,    “utilization”,    new_process,
(OSTIME) 100, (PROCESS) 0,(struct OS_redir_entry *) NULL), (OSVECTOR)
0, (OSUSER) 0);
      start(proc_);
      }
   }
   OS_PROCESS(new_process){
   UNION SIGNAL *rec_sig;
   rec_sig=receive_w_tmo(100, any_signal);
   if(rec_sig->sigNo==IDLE_TASK)
      idle_task=idle;
   time=period-idle/period*100;
   printf(“processor utilization: %d”,time);
   }
```

4. Clock per Instructions (CPI): This implementation is usually determined based on the system architecture, and system organization. To measure CPI it is required to know the path length of Instruction Set Architecture (ISA) for a particular processor. The execution time can then be achieved by multiplying path length of ISA with seconds per cycle processor takes. [13,9] This gives precise information about how many instructions processor is executing in a given period of time. This is considered to be one of the most effective ways of measuring utilization but the information produced in precision in this method is of not that much great use in detection of overload protection.

As overload protection involves measuring processor utilization every 100 ms the best approach that suits our purpose is the sampling based measurement based on processor ticks. The code snippet is given and can be implemented in each controller or as a process in OSE to measure processor utilization. We will cover various ways of implementing overload protection and each method involves different processor utilization mechanism.

Based on processor utilization mechanism, action can be taken either by overload protection mechanism process, by scheduler, and RoseRT controller. Note that the results produced are estimation, as we take average of the execution time of the process.

6.2 RoseRT custom controller:

RoseRT eases a lot in software development activity of a complex system such as RBS. It has been recently concluded from work done on study of RoseRT in Telecom systems that automatic code generated by RoseRT is of high quality. The problem that can arise is the scheduling of capsules by RoseRT controller and fine-grained software development practice followed in developing software in RoseRT.

The other issue is that although code generated by RoseRT is highly optimised but how to compose C++ code for UML diagram in optimised way is area of research. The code that is added by the developer in between transition in state chart diagrams can spoil the optimised code produced by RoseRT. The messages that are used in RoseRT rely on system behaviour and cannot be verified formally.

As shown in figure 6.2, OSE process to communicate with RoseRT has to send a signal, as communication in OSE takes place through means of signals. Signal is then converted to RoseRT message by RoseRT controller, as messages are used for communication in RoseRT. The OSE signal received in RoseRT controller's message queue will be executed only when RoseRT message queue is empty.

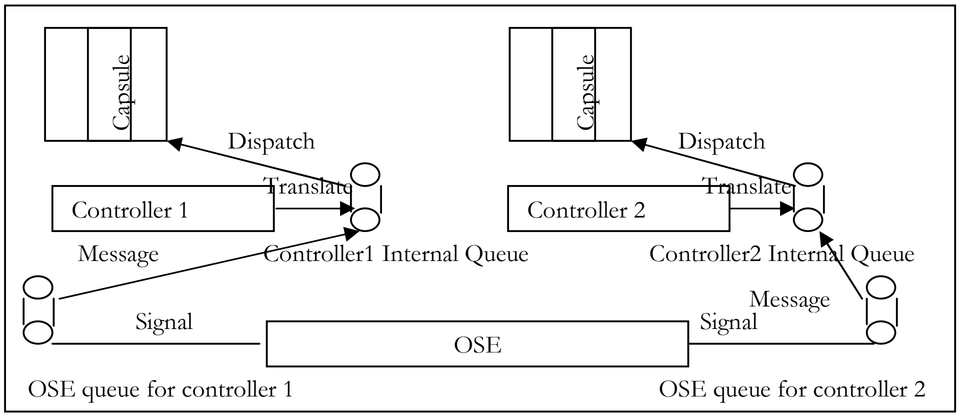
Capsule
Dispatch
Translate
Controller 1
Message
Controller1 Internal Queue
Signal
Capsule
Dispatch
Translate
Controller 2
Controller2 Internal Queue
Message
Signal
OSE
OSE queue for controller 1
OSE queue for controller 2

Figure 6.2: The flow of message between RoseRT message queue and OSE message queue [20]

The highly prioritized signal such as overload protection has to wait till message will be read from the queue. This delay introduced can bring in big difference as under situation of overload if RoseRT schedules RLS activity it will make task of scheduler more difficult and can lead to worse outcomes. This scheduling of tasks by both OSE and RoseRT makes scheduling bit complicated to achieve desired outcomes.

The implementation of the controller used in Ericsson's RBS is RTCustomController in comparison to RTPeerController and RTSoleController provided by RoseRT. RTSoleController is provided for single threaded application and RTPeerController is used for multi threading applications. RoseRT provides RTCustomController but Ericsson has implemented their own as the changes in custom controller are only done related to the cello platform needs and requirements.

As working of both OSE and RoseRT Controller in terms of scheduling and performing necessary action is known, we can use RoseRT mechanism in performing activities to control overload protection such as:

1. Perform traffic differentiation function
2. Controlling flow of synchronization message to OSE
3. Use dynamic scheduling for scheduling capsules based on messages received from OSE.

For implementation of traffic differentiation, controller can be used for checking if message is related to O&M block it if load is less than 70 but greater than 50 and if load is found to be more than 70 then RLS connections are blocked for a period of time. The system calls used and way it can be implemented is covered in section 6.4.

If controller is used for controlling synchronization message, only one message related to the controller is passed to the OSE message queue of the controller and rest of the synchronization messages received for particular controller is discarded. More detail about this implementation in custom controller is covered in section 6.5.

Scheduling implementation of RoseRT is based on FIFO. Real time system like RBS needs task to be done on time and this can be modified through making task schedulable in RoseRT based on dynamic scheduling or based on overload scheduling as presented in section 6.3.

6.3 Process scheduling:

Systems that are not overloaded, scheduling can be achieved through various scheduling algorithms such as Rate Monotonic Scheduling (RMS), Earliest Deadline First (EDF), and Deadline scheduling. But in overload situation RMS scheduling fails and cannot schedule activities when processor utilization is more than 69.5%. [44] In terms of resource availability, only 25% of resources are available [1,2,3] making task of scheduler very complicated and difficult.

The scheduler that is required in RBS has to be one that can effectively take measures in overload situations and also schedule activities efficiently in normal situations.

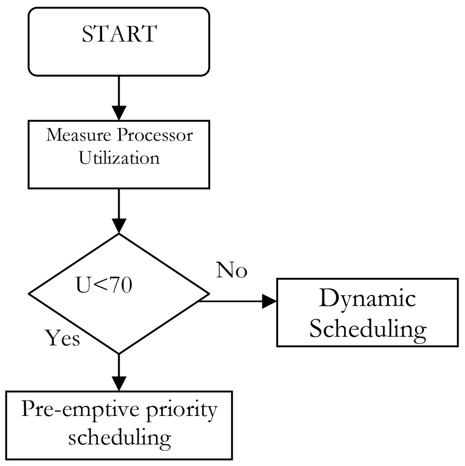
START
Measure Processor Utilization
U<70
No
Yes
Dynamic Scheduling
Pre-emptive priority scheduling

Figure 6.3: Flowchart diagram for steps that scheduler needs to have for controlling overload protection situation.

As shown in Figure 6.3, scheduling mechanism takes an activity and sees if the processor utilization is greater than 70 or less than it. If utilization found is less than 70 then it continues with pre-emptive priority scheduling and if utilization is found to be greater than 70 then dynamic scheduling is used. The advantage of using dynamic scheduling over pre-emptive scheduling is that dynamic scheduling schedules processes based on runtime parameters not on priority defined at development.

Dynamic scheduling can schedule processes in much better way under overload situation. It schedules based on deadline and execution time required for the processes. [48] As dynamic scheduling is not only based on execution time but also on deadline, scheduling algorithm will arrange execution of only TA functions as O&M functions in spite of having low execution time will have higher deadline compared to TA. Scheduler will continuously reshuffle the queue, order them, and allow some new task to be added as a run-able task, the effective priority of a task is in constant flux in dynamic scheduling.

To properly schedule in an overload situation Sanjoy and Jayant [1,2,3] divides scheduling of overload protection into two domains. One domain is based on how much processor utilization does activity takes and other is based on process deadlines. During overload situation the tasks should be handled efficiently by allowing task that has nearest deadline to complete but only those activities that are already in the ready state. The tasks present in the ready state must be able to complete its operations in order to enable other tasks to complete their work.

Dynamic scheduling is a form of scheduling that is based on deadlines of the processes and in that way resembles method suggested by Sanjoy but the difference is that without dividing it in two domains we can schedule TA activities that have nearest deadline and short execution time. If we implement this scheduling in OSE when processor utilization is found high then RLS activities will not be scheduled as they have nearest deadline but long execution time.

The scheduling in RoseRT is based on FIFO but if it based on dynamic scheduling, task will be scheduled based on their deadline and execution time, making a big difference in the system performance in overload situation. As task will be scheduled based on the deadline and execution time, RLS activity will be stopped and will lead to reduce in throughput as required by the overload protection.

But care should be taken that the dynamic scheduling is used only when load is found in the system otherwise it can lead to severe consequences that RLS is never allowed because of high execution time. In normal situation if it uses RMS where task will be scheduled based on some priorities it will make a big difference compared to FIFO implementation. There is of course a scheduling overhead but proper scheduling can give better results.

Both scheduling suggests ways of dealing with overload situation and this can bring good results in scheduling system in effective way. But the problem is that scheduling implementation like such requires changes in OSE. This scheduling can be recommended to ENEA to be included for better control of overload protection in the future release. As scheduling cannot be implemented without OSE, we can look for other ways of controlling overload protection without disturbing scheduler.

6.4 Ways of better implementing overload protection:

This section presents three ways of controlling overload protection. First method tries to use custom controller for detecting processor utilization and then based on the utilization takes same action as present mechanism, change parameters in NBAP handler for new radio link setup. Second method measures processor utilization through a processor utilization measurement process (PUMP) in OSE and implements overload protection through traffic differentiation through an OSE process, the idea behind this method is that TA does not receive RLS so there will be no chance of it getting scheduled in RoseRT. Last method uses PUMP for measuring processor utilization but custom controller handles traffic differentiation and does task of reducing throughput.

First approach:

This approach modifies way of detecting processor utilization but action taken to reduce workload is same like in the present system. The approach uses BCNM capsule as the base capsule that handles processor utilization function through aid of custom controllers and turns off the parameter of NBAP when overload situation is detected.

Each controller has a timer interrupt process that invokes every set time period and sends information about processor utilization to BCNM capsule. To measure processor utilization by controller it uses event driven scheduling where each controller keeps accounts of total time taken for controller to complete its task. On interruption from timer interrupt process message is sent to BCNM capsule.

BCNM on receipt of message from controller calculates utilization as the time taken by all controllers in set time interval by total set interval period

$$\text{Utilization} = \frac{\text{time taken by all controllers}}{\text{total set time interval}} * 100\%$$

If utilization is found to be less than 70 no action is taken and if the utilization is found to be greater than 70% same way of reduction of workload is followed as present mechanism. The action taken is to change OnOff parameter of NBAP controller, which NBAP checks every time before setting new RLS activity.

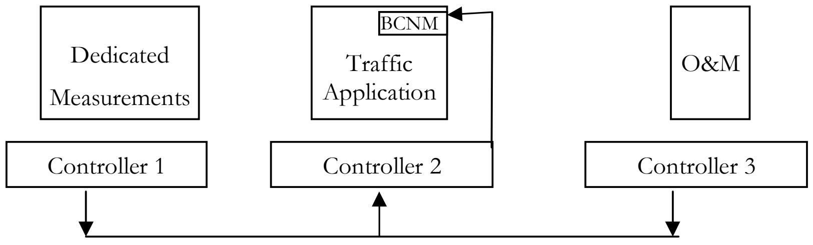

Figure 6.4: Showing synchronization message sent by each controller after elapse of the set time interval period to BCNM capsule

As shown in Figure 6.4, each controller records time interval it starts executing and stopped executing this time is sent as a message to BCNM capsule, if it did note execute then it sends 0 to BCNM. Each controller sends synchronization message to the TA controller that has BCNM capsule. As majority of task performed is in TA custom controller, OSE will schedule TA controller and it will lead to an action taken by BCNM to either control overload protection or allow new RLS to be undertaken. If TA controller is already a running process then message will be delivered to BCNM capsule immediately.

This methodology tries to follow same way of controlling throughput by stopping NBAP for new RLS connections. But the change achieved is that utilization is achieved in numerical value and not on the timeout mechanism and all activities related is carried out in RoseRT environment and does not involve messages from OSE like the present system. If TA controller is not running then DM controller and O&M controller will have to send synchronization message to make TA controller schedulable and if TA is running it has to just inter capsule message to BCNM capsule.

The advantage from this approach is that only one of the scheduling mechanisms will be used for overload protection and does not have to wait in OSE message queue of the controller to inform BCNM about the overload protection as the present mechanism. It does not have to rely on two message queues for the message to be delivered to the BCNM capsule.

The major problem that is present in this methodology is that it considers processor load created by application in RoseRT and does not consider load created by OSE processes. As major work of RBS is spent in performing RoseRT, this method in spite of not giving correct processor utilization can still give reasonable results.

The second problem is how to handle latest RLS request. As there is no traffic differentiation, RLS still enters in RoseRT environment and are scheduled. The third issue is that it requires synchronous message from each controller to BCNM at set time interval if TA controller is not running. This may involve context switch between processes running in OSE, as controller is after all process in OSE. The fourth issue is how to control when load is found to be 50% and this can be done through sending message to OSE to intercept custom controller responsible for O&M but it destroys layered architecture principles that top layer contacts with low layers to perform some action.

Second approach:

This approach follows different approach to detect processor utilization and also action that is taken for overload protection is completely different from above approach. It does not try to involve RoseRT as it involves RoseRT scheduling and checking for NBAP parameter before establishing link with RNC for new connection. If overload situation is detected action is undertaken in OSE, as it will not involve relying on RoseRT message queue to pick up the overload message and take some action. The other advantage is that RLS activities are killed before they enter RoseRT environment making system free of RLS in overload situation.

It implements processor utilization measurement process (PUMP) in OSE that detects processor utilization for set time interval, and based on processor load it starts traffic differentiation if overload situation is found in order to stop RLS and in high utilization situation PUMP intercepts O&M activities for a period till workload is reduced in the system. This method does not try to change OnOff parameter of NBAP but tries to stop RLS request before entering RoseRT environment.

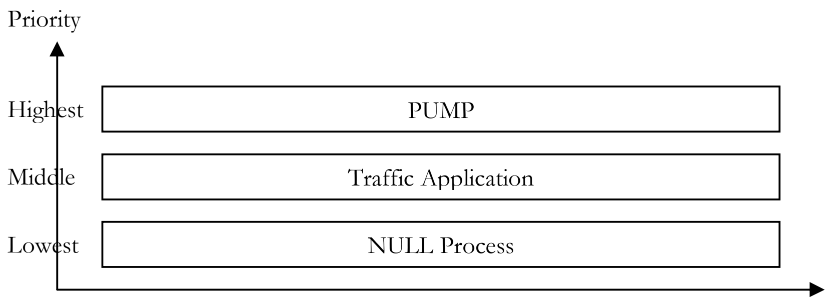

Figure 6.5: Showing processes running at their priority levels.

As shown in figure 6.5, this methodology divides processes in three priorities. At highest priority there is PUMP, which is a timer-interrupt process. At lowest priority there is NULL process that is responsible for measuring the time interval it could execute in set time interval. Highest priority process on its invocation tries to seek signal from NULL process and if does not receive any signal it assumes NULL process could not execute in that interval.

The time interval utilization is measured by time not taken by NULL process in the time interval set by total sampling period. The result shows much processor load was observed for that period.

$$\text{Utilization} = \frac{(\text{Sample time interval} - \text{idle task})}{\text{Sample time interval}} * 100$$

The higher-level priority process uses hunt_from function of OSE if it detects utilization to be greater than 50% or 70%. If utilization is observed to be 50% then it hunts for O&M signals and intercepts them, activities related to O&M are like background task and can be delayed, as they do not have real time system requirements.

If load is observed to be 70% then it kills RLS function after sending reject signal to RNC and keeps an account of reject counter for each link dropped, this is done for O&M purposes. In next sampling period if load is found to be less than 70 then RLS activities continue to work as before. But if load is not reduced and shows utilization of more than 50 but less than 70 then O&M activities are still intercepted. If load is found to be less than 50% then no action is undertaken.

Actions undertaken when high utilization situation is found are as follows:

1. Processor utilization measurement measured. Based on the measurement if it is greater than 50 and less than 70, hunt function of OSE is used to search for name specified and if it finds process with that name it returns it into name_ parameter. Following is the syntax call that will be used for searching for process:

   ```
   OSBOOLEAN hunt(char *name, OSUSER user, PROCESS *name_, union SIGNAL
   **hunt_sig);
   ```
2. Then intercept function is used that will change the status of process or block of processes as intercepted. Intercept function creates a kind of breakpoint for the process. It uses process ID to intercept a particular process or block of processes. The syntax of intercept is as follows:

   ```
   void intercept (PROCESS pid);
   ```

3. When the load is found less than 50% then the continuation of service are specified through resume call. The reason for resuming activities is that O&M activity if delayed can still be carried out as they do not real time requirements. The syntax of resume function call.
   `void resume(PROCESS pid);`

The code snipped in OSE for taking action specified above:

```
PROCESS hunt;
union SIGNAL huntsignal;
OSBOOLEAN check;
for(;;){
// processor utilization is measure through sampling method described in
section 6.1
if(u>50 and u<70){
      check=hunt(“O_M”,0, &hunt, &huntsignal);
      huntsignal=receive((SIGSELECT *)any_sig);
      intercept(huntsignal);
      }
}
```

The step that has to be undertaken when overload situation is found is bit complicated compared to O&M control.

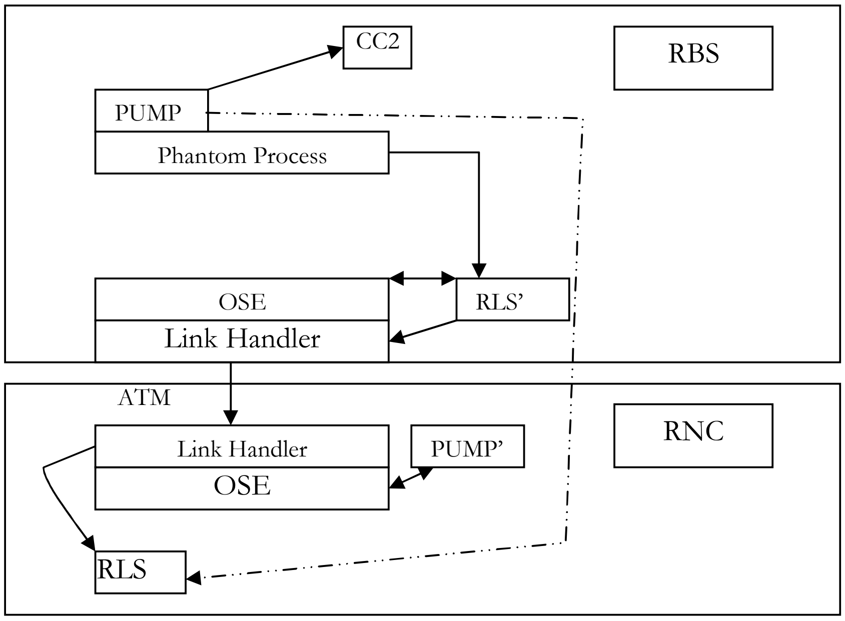

Figure 6.6: Flow of control of messages in OSE through link handler to hunt for RLS signals

The figure 6.6 shows the flow of messages that take place when PUMP is used for measuring processor utilization and taking action to control throughput and stop radio link set-up. As illustrated in figure following actions are undertaken under situation of overload:

1. PUMP calculates processor utilization in numerical value, and then checks whether utilization is greater than 70. If utilization is greater than 70, a phantom process is started which runs along with PUMP and sends all RLS signals redirected to it before being delivered to TA custom controller. Phantom process of RLS is required as it creates replica for enabling redirection table. As phantom process creates a redirection table and it is used for redirecting RLS signal found through link handler. Link handler helps in hunting for RLS messages and provides a logical map between two processes in two different target environments, RBS and RNC.

```
#include “ose.h”
extern OSENTRYPOINT new_process;
OS_PROCESS(my_process)
{
      PROCESS proc_;
      static const SIGSELECT rls_sig={12}; //assume rls_sig is 12
      struct OS_redir_table redir_table[1];
      redir_table[0].sig=(SIGSELECT) rls_sig;
      redir_table[0].pid=rls_pid;//pid of the rls
      proc_=create_process(OS_PHANTOM, signal->remote_calls,
      rem_hunt.name, (OSENTRYPOINT *) 0, (OSADDRESS) 0, (OSPRIORITY) 0,
      (OSTIME)0, (PROCESS) 0,&redir_table, (OSVECTOR) 0, (OSUSER) 0);
      start(proc_);}
}
OS_PROCESS(PROC_){
if(U>70){
    sig->remote_call.rls.reject_signal;
    reject++;
     kill(rls _signal);
}
}
```

2. After hunting of RLS signals, they are redirected to process created by PUMP in order to kill RLS signal.
3. A reject signal is send to RNC and rejection counter is incremented

4. If load is reduced then still signal will be redirected through the phantom process created that does filtering of RLS signals only under overload situation and rest of the time allows flow of signals without hindrance.

Third approach:

In this approach PUMP measures processor utilization and instead of creating phantom process and redirecting all RLS signals through it, the controller does traffic differentiation and function of intercepting O&M signals. In above approach link handler, phantom process, and redirecting all signals have to be undertaken in order to avoid RLS signal from entering RoseRT, making an approach bit complicated.

Above approach misses the point that the signals already present in RoseRT are not stopped and if they get to execute they will create more workload in the system. So incoming signals in OSE are stopped though above approach but already accepted signals are not given any preference. This approach tries to address that issue and makes overload protection solution suitable for the target environment.

Each controller seeks for signal from PUMP process, TA controller seeks whether there is any overload signal and O&M controller seeks for utilization signal in system message queue. Each controller makes a system call to check for a signal and if signal is found it takes action accordingly.

The controller on receive of utilization signal performs action to intercept signal and if signal is overload it just discards the RLS signal before scheduling them and sends rejection signal to RNC. TA controller and O&M controller undertake the overload protection functions. The controllers are ongoing loop which read from controller queue but before it is read from the queue it makes a system call that will read from OSE queue and looks for particular signal, if found action related to overload protection is undertaken or action related to high utilization is undertaken by respective controllers and if not found then controller continues reading from its message queue.

PUMP role in this approach is to measure processor utilization and this is done based on the signals it receives from the NULL process. The result of the utilization measurements is in form of numerical values and this value is used for generating signal. If load generated is greater than

50% and less than 70%, then utilization signal is sent and if load is greater than 70 then overload signal is sent.

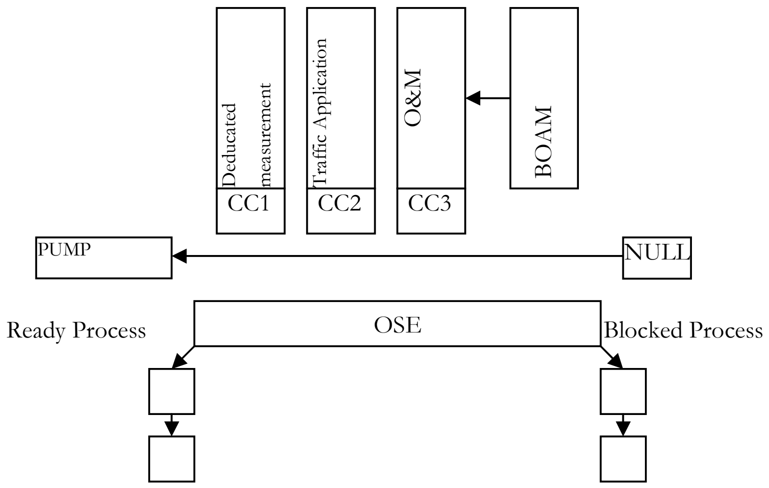

Figure 6.7: Shows PUMP, null process, and RoseRT controller working together

As shown in figure 6.7, OSE comprises of some processes that are blocked and some processes that are ready. The scheduler arranges execution of the ready processes. Every set interval period PUMP based on the inputs it receives from NULL process, it measure processor utilization and if there is no input from NULL process it considers utilization to be 100%. Based on the utilization numerical value, PUMP generates signals. The action taken on invoke of signal defers in both controllers.

Outline of an action taken by PUMP after finding utilization of the process:

```
//Code for processor utilization measurement process:
OSPROCESS(my_proc){
//utilization measurement is similar to sampling code presented in
section 6.1
if(u>50 and u>70)
      signal(utilization);
if (u<70)
      signal(overload);
}
```

The steps undertaken by O&M controller are as follows:

1. Check for signals at start of loop and take action accordingly.

2. If utilization signal is found then all the processes related to O&M are intercepted using intercept function call of OSE.
   void intercept (PROCESS pid);
3. If signal is not found then it performs read from its own message queue.

```
//outline of code for taking action in O&M controller
OS_PROCESS(my_proc){
for(;;){
      sig=receive_w_tmo(0,o_m); //read from OSE message queue
      if(sig==o_m)
            intercept(o_m);
      receive(any_sig);//read from its own message queue
      }
}
```

The action undertaken by TA controller is bit different from O&M controller as it does not involve intercepting signal but taking up several actions other than it. Following are the steps undertaken by TA:

1. Check for signals at start of loop and take action accordingly.
2. If overload signal is found then it sends rejection signal to RNC
3. It increments rejection counter for O&M purpose.
4. It kills signals that are related to RLS
5. If signal is not found then it performs read from its own message queue

```
//outline of code implementation in Custom Controller
OS_PROCESS(my_proc){
for(;;){
      sig=receive_w_tmo(rls_signal);
      if(sig==rls_signal){
            // send rejection signal to RNC
                sig->RNC.rls.reject_signal;
               reject++;
               kill(rls _signal);
            }
      receive(any_sig);
      }
}
```

The advantage of this approach is that it is easy to implement and can be done with small changes in RoseRT. The other advantage from this approach is that RLS signals are dropped before they enter traffic application. This method tries to reduce overload from the OSE and does not make it schedulable and thus freeing up the resource of RoseRT.

6.5 Synchronization Message Solution

RoseRT controllers are OSE processes and each controller are assigned based on what functions does it work and based on the functionality related to RBS it performs it is assigned priority. The functions of RBS are divided into dedicated measurement, traffic application, and O&M activities. For this each function there is a controller running as an OSE process.

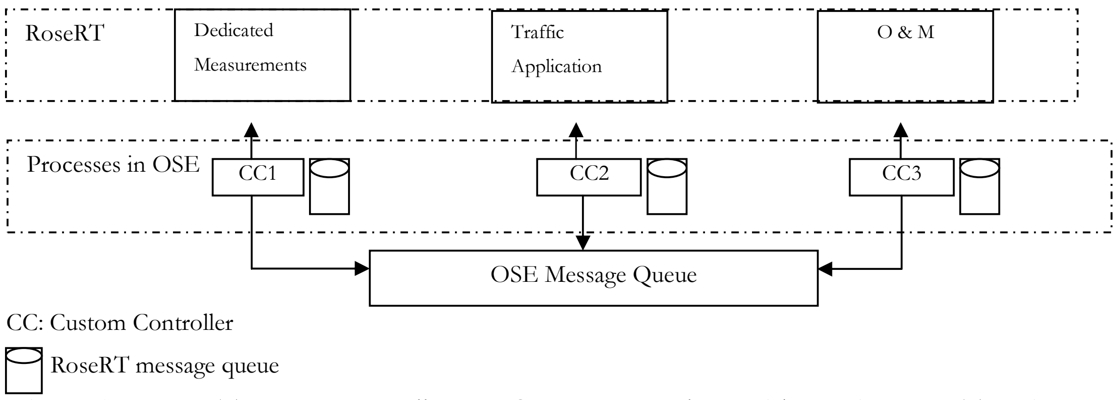

Figure 6.8: RoseRT custom controller as OSE processes along with part in RoseRT environment that controller handles

As shown in figure 6.8, the layered approach divides RoseRT environment in three logical parts with each logical part is controlled by its own controller. This allows software to follow layered approach and to take advantage of this methodology as activities related to a particular controller can be carried out separately from each controller. But the functionality requires sending messages to other controller and leads to lots of communication between these OSE processes. For example TA sends message related to O&M regarding number of miss calls during overload protection.

There is lot of message passing between these logical parts and that is usually done using inter controller messages and message sent between capsules. The scheduling of processes is done by OSE and to make scheduler arrange for execution of particular controller it has to be informed by RoseRT that message has been passed to other controller and makes other controller schedulable. To inform OSE to schedule each inter capsule message involves sending synchronization message to OSE. The reason for sending this message is just an indication to OSE to schedule other controller too.

The problem in this approach is that as TA is a fine-grained application. There is lot of interaction with other controllers' and these results in capsules sending synchronization message to OSE message queue. As capsule does not know whether other capsule have sent message to OSE, each inter capsule message involves synchronization message and sometimes leads to a buffer crash problem.

To solve above situation the best way is to filter out the messages before it reaches OSE message queue. Just one message is enough to indicate OSE scheduler to schedule respective RoseRT controller for the message sent as intra-thread message between other logical parts.

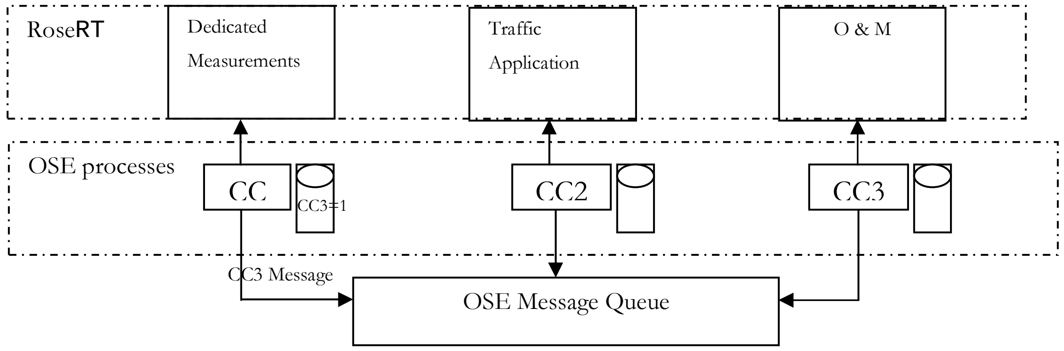

Figure 6.9: Controller taking up action as part of controlling synchronization messages

As shown in figure 6.9, to control OSE synchronization message, each RoseRT controller keeps a flag for other two controllers. After sending first message related to synchronization message it receives from any capsule in its environment it stops all other synchronization messages from reaching OSE message queue. To implement this functionality in RoseRT controller each RoseRT controller sets a flag for controller after message was sent. For example if CC1 gets synchronization message from some capsule then message will be passed to OSE message queue and if that message was for CC3 the flag is set for CC3. The next synchronization messages sent by any capsule to OSE for CC3 will be dropped.

Once the first synchronization message is sent from RoseRT to OSE, a flag is set to indicate that message has been sent. The flag that has been set needs to be changed every time another RoseRT controller is scheduled. So controller that is scheduled should inform after its complete its operation that it has completed its operation.

RoseRT controller thus can play a very pivotal role in controlling of overload protection as well in controlling of RoseRT message queues.

7. Conclusion

Wireless communication has gone through some changes and has some new facilities in providing new facilities to its users. The change in air interface requires new support of facilities from RBS and thus making development of such system more complex and some time create high workload on the system.

The components used in development of RBS plays a pivotal role in the performance that is achievable through the system when executed on main processor. The process is scheduled based on OSE priority driven scheduling. RoseRT provides a virtual environment to allow execution of a capsule. This virtual environment allows feasibility and interoperability in the development of the product.

Overload protection is a method devised to control the system from restart when system is having workload more than it can handle. The present mechanism is based on starvation theory and employs two process running in OSE that run along with RoseRT traffic application and responds to the overload situation by sending message to stop Radio Link Setup function.

The simulation program was used to check overload protection signal receiving by capsule under situation of overload. As the resources in the system under overload situation are scarce and only resources can be scheduled when processor load is 70% the receiving of message from OSE to the RoseRT is an issue.

As RoseRT controller reads from OSE message queue when it does not have any message left in its queues it delays message sent for stopping RLS. Overload protection method presently implemented does not take into account reducing of throughput by 10% when processor utilization is 50% it requires some better implementation.

The simulator was used to aid in understanding overload protection and results produced showed that the simulator shows how overload protection mechanism varies with load. The behaviour of the system shows that OSE and RoseRT cannot work together and control for overload protection should be in either OSE or RoseRT environment but to rely on both will produce such results.

The better implementation will allow the processor workload reduction under overloaded situation. The methods suggested about overload protection method need some implementation in cello platform and will require some modification to support overload protection. A proper way of traffic differentiation is needed to allow processor taking appropriate action when filtering traffic under overload situation.

The first step was to find processor utilization in some numerical value so that overload protection action can be taken in effective way. The sampling based method of processor utilization is appropriate for overload protection as load measurement has to be done for specific period of time.

Then based on processor utilization either we can take appropriate either in RoseRT controllers and BCNM capsule, or do both processor utilization and traffic differentiation in OSE, or do processor utilization in OSE and RoseRT controller checking continuously for overload protection signal.

The task of thesis to analyze, simulate, and present overload protection was done and solutions were suggested in such a way that their implementation can be carried out with little effort.s

8. References

[Research Papers and White Papers]

1. Sanjoy K. Baruah et.al., 1991, *On-Line Scheduling in the Presence of Overload,* IEEE, PP 100-110
2. Sanjoy K. Baruah et.al., 1994, *On-Line Scheduling to Maximize Task Completions,* IEEE, PP 228-236
3. Sanjoy K. Baruah et.al., 1997, *Scheduling for Overload in Real Time Systems,* IEEE Transactions on Computers Vol 46 No 9, PP 1034-1039
4. Pedro Mejia-Alvarez et.al, 2003, *An Incremental Server for Scheduling Overload Real-Time Systems,* IEEE, PP 1347-1361.
5. Swim et.al., 1995, *Avoiding Deadline Decay Under Transient Overload*, IEEE, PP 198 - 200
6. Risto Teittinen, 2003, *Mobile Internet Technical Architecture - Vol. 1 Technologies and Standardization*, Mobile SMITA Seminar WCDMA technology on UTRAN, (pp. 155 - 167).
7. Tomas Lingvall, 2002, *Load Control in a Radio Network Controller within UMTS*, Thesis Work, Linköping University.
8. Anders Ive, 2002, *Runtime performance evaluation of embedded software*, Department of Computer Science, Lund University
9. Byron Miller, 2002, *Determining Processor Utilization*, Dr Dobbs Jounrnal
10. Pradip Bose and Thomas M. Conte, 1998, *Performance Analysis and Its Impact on Design*, IEEE (pg 41-49)
11. David B. Stewart, 2001, *Measuring Execution Time and Real Time Performance*, Embedded Systems Conference San Francisco CA.
12. F.E. Levine and C.P. Roth, 1997, *Programmer's view of Performance monitoring in the PowerPC microprocessor*, IBM J. Res Develop Vol 41 No 3 (PP 345-356)
13. Emma, 2001, *Understanding some simple processor performance limits*, IBM J. Res Develop
14. Konstantin Popov and Karl Filip Faxen, 2005, *Analysing and Improving Performance of Telecom Applications in Rational Rose Real Time*, Ericsson
15. Lars Örjan et.al., 2002, *CPP – Cello Packet Platform*, Ericsson Review No.2, 2002, PP 68 – 74
16. Ericsson, *Basic Concepts of Radio Access Network*

17. Bengt Gestner and Bengt Persson, 2002, *Ericsson's first WCDMA radio network controller*, Ericsson
18. Terry Quatrani, 6 Nov 2003, *Introduction to UML*, Rational Software, IBM
19. Jonas Mellin, 2002, *OSE Delta*, Distributed Real Time System Research, University of Skövde
20. Olle Rosendahl, 2005, *Structure and Runtime Mechanism*, Ericsson
21. Friedmand and Waldbaum, 1975, *Evaluating System changes under uncontrolled workloads*, IBM System Journal, PP 345-352
22. Peter Ball, 1996, *Introduction to Discrete Event Simulation*, 2nd DYCOMANS workshop on Management and Control: Tools in Action PP 367-376
23. David Kalinsky, 2003, *Introduction to Real Time Operating System*, Enea Embedded Technology.
24. Manimaran and Murthy, 1998, *An Efficient Dynamic Scheduling Algorithm for Multiprocessor Real Time System*, IEEE Transactions on Parallel and Distributed System Vol 9 No 3, PP 312-319
25. Silcock and Kutti, 1991, *Taxonomy of Real Time Scheduling*, School of Computing and Mathematics, Deaking University
26. Agarwal et. al*., Measurement and Analysis of Process and Workload CPU times in Unix environments.*
27. Janssen and Graham, 2004, *Model Driven Development of Resource Constrained Embedded Applications*, IBM Developer Works (Rational)
28. Oehlerich et. al., 1995, *Load Control and Load Balancing in Switching Systems with Distributed Processor Architecture*, Siemens
29. Yufenz Zhao, 1999, *Derivations of Local Timing Constraints in Early Design Stages*, Master Thesis, Chalmers University of Technology.
30. Gary Nutt, 1975, *Computer System Monitors*, IEEE Computer Society, Performance Evaluation Review, PP 41-50
31. Colnaric and Veber, 2001, *Dealing with tasking overload in object oriented real-time application design*, IEEE, PP 214-220
32. Buttazo et.al., 1995, *Value vs. Deadline Scheduling in Overload Conditions*, IEEE, PP 90 – 99
33. Sneha et. al., Fast and Robust Signalling Overload Control, Bell Laboratories
34. Alvarez et.al., 2003, *An incremental server for Scheduling Overload Real Time Sys*tems, IEEE Transactions on Computers, Vol 52 No 10, PP 1347-1361
35. Åkerblad and Olsson, 2000, *Overload Protection*, Ericsson

36. Rajesh K. Gupta, 2002, *Embedded Systems*, University of California, Irvine

[Books]

37. Raj Jain, 1991, The art of Computer System Performance Analysis, John Wiley and Sons, ISBN: 0-471-50336-3.
38. Harri Holma and Antti Toskala, 2001, WCDMA for UMTS, John Wiley and Sons, ISBN:0 471 48687 6.
39. Deitel, 1990, Operating Systems, John Wiley and Sons, ISBN: 0201509393
40. John L Hennessey and David A Patterson, 1996, Computer Architecture: A Quantitative approach, Second Edition, Morgon Kaufmann, ISBN: 1558603298
41. Peter Marwedel, 2003, Embedded System Design, Kluwer Academic Publishers, ISBN: 1402076908
42. OSE, 2003, OSE Documentation Volume 1 Kernel
43. Peter Brucker, 1998, Scheduling algorithms, Second Edition, Springer, ISBN: 354064105x
44. Wayne Wolf, Computers as Components: Principles of Embedded Computer Systems Design, Morgan Kaufmanns

[Websites]

45. http://edition.cnn.com/2001/TECH/industry/10/22/3g.defined.idg/index.html,
46. Michael T. Trader, 2004, How to calculate CPU utilization, www.embedded.com/showArticle.jhtml?article=23900614
47. Peter Dibble, 2001, Deadline Scheduling, http://www.embedded.com/showArticle.jhtml?article=9900112
48. [Lecture Notes] CPU Scheduling, http://cs-alb-pc3.messey.ac.nz/notes/59305/mod5.html
49. Michael Barr, 2002, Introduction to Real Monotonic Scheduling, http://www.embedded.com/showArticle.jhtml?article=9900522
50. Srinivas Dharamasanam, 2003, Multiprocessing with Real Time Operating Systems, http://www.embedded.com/showArticle.jhtml?article=10700141
51. David B Stewart, 2001, Introduction to Real Time, http://www.embedded.com/showArticle.jhtml?article=9900353
52. Survey of Task Schedulers, www.kalinskyassociates.com/Wpaper3.html
53. www.ericsson.com/press/archive/ backgrounder/wcdma10000backgrounder.doc

Appendix A: UML

UML (Unified Modelling language) is the standard language for specifying, visualizing, constructing, and documenting all the artefacts of a software system. It provides abstraction at many layers in design process. UML is useful because it encourages design by successive refinement and progressively adding details of the design rather thinking about design at each new level of abstraction.

The structural diagrams in UML are class diagrams, collaboration diagrams, component diagrams, and deployment diagrams. The behavioural diagrams comprises of use case diagrams, sequence diagrams, state diagram, and activity diagram.

The different activities carried out in UML are [18]:

1. The first activity of modelling start with the activity diagram that shows the flows of control. Transition shows the flow of information. Each state of activity diagram can in turn form into the use case diagram. Activity diagram can be used in planning stage where the steps are not detailed as well as in design stage where it becomes more detailed. One of another way to represent activity diagram is through swim-lanes. It shows flow of control from ownership point of view.
2. Use case diagram shows flow of events by actors interacting with the system. Actors are those entities that interact with the system. Overall use case provides with the correct view about the different functionalities of the system.
3. Sequence Diagram shows the object interactions from time period perspective. This diagram is very useful during analysis phase of software development and helps in finding time sequence about the behaviour in use case diagram.
4. Collaboration diagram shows relationships and interaction between objects. This diagram is used in later stage after program has its implementation done.
5. Class Diagram: This diagram shows relationships between the classes, classes' attributes and operations. Class diagram shows the static nature of system. The inputs in the class diagram may be from different diagrams such as sequence diagram that gives the view way operation should perform.
6. State transition diagram: It shows the different states class will be in.

Appendix B: Cello

Cello is base product that has its usage in RBC, RNC and also in mobile handset. They provide ATM cell switching network. It includes of ATM transport system, distinguished real time telecom control system, and elements that can be used for network management system.

Cello provides tools and instructions to develop custom software and hardware for ATM cell switching node, RBS and RNC. Cello consists of modules that include software, function to set up connections and modify operating parameters, and hardware, such as processors board, switchboards and backplane connectors.

Cello several services to the application program running on it [15]:

- Software execution platform: It uses OSE Delta on all processors present in the node. It uses inband AAL5 paths for inter processor communication and has processor cluster to allow robust and scalable software.
- Operation and maintenance of node: It provides management interface based on CORBA, HTTP, Telnet and FTP. It makes use of Java environment for the implementation purposes. It is also responsible for providing real time database and for loading of MP (Main Processor)/BP (Board Processor) for fundamental configuration and start/restart functions.
- Network and connection handling: It provides functions such as network routing, traffic management, virtual connection for cross connections, and data transfer between ATM end points. It provides signalling service and is responsible for creating time slots in frame to transfer ATM cells. It provides network synchronization to enable cello node in network to have a common timing rate.
- Physical infrastructure: It provides upto 20 subracks with 26 processor or device boards in each subrack. It provides processor cluster. 17Gbps cell switch per subrack and timing for network synchronization is handled by cello.

Cello platform provides a common base for all the application developed to it. Cello provides database as described above that is used to retrieve value to be passed to client and server running outside RoseRT environment. It gets value stored in it about the time overload protection mechanism rejected connections and these details are printed to the operator to give idea about system performance.

Appendix C: Overload Protection Code

This part contains processes used in client and server communication.

Note: These codes are not my piece of code but Ericsson implementation of overload protection. I have made some changes so that it works in simcello environment.

```
#include <ose.h>
#include <osetypes.h>
#include <stdio.h>
#include <time.h>

#include "omcsf_te_ose_dex.h"
#include "omcsf_te_error_dex.h"
#include "omcsf_te_trace_dex.h"

#include "bc_bcnm1.sig"
#include "bcnm_overload_data1.h"

#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

time_t curtime;
SIGSELECT any_sig[] = {0};

OS_PROCESS( loadControlServer_procLoop )
{
  union SIGNAL *rec_sig, *send_sig;

  for (;;)
  {
    rec_sig = receive(any_sig);

    if (rec_sig->sigNo == BC_OP_INIT_SERVER_IND)
    {
      PROCESS bcNmPid = sender(&rec_sig);

      int sequenceNo = 0;
      /* sample interval for sending of signal BC_OP_PING_REQ */
      int sampleIntervalT = rec_sig->bcOpInitServerInd.opSampleIntervalT;
      /* process Id of the response server */
      PROCESS pid = rec_sig->bcOpInitServerInd.pidRespoderServer;
      /* overload indication, 0 = not overloaded, 1 = overloaded */
      struct     BcNmOverloadControlDataS     *overloadPtr     =     (struct
BcNmOverloadControlDataS *)rec_sig->bcOpInitServerInd.overloadPtr;
      /* supervision time for signal BC_OP_PING_RSP */
      int supervisionT = rec_sig->bcOpInitServerInd.opSupervisionT;

      send_sig = alloc(sizeof(struct BcOpPingReqS), BC_OP_PING_REQ);
      (void) time(&curtime);
      printf("\n %s",asctime(gmtime(&curtime)));

   printf("        BC_OP_INIT_SERVER_IND(samplIntervalT=%d,        pid=%d,
supervisionT=%d,  overloadIndicator=%d)  received",  sampleIntervalT,  pid,
supervisionT, overloadPtr->overloadIndicator);
      sequenceNo = sequenceNo + 1;
      send_sig->bcOpPingReq.sequenceNo = sequenceNo;
      send(&send_sig, pid);
      for(;;)
      {
        rec_sig = receive_w_tmo((OSTIME) supervisionT, any_sig);
        if (rec_sig)
        {
          switch(rec_sig->sigNo)
          {
            case BC_OP_PING_RSP:
              (void) time(&curtime);
            printf("\n %s",asctime(gmtime(&curtime)));
                  printf("      BC_OP_PING_RSP(sequenceNo=%d),      expected
sequenceNo=%d received", rec_sig->bcOpPingRsp.sequenceNo, sequenceNo);

              if (sequenceNo == rec_sig->bcOpPingRsp.sequenceNo)
              {
                delay(sampleIntervalT);
                send_sig      =      alloc(sizeof(struct      BcOpPingReqS),
BC_OP_PING_REQ);
                sequenceNo = sequenceNo + 1;
                send_sig->bcOpPingReq.sequenceNo = sequenceNo;

                send(&send_sig, pid);
            (void) time(&curtime);
            printf("\n %s",asctime(gmtime(&curtime)));
            printf(" Message sent : %d", sequenceNo);

                if (overloadPtr->overloadIndicator == 1)
                {
              (void) time(&curtime);
              printf("\n %s",asctime(gmtime(&curtime)));
                        printf("   overloadIndicator   changed   to:   not
overload(0), sending BC_OP_OVERLOAD_CEASED_IND");
                        overloadPtr->overloadIndicator = 0; /*   set the
overload indicator to false */
                        send_sig           =           alloc(sizeof(struct
BcOpOverloadCeasedIndS), BC_OP_OVERLOAD_CEASED_IND);
                    send(&send_sig, bcNmPid);
                }
              }
              break;
            default: ; /*  recieved unexpected signal */
          }
          free_buf(&rec_sig);
        }
        else  /* timeout, the base station is overloaded */
        {
          (void) time(&curtime);
          printf("\n %s",asctime(gmtime(&curtime)));
         printf(" Timeout sequenceNo=%d received", sequenceNo);
           if (overloadPtr->overloadIndicator == 0)
           {
           (void) time(&curtime);
           printf("\n %s",asctime(gmtime(&curtime)));
                 printf("   overloadIndicator   changed   to:   overload(1),
sending BC_OP_OVERLOAD_IND");
                 overloadPtr->overloadIndicator = 1; /*   set the overload
indicator to true */
                 send_sig    =    alloc(sizeof(struct    BcOpOverloadIndS),
BC_OP_OVERLOAD_IND);
                 send(&send_sig, bcNmPid);
           }
                send_sig      =      alloc(sizeof(struct      BcOpPingReqS),
BC_OP_PING_REQ);
                sequenceNo = sequenceNo + 1;
                send_sig->bcOpPingReq.sequenceNo = sequenceNo;
                send(&send_sig, pid);
        }
      }
    }
    else
    {
      /*  received unexpected signal */
      free_buf(&rec_sig);
    }
  }
}

OS_PROCESS( loadControlResponseServer_procLoop )
{
  int sequenceNo;
  /* This is the main process loop */
  for(;;)
  {
    union SIGNAL *rec_sig, *send_sig;
    rec_sig = receive(any_sig);
    switch(rec_sig->sigNo)
    {
       case BC_OP_PING_REQ:
         sequenceNo = rec_sig->bcOpPingReq.sequenceNo;

         send_sig = alloc(sizeof(struct BcOpPingRspS), BC_OP_PING_RSP);
         send_sig->bcOpPingReq.sequenceNo = sequenceNo;
       (void) time(&curtime);
      printf("\n %s",asctime(gmtime(&curtime)));
       printf(" Server has sent the reply: %d", sequenceNo);
         send(&send_sig, sender(&rec_sig));
         break;
       default: ; /*  received unexpected signal */
    }
    free_buf(&rec_sig);

  }
}
PROCESS startClient1(int prio, int stacksize)
{
  PROCESS proc;
  proc = create_process ( OS_PRI_PROC,
                    "bcNmLoadControlServer",
                    loadControlServer_procLoop,
                    stacksize,
                    (OSPRIORITY) prio,
                    (OSTIME) 0,
                    (PROCESS) 0,
                    (struct OS_redir_entry *) 0,
                    (OSVECTOR) 0,
                    (OSUSER) 0);
  (void) time(&curtime);
      printf("\n %s",asctime(gmtime(&curtime)));


  printf(" Client Started \n");
```

```
 /* start the new tread (OSE Process) */

  start(proc);
 return proc;


}
PROCESS startServer1(int prio, int stacksize)
{
  PROCESS proc;
  proc = create_process ( OS_PRI_PROC,
                    "bcNmLoadControlResponseServer",
                    loadControlResponseServer_procLoop,
                    stacksize,
                    (OSPRIORITY) prio,
                    (OSTIME) 0,
                    (PROCESS) 0,
                    (struct OS_redir_entry *) 0,
                    (OSVECTOR) 0,
                    (OSUSER) 0);
  (void) time(&curtime);
  printf("\n %s",asctime(gmtime(&curtime)));
  printf("Server Started\n ");
 /* start the new thread (OSE Process) */
 start(proc);

 return proc;
}

#ifdef __cplusplus
}
#endif /* __cplusplus */
```

Appendix D: Simulation Program Code

This part contains codes for capsules and ports used in rational rose environment.

```
// {{{RME classifier 'Logical View::DriverC'

#if defined( PRAGMA ) && ! defined( PRAGMA_IMPLEMENTED )
#pragma implementation "DriverC.h"
#endif

#include <RTSystem/OverloadProtectionSim.h>
#include <DriverC.h>

// {{{RME tool 'OT::Cpp' property 'ImplementationPreface'
// {{{USR
#include <bc_bcnm.h>
#include <bc_bcnm.sig>
#include <tsl_te_trace_actor.h>
// }}}USR
// }}}RME

static const RTRelayDescriptor rtg_relays[] =
{
      {
            "DriverRlsP"
        , &DriverRlsPro::Base::rt_class
        , 1 // cardinality
      }
  , {
            "DriverPbP"
        , &DriverPbPro::Base::rt_class
        , 1 // cardinality
      }
};

static RTActor * new_DriverC_Actor( RTController * _rts, RTActorRef * _ref
)
{
      return new DriverC_Actor( _rts, _ref );
}

const RTActorClass DriverC =
{
      (const RTActorClass *)0
  , "DriverC"
  , (RTVersionId)0
  , 2
  , rtg_relays
  , new_DriverC_Actor
};

static const char * const rtg_state_names[] =
{
      "TOP"
  , "S1"
  , "S2"
};

const RTTypeModifier rtg_tm_DriverC_Actor_overloadptr =
{
      RTNumberConstant
  , 1
  , 1
};

#define SUPER RTActor

DriverC_Actor::DriverC_Actor( RTController * rtg_rts, RTActorRef * rtg_ref
)
      : RTActor( rtg_rts, rtg_ref )
      , opOnOff( 1 )
      , opRespServPrio( 18 )
      , opRoPrio( 14 )
      , opSampleIntervalT( 1000 )
      , opSupervisionT( 100 )
      , overloadRejectCounter( 0 )
{
}

DriverC_Actor::~DriverC_Actor( void )
{
}

// {{{RME operation 'registerOseSignals()'
bool DriverC_Actor::registerOseSignals( void )
{
      // {{{USR
      bool result = false;
      result = REGISTER_OSE_SIGNAL(BC_OP_OVERLOAD_IND,
OpSignalPro::Base::rti_extOverloadInd, General, &overloadInitiator );
      if (result)
      {
      TRACE(3, STR("Register for BC_OP_OVERLOAD_IND (%d) ",
BC_OP_OVERLOAD_IND));
      }
      else
      {
        TRACE_ERROR(STR("OSE signal registration failed for
BC_OP_OVERLOAD_IND (%d) ", BC_OP_OVERLOAD_IND));
      }


      if (result)
      {
        result = REGISTER_OSE_SIGNAL(BC_OP_OVERLOAD_CEASED_IND,
OpSignalPro::Base::rti_extOverloadCeasedInd, General, &overloadInitiator );
        if (result)
        {
           TRACE(3, STR("Register for BC_OP_OVERLOAD_CEASED_IND (%d) ",
BC_OP_OVERLOAD_CEASED_IND));

        }
        else
        {
          TRACE_ERROR(STR("OSE signal registration failed for
BC_OP_OVERLOAD_CEASED_IND (%d) ", BC_OP_OVERLOAD_CEASED_IND));
        }
      }

      return result;
      // }}}USR
}
// }}}RME

int DriverC_Actor::_followInV( RTBindingEnd & rtg_end, int rtg_portId, int
rtg_repIndex )
{
      switch( rtg_portId )
      {
      case 0:
            // DriverRlsP
            if( rtg_repIndex < 1 )
            {
                  rtg_end.port = &DriverRlsP;
                  rtg_end.index = rtg_repIndex;
                  return 1;
            }
            break;
      case 1:
            // DriverPbP
            if( rtg_repIndex < 1 )
            {
                  rtg_end.port = &DriverPbP;
                  rtg_end.index = rtg_repIndex;
                  return 1;
            }
            break;
      default:
            break;
      }
      return RTActor::_followInV( rtg_end, rtg_portId, rtg_repIndex );
}

// {{{RME transition ':TOP:Initial:Initial'
INLINE_METHODS void DriverC_Actor::transition1_Initial( const void *
rtdata, RTProtocol * rtport )
{
      // {{{USR
      (void) Timer.informIn(RTTimespec(0,0));
      // }}}USR
}
// }}}RME

// {{{RME transition ':TOP:S1:J42FB23C50280:t3'
INLINE_METHODS void DriverC_Actor::transition2_t3( const void * rtdata,
Timing::Base * rtport )
{
      // {{{USR
      if (!registerOseSignals())
      {
          TRACE_ERROR("OSE signal registration failed.");
      }
      TRACE(1, STR("OSE signal registration done"));

      int stacksizeServer = 256; // the pong process
      int stacksizeClient = 512; // the ping process

      TRACE(3,STR("Overload Protection server and client process to be
created, Stacksize = %d and %d", stacksizeServer , stacksizeClient ));

      serverProc = startServer(opRespServPrio, stacksizeServer);
      clientProc = startClient(opRoPrio, stacksizeClient);
      overloadRejectCounter = 0;
      overloadptr = new (BcNmOverloadControlDataS);
      overloadptr->overloadIndicator = 0; // false
      overloadptr->overloadRejectCounter = 0;
      //(void)bcNmOverloadControlP.startOverloadControlInd( (void
*)overloadptr ).send();

      union SIGNAL *send_p = alloc(sizeof(struct BcOpInitServerIndS),
BC_OP_INIT_SERVER_IND);
      send_p->bcOpInitServerInd.pidRespoderServer = serverProc;
      send_p->bcOpInitServerInd.overloadPtr = (long unsigned
int)overloadptr;
      send_p->bcOpInitServerInd.opSampleIntervalT = opSampleIntervalT;
      send_p->bcOpInitServerInd.opSupervisionT = opSupervisionT;
      send(&send_p, clientProc);
      (void) Timer.informIn(RTTimespec(10,0));
      INFO("Overload Protection Activated");

      int n;
      for(int i=0;i<200;i++){
            n=rand();
          TRACE(3,STR("OverloadProtectionDriver"));
            if(n%2==0)
                  (void)DriverRlsP.DriRls().send();
            else
                  (void)DriverPbP.DriPb().send();
      }
      exit(0);
      // }}}USR
}
// }}}RME

// {{{RME transition ':TOP:S2:J430474AD00BB:pb'
INLINE_METHODS void DriverC_Actor::transition3_pb( const void * rtdata,
DriverPbPro::Base * rtport )
{
      // {{{USR
      TRACE(3,STR("PB MSG"));
      // }}}USR
}
// }}}RME

// {{{RME transition ':TOP:S2:J430474D302AE:rls'
INLINE_METHODS void DriverC_Actor::transition4_rls( const void * rtdata,
DriverRlsPro::Base * rtport )
{
      // {{{USR
      TRACE(3,"RLS MSG");
      // }}}USR
}
// }}}RME

INLINE_CHAINS void DriverC_Actor::chain1_Initial( void )
{
      // transition ':TOP:Initial:Initial'
      rtgChainBegin( 1, "Initial" );
      rtgTransitionBegin();
      transition1_Initial( msg->data, msg->sap() );
      rtgTransitionEnd();
      enterState( 2 );
}

INLINE_CHAINS void DriverC_Actor::chain2_t3( void )
{
      // transition ':TOP:S1:J42FB23C50280:t3'
      rtgChainBegin( 2, "t3" );
      exitState( rtg_parent_state );
      rtgTransitionBegin();
      transition2_t3( msg->data, (Timing::Base *)msg->sap() );
      rtgTransitionEnd();
      enterState( 3 );
}

INLINE_CHAINS void DriverC_Actor::chain4_rls( void )
{
      // transition ':TOP:S2:J430474D302AE:rls'
      rtgChainBegin( 3, "rls" );
      exitState( rtg_parent_state );
      rtgTransitionBegin();
      transition4_rls( msg->data, (DriverRlsPro::Base *)msg->sap() );
      rtgTransitionEnd();
      enterState( 3 );
}

INLINE_CHAINS void DriverC_Actor::chain3_pb( void )
{
      // transition ':TOP:S2:J430474AD00BB:pb'
      rtgChainBegin( 3, "pb" );
      exitState( rtg_parent_state );
      rtgTransitionBegin();
      transition3_pb( msg->data, (DriverPbPro::Base *)msg->sap() );
      rtgTransitionEnd();
      enterState( 3 );
}

void DriverC_Actor::rtsBehavior( int signalIndex, int portIndex )
{
      for( int stateIndex = getCurrentState(); ; stateIndex =
rtg_parent_state[ stateIndex - 1 ] )
            switch( stateIndex )
            {
            case 1:
                  // {{{RME state ':TOP'
                  switch( portIndex )
                  {
                  case 0:
                        switch( signalIndex )
                        {
                        case 1:
                              chain1_Initial();
                              return;
                        default:
                              break;
                        }
                        break;
                  default:
                        break;
                  }
                  unexpectedMessage();
                  return;
                  // }}}RME
            case 2:
                  // {{{RME state ':TOP:S1'
                  switch( portIndex )
                  {
                  case 0:
                        switch( signalIndex )
                        {
                        case 1:
                              return;
                        default:
                              break;
                        }
                        break;
                  case 4:
                        // {{{RME port 'Timer'
                        switch( signalIndex )
                        {
                        case Timing::Base::rti_timeout:
                              chain2_t3();
                              return;
                        default:
                              break;
                        }
                        break;
                        // }}}RME
                  default:
                        break;
                  }
```

```
                        break;
                        // }}}RME
                case 3:
                        // {{{RME state ':TOP:S2'
                        switch( portIndex )
                        {
                        case 0:
                                switch( signalIndex )
                                {
                                case 1:
                                        return;
                                default:
                                        break;
                                }
                                break;
                        case 1:
                                // {{{RME port 'DriverRlsP'
                                switch( signalIndex )
                                {
                                case DriverRlsPro::Base::rti_RlsDri:
                                        chain4_rls();
                                        return;
                                default:
                                        break;
                                }
                                break;
                                // }}}RME
                        case 2:
                                // {{{RME port 'DriverPbP'
                                switch( signalIndex )
                                {
                                case DriverPbPro::Base::rti_PbDri:
                                        chain3_pb();
                                        return;
                                default:
                                        break;
                                }
                                break;
                                // }}}RME
                        default:
                                break;
                        }
                        break;
                        // }}}RME
                default:
                        unexpectedState();
                        return;
                }
}

const RTActor_class * DriverC_Actor::getActorData( void ) const
{
        return &DriverC_Actor::rtg_class;
}

const RTActor_class DriverC_Actor::rtg_class =
{
        (const RTActor_class *)0
  , rtg_state_names
  , 3
  , DriverC_Actor::rtg_parent_state
  , &DriverC
  , 0
  , (const RTComponentDescriptor *)0
  , 4
  , DriverC_Actor::rtg_ports
  , 0
  , (const RTLocalBindingDescriptor *)0
  , 9
  , DriverC_Actor::rtg_DriverC_fields
};

const RTStateId DriverC_Actor::rtg_parent_state[] =
{
      0
  , 1
  , 1
};

const RTPortDescriptor DriverC_Actor::rtg_ports[] =
{
      {
            "DriverRlsP"
        , (const char *)0
        , &DriverRlsPro::Base::rt_class
        , RTOffsetOf( DriverC_Actor, DriverC_Actor::DriverRlsP )
        , 1 // cardinality
        , 1
        , RTPortDescriptor::KindWired +
RTPortDescriptor::NotificationDisabled +
RTPortDescriptor::RegisterNotPermitted + RTPortDescriptor::VisibilityPublic
      }
  , {
            "DriverPbP"
        , (const char *)0
        , &DriverPbPro::Base::rt_class
        , RTOffsetOf( DriverC_Actor, DriverC_Actor::DriverPbP )
        , 1 // cardinality
        , 2
        , RTPortDescriptor::KindWired +
RTPortDescriptor::NotificationDisabled +
RTPortDescriptor::RegisterNotPermitted + RTPortDescriptor::VisibilityPublic
      }
  , {
            "overloadInitiator"
        , (const char *)0
        , &OpSignalPro::Base::rt_class
        , RTOffsetOf( DriverC_Actor, DriverC_Actor::overloadInitiator )
        , 1 // cardinality
        , 3
        , RTPortDescriptor::KindWired +
RTPortDescriptor::NotificationDisabled +
RTPortDescriptor::RegisterNotPermitted +
RTPortDescriptor::VisibilityProtected
      }
  , {
            "Timer"
        , (const char *)0
        , &Timing::Base::rt_class
        , RTOffsetOf( DriverC_Actor, DriverC_Actor::Timer )
        , 1 // cardinality
        , 4
        , RTPortDescriptor::KindSpecial +
RTPortDescriptor::NotificationDisabled +
RTPortDescriptor::RegisterNotPermitted +
RTPortDescriptor::VisibilityProtected
      }
};

const RTFieldDescriptor DriverC_Actor::rtg_DriverC_fields[] =
{
      // {{{RME classAttribute 'serverProc'
      {
            "serverProc"
        , RTOffsetOf( DriverC_Actor, serverProc )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_RTulong
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'clientProc'
  , {
            "clientProc"
        , RTOffsetOf( DriverC_Actor, clientProc )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_RTulong
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'opOnOff'
  , {
            "opOnOff"
        , RTOffsetOf( DriverC_Actor, opOnOff )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_int
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'opRespServPrio'
  , {
            "opRespServPrio"
        , RTOffsetOf( DriverC_Actor, opRespServPrio )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_int
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'opRoPrio'
  , {
            "opRoPrio"
        , RTOffsetOf( DriverC_Actor, opRoPrio )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_int
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'opSampleIntervalT'
  , {
            "opSampleIntervalT"
        , RTOffsetOf( DriverC_Actor, opSampleIntervalT )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_int
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'opSupervisionT'
  , {
            "opSupervisionT"
        , RTOffsetOf( DriverC_Actor, opSupervisionT )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_int
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'overloadptr'
  , {
            "overloadptr"
        , RTOffsetOf( DriverC_Actor, overloadptr )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , (const RTObject_class *)0
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
        , &rtg_tm_DriverC_Actor_overloadptr
            // }}}RME
      }
      // }}}RME
      // {{{RME classAttribute 'overloadRejectCounter'
  , {
            "overloadRejectCounter"
        , RTOffsetOf( DriverC_Actor, overloadRejectCounter )
            // {{{RME tool 'OT::CppTargetRTS' property 'TypeDescriptor'
        , &RTType_unsigned
            // }}}RME
            // {{{RME tool 'OT::CppTargetRTS' property
'GenerateTypeModifier'
```

```
        , (const RTTypeModifier *)0
            // }}}RME
      }
      // }}}RME
};
#undef SUPER

// {{{RME tool 'OT::Cpp' property 'ImplementationEnding'
// {{{USR

// }}}USR
// }}}RME

// }}}RME
```