“Simulation of Licklider Transmission Protocol in a OMNeT++® Environment”

A professional project presented
to the faculty of
the J. Warren McClure School of
Information and Telecommunication Systems

In partial fulfillment of the requirements for the degree
Master of Communication Technology and Policy

Luis Fernando Carrasco
December 2007
# Table of Contents

Introduction........................................................................................................................................ 7
The Project ......................................................................................................................................... 9
  Project Objectives .......................................................................................................................... 10
OMNeT++® Simulation Software .................................................................................................... 11
  Structure and Elements in OMNeT++® ....................................................................................... 13
  OMNeT++® Software Installation ................................................................................................. 13
Licklider Transmission Protocol Description ...................................................................................... 16
LTP Simulation.................................................................................................................................... 17
LTP Model Structure .......................................................................................................................... 18
  Components of the LTP Protocol Model .................................................................................... 19
    Cassini......................................................................................................................................... 19
    Earth Station.............................................................................................................................. 19
    Space_1 and Space_2 .............................................................................................................. 19
    Transceivers_(CS, ES)............................................................................................................ 20
    Rx_(CS, ES).......................................................................................................................... 20
    Tx_(CS, ES).......................................................................................................................... 20
LTP Messages.................................................................................................................................... 21
  LTP Segment .............................................................................................................................. 21
  LTP Segment Model Core Structure Model ............................................................................... 22
  LTP Segment Messages ............................................................................................................. 24
    Red Data Segment (RDS)......................................................................................................... 24
    Green Data Segment (GDS)...................................................................................................... 24
    Checkpoint (CP)...................................................................................................................... 24
    Report Segment (RS)............................................................................................................... 25
    Report Acknowledgment Segment (RA).................................................................................. 25
    Cancel-Segment (CS/CR)...................................................................................................... 26
    Cancel-Acknowledgement Segment (CAS/CAR).................................................................... 26
LTP Timers ........................................................................................................................................ 27
Introduction

Deep space communications differ from those designed to perform over terrestrial links due the severe conditions present in space. Different factors like signal propagation delays, limited power consumption of spacecraft, data corruption due to atmospheric parameters, solar flares and planet movements are just few of the problems to limit the exchange of information between two or more sources of data.

Long Signal Propagation Delays between spacecraft and earth stations are one of the principal factors of why optimized protocols for deep space communications must be used. As a reference, Point-to-Point Round Trip Time (RTT) from to earth to LEO satellites (200-2,000 km above the earth’s surface) ranges from several milliseconds to around 80 ms, while GEO satellites (36,000 km) RTTs are in the range of 239.6 to 279.0 ms[9]. For these RTTs, protocols commonly utilized in terrestrial applications can still be applied. When applications like interplanetary communications exceed these distance magnitudes, for example Earth to Mars links with more than 60 million kilometers and RTTs of 8 to 40 minutes[2][11], other techniques for the exchange of data must be used to manage these long delays.

Frequent interruptions in connectivity are also inherent to the deep space communications environment. Lengthy periods of blindness with the impossibility of transmitting any signal due to blockage of the line of sight by a planet’s rotation and orbit, in addition to the loss of visibility when the sun is between these two points, increase the time a message must wait before being sent the signal[2]. These waiting times must be managed in a way that receiver must know how much time it must freeze or reset the timers, interpreting this as a non-error in the communication link.
Engineering elements, such as limited spacecraft electric power reserved for transmissions, are very important, causing a limitation on communications bandwidth. An example of this is the Cassini-Huygens mission maximum downlink data transfer rate of 165.9 Kb/s reached by using asymmetric communication, where the uplink bandwidth is just 1 Kb/s.

With the absence of these factors, protocols designed to work on terrestrial infrastructure are designed from the perspective that in case of disruptions of reception channels or loss of data, continuous retransmissions are the answers to solve the problem, which in space projects this is not the best solution. As an example, in the TCP/IP protocol used for Internet connections, TCP stacks normally uses two-minute timeouts between nodes for transmission or reception before breaking down the connection, which for deep space communications would not be a practical behavior.

To customize a solution for the characteristics of the channel, the Delay Tolerant Network Research Group is proposing a communication protocol that can solve the particular problems in this environment. Licklider Transmission Protocol (LTP) is a
datalink layer protocol designed to function in deep-space data links, providing a reliable form of point-to-point communication between a tentative point on earth and a spacecraft in outer space.

The LTP protocol offers transmission of data in a unidirectional way, generating an acknowledgement for every packet received in the reverse channel. Because it was designed for long-haul communications, there is no handshake or negotiations (as we know it in other protocols) before starting a transmission, avoiding with this high delays between peers. Automatic Request for Retransmission (ARQ) mechanisms are managed different from those in TCP, which uses sliding windows and slow start processes to avoid any delays in the channel, doing this by previous configuration of parameters in transmitter / receiver.

The point-to-point protocol characteristic helps both peers to transmit data in an efficient way, acting as Data Link Layer protocol over an asynchronous connection. This simple set of primary protocols provides basic concepts necessary for the exchange of data such as ARQ, data integrity, origin-authentication and reliability, among others, without the need of several round-trips.

Another important feature of the LTP protocol is the scheduling technique that uses a low-layer cue to help manage the time the receivers must wait to receive an ACK.

**The Project**

The goal of this project is to model the Licklider Transmission Protocol from the concept, defined by the Delay Tolerant Networking Research Group (DTNRG) in the specification\[4\] and motivation\[2\] documents, to the implementation in a computer simulation. To do this, we will use the open-source simulation software tool called OMNeT++\® to apply the protocol and evaluate the response in a controlled environment. The software allows me to control three basic and important parameters completely applicable in the LTP protocol: *Propagation Delay, Bit Error Rate* and *Data Rate*. 
**OMNeT++** manages the simulation of networks in a Graphic User Interface (GUI) consisting in a hierarchical group of **modules** that use **messages** to represent frames or packets. These **messages** will contain the data sent to it by the transmitter with the appropriate structure of the protocol which in our case is LTP\[8\].

![OMNeT LTP Model Representation](image)

**Fig. 2.** - OMNeT LTP Model Representation [Fernando C.]

**Project Objectives**

By simulating the LPT protocol with the OMNeT software, we must obtain different data related to the behavior of the protocol in the deep space environment. For this project, we will do the following different tasks:

- **Modeling of LTP Protocol.** Using C++ language, we will model the structure of this protocol based on the LTP specification document including the overall segment structure (flags, codes, masks, etc), following the transmitter and receiver state transition diagram defined in the document.

- **Analysis of Data Propagation Delay.** Using timers in both modules, we will be able to obtain the delay of data using the real structure of the protocol.
• **Analysis of Protocol behavior due to Transmission Interruptions.** We will analyze how the timers of the nodes must work to keep packets on hold during an interruption in a data transmission.

• **Analysis of Bit Error Rate of the messages.** By using a model to inject errors in the link, we will simulate the response of the protocol based on number of successful packet received during the transmission.

**OMNeT++® Simulation Software**

This program is an open-source simulation software designed to model communication networks and the protocols that rules these networks. This program works based in a structure formed by hierarchical nested modules that are assembled in a methodical way, from a high level, which can be described as the general top structure of the network. Inside this system module exist subsystem modules than can contain more subsystems modules, depending the complexity of the network.

Modules enclosing other modules are known as compound modules and the last module in the structure, is known as simple module, which is the module containing the algorithm of the model, and therefore the C++ code of the protocol.

![Hierarchical Structure in OMNeT++®](Fig. 3 - Hierarchical Structure in OMNeT++® [8])

**Messages.** The way the modules establish communication among them, is by the exchange of information in the form of messages. These messages represent the frames or
packets containing important data to send. For the purpose of our model, messages in our model will be represented by segments containing data, reports or cancellations.

**Gates.** Used in our modules to work as a representation of input and output interfaces, using them as arrival and departure ports for our data. Each module in our model contains an input and an output gate. Outputs can only be connected to inputs and vice versa.

**Links.** Links or connections in our model will be used to represent the channel through which the messages are sent between modules. These connections can be customized by using optional parameters to modify the behavior of the model:

- **Propagation Delay**, representing the amount of time the arrival of the messages is delayed by when is sent it by a transmitter module and travels through the channel. This parameter is commonly represented in seconds as a time base.

- **Bit Error Rate**, which specifies the likelihood that a bit is wrongly transmitted over a noisy channel and defined by a ratio. This ratio is measured dividing the number of erroneous bits received by the total number of bits that was transmitted by the sender as can be seen in the next equation:

- **Data Rate** is the average number of data, in our case a segment, per unit of time (in seconds) transferred from one point to other, measure as bits per second (bps).
Structure and Elements in OMNeT++®

To build a simulation using the OMNeT++® software tool, programming of modules by using C++ language is needed. The simulator is formed in general by the following parts:

- The NED language topology description (.ned files) is used to define the structure of each module using parameters, gates, connections, etc. These ‘NED’ files can be generated by using an existing text editor or the GNED included in the simulator tool.

- The definition of Messages (.msg files) where we can define a variety of message types, adding other data fields to them. The simulation tool will convert these message definitions into C++ classes.

- Module sources are C++ files with standard .h and .cc extensions. The algorithm of the model resides in this section. Programming of the code is done in C++, using Microsoft Visual C++ (Windows version).

OMNeT++® Software Installation

System Requirements [10]
Windows XP, Windows 2000, NT 4.0 or other Windows version which build on NT technology are required. (OMNeT++ is not supported and will not work on Win95, Win98, or ME.). Linux version also available. [10].

Software packages required.

4. Platform

SDK:


**Installation Guidelines:**

- Install the OMNeT installation package in your Windows XP system. Be sure the installation was done without problems. All the default settings and paths should be ok. [10]
- Download and install the SDK Platform (~400M) [10]
- Download and install MSVC Toolkit (~32M) [10]
- Download and install MSVC++ Express Edition (useful - ~400M) Visual C++ 2005 Express Edition:

Some small problems can be found in your first run with the OMNeT file from the CMD window, like missing files required in the command prompt. Each of these files must be manually copied from the SDK package /lib, /include, and /bin folders to the omnet/lib /bin and /include folders.

Be sure the folder containing your model application program and files is running directly at the C:/ root (i.e.: C:\LTP_Simulation\LTP_Simulation.exe).

Initially, your project folder must contain at least these 8 files:

- LTP.ned (File containing the structure of the network.
- Receiver_CS.cpp (Cassini’s Receiver Simple Module)
- Transmitter_CS.cpp (Cassini’s Transmitter Simple Module)
- Receiver_ES.cpp (Earth Station’s Receiver Simple Module)
- Transmitter_ES.cpp (Earth Station’s Transmitter Simple Module)
- Dummy_Cloud.cpp (Simple Module for Space_1 and Space_2 Clouds)
- Omnetpp.ini (file containing the parameter and networks we want to simulate)
- Vcvars32.bat. (file to set the appropriate environment variables to enable 32-bit command-line builds)

Having these files in our project folder, we can start running the simulation typing these commands:

- c:\LTP_Simulation\vcvars32.bat
- c:\LTP_Simulation\opp_nmakemake (to create Makefile.vc, first run)
- c:\LTP_Simulation\nmake –f Makefile.vc (to compile and link our most recent .cpp file)
- c:\LTP_Simulation\LTP_Simulation.exe
Licklider Transmission Protocol Description

As it was mentioned in the introduction, the Licklider Transmission Protocol (LTP) was created to answer the need of establishing a reliable interplanetary space network, dealing with several factors not found or relatively easily solved within the terrestrial atmosphere. LTP protocol was intended to unravel inherent problems in the deep space Radio Frequency (RF) links between spacecraft and terrestrial transceiver equipments. In the next paragraphs we explain some factors that could affect these kinds of outer-space links.

**Long RF signal propagation delays**\(^7\) - referring to the time lag for a RF signal to be sent through the space by a peer, called “Transmitter” and the arrival of this signal to a counterpart or destination known as “Receiver.” The delay is a natural effect directly proportional to the distance between peers and the medium of transmission.

Due to these distances, propagation delay for deep space communications can be in the order of hours. The example I’m using for this project is the current distance between Earth and the Cassini probe, with a one-way propagation delay of 1 hour and 14 minutes, with a Round trip time (RTT) of 2 hr. 28 min..

**Connectivity Interruptions**\(^7\) - These effects are the result of planetary movement, occultation, solar flares and other problems inherent to the medium. Because the need of line-of-sight to transmit and receive the data, we are dependant on the rotation and translation movements of the earth, as well other planets. These interruptions can be made by freezing timer on each engine, following a previous connectivity chart schedule to stop/start the transmission.

**High Rate Transmission Errors**\(^7\) - As I previously mentioned, errors continuously threaten data integrity on their way to the receiver. These effects are unpredictable and the way to solve it is by implementing retransmission requests and acknowledgements from sender, based on check points (CP) per each block of data segments (DS), coding
techniques like Self-Delimiting Numeric Value (SDVN) and complex RF modulation to improve the quality and easiness of signal retrieving at the receiver. Current Deep Space Network projects are currently working with a Bit Error Rate (BER) between $1 \times 10^{-1}$ and $1 \times 10^{-3}$.

**Limited Bandwidth and Asymmetrical Data Rates** Attenuation signal due to the huge distance between peers, economy behind the Earth Stations infrastructure and low-power capabilities of spacecraft results in a limited bandwidth. These restrictions in capacity are reflected also in the non-symmetry or Asymmetric transmissions, which is the characteristic to transmit in different rates based on the transmission direction. Normally, due the importance of data, high data rates (i.e. 1-10 Mbps) are assigned to the link from spacecraft to Earth, in ratios of 1:100 or even 1:1000, depending the importance of the information. Control or Command channel has a lower bandwidth because Monitoring and Control (M&C) use just a few commands and little data to keep an accurate feedback from spacecraft.

**LTP Simulation**
The goal for this project is to successfully represent a model in a controlled environment using the tools that OMNeT++ offers. This is done to obtain a real representation of the model and understand the behavior of the Licklider Transmission Protocol in a Deep Space Network.

Modeling of this protocol has been done according to the LTP specification paper, `<draft-irtf-dtnrg-ltp-05.txt>` posted on September 2006, by the Delay Tolerant Networking Research Group, specifically by the following authors:

**Authors:**
Manikantan Ramadas
Scott C. Burleigh
Stephen Farrell

**Research Institution:**
Ohio University
NASA / Jet Propulsion Laboratory
Trinity College Dublin
LTP Model Structure

To model the Licklider Transmission Protocol, we are including a total of 10 modules to effectively represent the transmission, the middle steps of data exchange and the reception of data. We are using modules that normally would be part of a real radiofrequency (RF) telecommunication system for this purpose. With this structure, we will be able to follow these steps in graphical manner, and also, by following the information displayed in the log window offered by OMNeT++. 

Each module has an ID assigned to it, which can be used to follow the process during the running of the program. The next figure contains all modules and their ID’s assigned to it, and the connectors used to link these modules each other on both sides of the engines.

![Fig. 4 LTP Modules Configuration Structure](attachment:image.png)
Components of the LTP Protocol Model

The main components of this LTP model are represented by compound and simple modules:

**Cassini**
The LTP engine representing the spacecraft traveling outer space. Cassini will be transmitting a simulated RF signal to Earth Station and will be receiving a simulated RF signal from Earth Station.

**Earth Station**
The LTP engine representing the antennas and telecommunications equipment located at planet Earth. The Earth station will be transmitting a simulated RF signal to Cassini and is receiving a simulated RF signal from Cassini.

**Space_1 and Space_2**
Middle step modules between Cassini and Earth Station in both directions to represent the delay added by space. Space_1 will be forwarding the simulated RF signal from Earth Station to Cassini. Space_2 will be forwarding the simulated RF signal from Cassini to Earth Station.

![Fig. 5.- LTP Cassini and Earth Station Engines](Fernando C / OMNeT++)
Transceivers (CS, ES)
Compound module containing the receiver and transmitter simple modules on Cassini (CS) and Earth Station (ES). This module represents a middle step between Cassini / Earth Station Compound Modules and transmission / reception hardware on each engine. It does not contain any software processing.

Fig. 6.- LTP Cassini and Earth Station Transceivers(_CS & _ES) [Fernando C / OMNeT++]

Rx (CS, ES)
Cassini and Earth station Receivers are simple modules containing the processing software of the LTP network. Most of the LTP modeling computing process (~98%) is done in these modules. Steps like message detection, framing, deframing, error injection and other tasks made on C++ are represented here.

Tx (CS, ES)
Cassini and Earth Station Transmitters are simple modules. Their main task is to forward the LTP message to the destination engine. Few processes and graphic messages resides on these modules.
LTP Messages

**LTP Segment**
Is the basic unit of LTP data transmission, with a bit structure which includes version number, segment type flags, session ID, header and trailer extensions, and content or data to transmit.

![LTP Segment Structure](image)

Because a Segment is the elemental structure of data transmission, depending on the information in the flag, this segment can be used to represent different types of messages.
**LTP Segment Model Core Structure Model**

For this model, I chose to use a core structure for the exchange of messages to comply with the LTP protocol. This core structure used by all the messages contains 5 different fields that are used by each message. Depending on the message type, different fields will be used to encode, send, receive and decode the information contained in this structure.

This elemental structure is formed by 32 bits, which contains the following fields:

**Value1 Field (8 bits)** - Contains the Report Segment number for the exchange of data from the Earth Station to Cassini. For this model, a maximum \(^1\) of 256 Report Segment numbers can be used.

**Value2 Field (8 bits)** - Contains the Checkpoint number for the exchange of data from Cassini to the Earth Station and vice versa. For this model, a maximum \(^2\) of 256 Checkpoint numbers can be used. This field is also used from Earth Station to Cassini sending the Error Byte or Data Segment with Error to retransmit. A maximum \(^3\) of 8 Data segments with errors can be requested in case of a complete loss of a 1 block.

**Value3 Field (8 bits)** - Contains the Data Segment number for the exchange of data from Cassini to the Earth Station. For this model, a maximum \(^2\) of 256 Checkpoint numbers can be used. This field is also used from Earth Station to Cassini sending the Error Byte or Data Segment with Error to retransmit. A maximum \(^3\) of 8 Data segments with errors can be requested in case of a complete loss of a 1 block.

**Value4 Field (3 bits)** - Field reserved for future expansion of DS number and/or Error Byte.

---

[Fernando C.]
Simulation of LPT in OMNet++

**Flags field (4 bits)** - This field contains the Segment Type codes specified by the LTP Specification Document[4] and that are shown below. The code on each message is the first field read by the program during the arrival of messages on each simple module.

**Extra Bit (1 bit)** - This bit was left intentionally empty to avoid counting errors due to overflow of data during the exchange of segment. (4)

**Note:** The sizes of the arrays were chosen to represent this model in an appropriate manner. There is not restriction set by the program to increase these parameters.

(1). Size of the Report Segment number array in the TxRx.cpp program. Modifiable by user needs.
(2). Size of the Checkpoint number array in the TxRx.cpp program. Modifiable by user needs.
(3). For this program version, we can make an upgrade to manage up to 11 bits by using the value4 field.
(4).- For the 32-bit compiler used in our program, the range of values is -2147483648 to 2147483647.

<table>
<thead>
<tr>
<th>Control</th>
<th>Exec</th>
<th>Flag 1</th>
<th>Flag 0</th>
<th>Code</th>
<th>Nature of Segment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Red Data, NOT (Checkpoint, EORP or EOB)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>Red Data, Checkpoint, NOT (EORP or EOB)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>Red Data, Checkpoint, EORP, NOT (EOB)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>Red Data, Checkpoint, EORP, EOB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>Green Data, NOT EOB</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>Undefined</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Green Data, EOB</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>Report Segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>Report-Acknowledgement Segment</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>Undefined</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>Undefined</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>Cancel Segment from Block Sender</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>Cancel-Ack. Segment to Block Sender</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>Cancel Segment from Block Receiver</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>Cancel-Ack. Segment to Block Receiver</td>
</tr>
</tbody>
</table>

Table 1.- Segment Type Codes specified by LTP Protocol [4]
LTP Segment Messages

Red Data Segment (RDS). – This type of data is tagged as a priority delivery and reliable data. These segments are subject to acknowledgement and retransmission from/to the peer receiver. The RDS message traveling through the LTP network model will be graphically shown as: “● (cMessage) Red Data Segment-1”, where the number next to the description is the number of the RDS sequence. These messages also contain red dots above the legend to show their priority.

Green Data Segment (GDS).- Data that can be transmitted in an unreliable manner and is not subject to any acknowledgement or retransmissions. When a GDS is present in a session, it is always transmitted following the last octet of RDS. The GDS message traveling through the LTP network model will be graphically shown as: “● (cMessage) Red Data Segment-1”, where the number next to the description is the number of the GDS sequence. These messages contain green dots above the legend to show their priority. If the message sent by the engine is the last GDS message of the block’s green part or the eighth or multiple (specifically for this LTP Model version) in the transmission, this will be shown as “● (cMessage) Green Data Segment-23, & EOB”, where EOB stands for “End of Block”.

Checkpoint (CP).- It is part of the RDS reliable transmission process and is used to represent an End of Block (EOB) and an End of Red Part (EORP). A CP must be included on every retransmission. The RDS message with CP traveling through the LTP network model will be graphically shown as: “● (cMessage) Red Data Segment-7, CP-79”, where the number next to the description is the number of the RDS sequence. If this RDS is the eighth or multiple of it (specifically for this LTP Model version) in the transmission, it will add a Checkpoint number. These messages also contain red dots above the legend to show their priority. If the message sent by the engine is the last RDS message of the block’s red part in the transmission, this will be shown as “● (cMessage) Red Data Segment-7, CP-79 & EORP”, where EORP stands for “End of Red-Part”.
Fig. 10.- RDS and RDS with Checkpoint

Report Segment (RS).- Segment with data reception claims. This report segment includes the number of the report (chosen randomly), Check Point number to which it is responding and the data segments being claimed. The RS message traveling through the LTP network model will be graphically shown as: “

Fig. 11.- Report Segment Structure

Report Acknowledgment Segment (RA). – transmits only the serial number of the report segment (RS) that is responding to. The RA message traveling through the LTP network model will be graphically shown as: “

Page 25 of 144
of the Report Segment (RS) sequence which this RA is responding to. These messages are represented by random color dots above the legend to be distinguished.

![Report Acknowledgement Segment Structure](image1.png)

**Fig. 12.- Report Acknowledgement Segment Structure** [Fernando C.]

**Cancel-Segment (CS/CR).** It cancels a segment from the block sender (CS) or receiver (CR). This type of segment contains a single byte reason-code, defining the cause of the cancellation.

![Cancel CS/CR Segment Structure](image2.png)

**Fig. 13.- Cancel CS/CR Segment Structure** [Fernando C.]

<table>
<thead>
<tr>
<th>Reason-Code</th>
<th>Mnemonic</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>USR_CNCLD</td>
<td>Client Service Canceled Session</td>
</tr>
<tr>
<td>01</td>
<td>UNREACH</td>
<td>Unreachable Client Service</td>
</tr>
<tr>
<td>02</td>
<td>RLEXC</td>
<td>Retransmission Limit Exceeded</td>
</tr>
<tr>
<td>03</td>
<td>MISCOLORED</td>
<td>Red-Part and Green-Part Segments overlapped</td>
</tr>
<tr>
<td>04</td>
<td>SYS_CNCLD</td>
<td>Unexpected Session Termination caused by a System Error</td>
</tr>
<tr>
<td>05</td>
<td>RXMTCYCEXC</td>
<td>Exceeded the Retransmission-Cycles limit</td>
</tr>
<tr>
<td>06-FF</td>
<td>Reserved</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.- Cancel CS/CR Segment Structure** [Fernando C.]

**Cancel-Acknowledgement Segment (CAS/CAR).** It cancels an acknowledgement segment to the block sender (CAS) or receiver (CAR). This type of segment does not carry any type of code.
**Block.**- An array of continuous octets of application data or segments. For the purpose of LTP simulation, blocks in this model are formed by 8 data segments. The size of the block in the software model can be modified upon needs.

**LTP Timers**
The LTP Specification [4] defines different timers for the retransmission of lost messages or delayed data generated by other factors. Checkpoints (CP) and Report Segments (RS) set a timer every time one of these segments are sent from one engine to other. The Round Trip Time (RTT) used in this model is 2 hours and 28 minutes (8880 seconds), being an approximation of the real delay between Earth and the Cassini spacecraft surrounding Saturn and Titan. By adding an “Additional Anticipated Latency” (AAL) normal delay of 5% of the total RTT, we have an additional time of 7 minutes and 24 sec (444 sec) resulting in a total of 2 hours, 35 minutes and 24 seconds (9324 seconds). This 5% percent in time is purely experimental and can be changed by the user at any time in the omnetpp.ini file.

**Checkpoint Timer**
This Timer represented by “timeoutEvent_CP” within the Receiver_CS simple module program will be triggered immediately after any Red Data Segment with Flag 0x01, 0x02 or 0x03 including Checkpoints, is sent from the Cassini engine (transmitter) to the Earth Station engine (receiver). The number of CP timers running at the same time will depend on the number of CP generated by the transmitter engine. These timers will be “named”
as the number of the CP for which it was generated (i.e. for CP-79, the timer is the TimeoutEvent_CP[79] ) and they will have a countdown timer of 9324 seconds.

A retransmission of a Checkpoint will be generated due the expiration of the countdown timer because of the “no” reception on time of the Report Segment (RS) tied to that Checkpoint.

This event will trigger a retransmission of the DS with a CP matching the timer that was set up when the CP was sent. For the purpose of our model, a maximum number of 256 Checkpoints can be generated, with the possibility to increase it in future versions of this OMNeT based LTP simulator.

Report Segment Timer
This Timer represented by “timeoutEvent_RS” within the Receiver_ES simple module program will be triggered immediately after a Report Segment (RS) responding an specific CP is sent from the Earth Station (ES) engine (receiver) to the Cassini engine (transmitter). The number of RS timers running at the same time will depend on the number of RS generated by the receiver engine. These timers will be “named” as the
number of the RS for which it was generated (i.e. for RS-18, the timer will be timeoutEvent_RS[18]) and they will have a countdown timer of 9324 seconds.

A retransmission of a Report Segment will be generated because of these two reasons:

a) An expiration of the timer because of the “no” reception on time of the Report Acknowledgement (RS) tied to that Report Segment.
b) The reception of a CP segment without correspondence for that or previous CP’s generated by the transmitter.

Any of these events will trigger a retransmission of the RS with a RS matching the timer that was set up when the RS was sent. For the purpose of our model, a maximum number of 256 Report Segments can be generated, with the possibility to increase it in future versions of this OMNeT based LTP simulator.
Cancel Segment Timer
To enable this timer, we are using a Pointer array called “timeoutEvent_Cancel.” This timer will be set up immediately after sending a cancellation message “Cancel Segment from Block Sender (Code-#)” to the receiver engine. For the purpose of our model, we are generating this cancellation message only after have received 3 Report Segments with the same identification number.

![Diagram of Cancel Segment Timer Setup / Cancellation Process](image)

Fig. 17.- Cancel Segment Timer Setup / Cancellation Process [Fernando C.]

LTP Model Example

**LTP Transmission Process with no Channel Errors**
Having explained and defined all the important elements of the Licklider Transmission Protocol and the modules used to represent it, we can start explaining the different steps during a process of information exchange between engines. For this model, Cassini (id=3) will start the transmission of data, thinking that most important data in this kind of project is sent by the spacecraft. Let us suppose Cassini (CS) needs to transmit a block of 8 pure Red Data Segments (RDS). For this example, no Green Data Segments (GDS) will
be included, because this type of data represents no challenge due to the nature of their design, which requires no acknowledgment from receiver for the transmitted data.

Cassini (id=3) will start sending RDS one after another, in a sequence starting from zero to the total number of RDS entered at the beginning of the simulation (Total number of RDS = 8, RDS-0, RDS-1, ...,RDS-7). RDS-0 to RDS-6 will carry a flag 0x00, marking these RDS as simple RDS. Last RDS (RDS-7) will be marked as “End of Block” (EOB) and “End of Red Part” (EORP) and it will include a Checkpoint number randomly-chosen (CP-79). This RDS will change its flag to 0x03 and it will be sent to Earth Station. For this example, the Timer set up for RDS-7 (Timer CP-79) is not shown.

1 hour and 14 minutes or 4440 seconds (simulated time) later, the first RDS will be arriving to Earth Station (id = 2) which will be reading the flag of each RDS arriving at the module. RDS-0 to RDS-6 will be individually checked to detect any error in the data. The last RDS-7 (CP-79) is received and a Report Segment (RS) identified by a flag 0x08 is sent back to Cassini. This RS also includes a randomly-chosen number (RS-18) that will be tied to the CP number originally sent by Cassini. One of the fields (value3) in the RS will carry an encoded digital number with the RDS received with the error and that has to be retransmitted by Cassini. In this example, the Error Byte (value3) is empty and no retransmissions are required. Immediately after this RS (RS-18) is sent by the Earth Station, a Report Segment is set up. This RS timer will be shown later.

After a complete Round Trip Time (RTT) of 2 hours and 28 seconds (8880 seconds), the RS is received at Cassini, stopping the timer for CP-79. After this, a Report Acknowledgment (RA) for the RS-18 received will be sent back to Earth Station. This new segment with flag 0x09 will only carry the RS number to which it is responding (RS-18). Because we had no channel errors during this link, the transmission for this session, from Cassini’s perspective, is closed.

The Report Acknowledgment (RA) for the RS-18 is received 1 hour and 14 min. later (4440 sec.) at Earth Station (ES). Again, as soon as the RA (RS-18) arrives at ES, the RS-
18 timer is stopped. After approximately 3 hours and 42 minutes of data exchange between engines, this session will be closed from ES’s perspective.

![Diagram](image)

**Fig. 18.- LTP Data Exchange between Engines with no Channel Errors** [Fernando C.]

**LTP Transmission Process with Channel Errors**

The same analysis is done in the next example to show the behavior of the protocol when data traveling through space is corrupted by other conditions. Again we will be sending RDS-0 to RDS-7 in a consecutive manner from Cassini (CS). RDS-7 is marked again as CP-79. During this transmission, DS-3 and DS-5 arrive at Earth Station (ES) with error. As soon as the Report Segment for CP-79 is generated (RS-18), a calculation and encoding of the RDS received with the error is included in the “Error Byte” of the RS.

The RS-18 is sent and received at CS, and a Report Acknowledgment (RA) for the RS-18 is sent back to ES. After decoding the “Error Byte”, CS detects that DS-3 and DS-5 are claimed for retransmission. CS starts sending DS-3 and DS-7, marking this last one as CP-80, following the sequence of CP’s generated by Cassini. ES will receive successfully RA (RS-18), DS-3 and DS-7 (CP-80). ES will send a RS-19, following the sequence of
RS’s generated by Earth Station, to respond to CP-80 sent by ES. This time, “Error Byte” is empty, requiring no retransmission of any RDS.

As soon as the RS-18 (CP-80) arrives at CS, a Report Acknowledgment for the RS-19 received is sent back to ES, closing the session in CS and doing the same at ES, after the arrival of this RA at this module. In this example, approximately 6 hours and 10 minutes were needed to complete the transmission. The start and stop of the timer will be shown in later examples.

**Fig. 19.- LTP Data Exchange between Engines with Channel Errors** [Fernando C.]

**LTP Transmission Process with Channel Errors and Time out Events**

To have an LTP system with retransmissions and error detection working properly, it is necessary to enable and disable timers for each transmission and reception of CP, RS and Cancellation Events on each engine. As was mentioned previously on the Timers section, every time an engine sets up a CP timer, RS timer and Cx Timer, it will be waiting for the
reception of the RS, RA and CAx, respectively, to cancel each of these timers. If channel error occurs during the transmission of the segments and these timers are not disabled, retransmission of data is executed and it will contain the same information previously sent in the lost segment. For these examples, Cancellation segments are not shown.

The example below was simulated on the LTP simulator using 25 RDS, No GDS and a channel with an extreme error coefficient of 40%. During the transmission of the 25 RDS, we see that errors occur on RDS 11, 13, 15 (CP-80), 17 and 18. RS-18 (CP-79), RS-19 (CP-81) and RS-20 (CP-82) are sent back to Cassini. Error bytes on RS-18 and RS-20 are empty because all RDS for those blocks were received successfully. Error Byte on RS-19 is carrying retransmission request for RDS 17 and 18. Report Segment for RDS-15, CP (80) is not created because it never arrived to Earth Station. RS generated by the ES engine has a sequence. This sequence of numbers assigned to the RS will continue until the end of the process.

Timeout events for each CP and RS segment are enabled on each engine. The second round of transmissions on the CS module will include Reports Acknowledgments for RS 18,19 and 20. Retransmissions of RDS-17 and RDS-18 (CP-83) are also sent to ES, based on the reading of the Error Byte included on RS-19. In the last part of this round, RDS-15 (CP-80) is generated again and sent to ES. This is the first timeout event generated by CS due to the fact that CP-80 timer never was turned-off by any RS. Remember that a CP-80 never arrived to ES.

On the ES side, RA (RS-18) and RA (RS-20) never showed up, as well as DS-18 (CP-83) and DS-15 (CP-80). After this, time out events trigger the retransmission of RS-18(CP-79) and RS-20 (CP-82).

Back to Cassini for the third round, RA (RS-18) and RA (RS-20) are sent to ES in response to the RSs received. This time, two time out events will appear to trigger the retransmission of DS-18 (CP-83) and DS-15 (CP-80).
ES now will successfully receive RA (RS-18) and will close this session. RDS-18 and RA (RS-18) are received with errors. RDS-15 is received and will be responding with a RS-21 (CP-80). A timeout event appears and sends a RS-20 (CP-82) to CS.

Now on Cassini’s fourth round, RS-21 arrives and sends back a RA as a reply to this RS, as well a RDS-11 and RDS-13 (CP-84). A RA (RS-20) is generated and sent to ES. A time out event sends a RDS-18 again.

On Earth Station, all segments but DS-13 are received. This time, the session for RA (RS-20) is closed. A RS-22 (CP-83) is sent back to Cassini.

Cassini replies with RA (RS-22) and again with DS-13, which never was received at the ES. On ES, the session for RA (RS-22) is closed and a RS-23 is sent back to Cassini. Cassini receives the RS and replies with RA (RS-23) to finish the exchange of data between the spacecraft and earth.

This example is very helpful to understand the timers management assigned to each CP and RS, and to understand the mechanics behind the system.
Fig. 20.- LTP Data Exchange and Set up of Timers  [Fernando C.]
Analysis of Data
To analyze Data Segments exchange among Cassini and Earth Station engines, we will use a software tool offered by OMNeT called “Plove” that will help us to plot the output vector file obtained every time a LTP simulation is executed. To obtain the vector for each wanted data, a \texttt{cOutVector} object is included after each variable to analyze in the code (inside simple modules). For our model we placed these vectors at the input and output of each engine in our model, to follow the sequence based on time and the frequency of events for each segment sent and received.

After each run, OMNeT++ will create two files called omnetpp.vec and omnetpp.sca. The omnetpp.vec file contains the information (segment type, segment number and time) for each vector and required for use by the “Plove” program. The omnetpp.sca file contains the information (segment type, segment number and time) for the “Scalar” tool, which is another program application to get data in a graphical form.

Fig. 21.- Distribution of cOutVector Objects around LTP Model. [Fernando C.]
Cassini RDS output vs. Earth Station RDS input

Now we will analyze the information of the previous example, to have an idea of the results in a graphical manner. In the chart, we have the Out_CS_Red_Data_Segment_CS vector against the In_ES_Red_Data_Segment_ES vector. As we can see, there are some RDS lost during the transmission from Cassini to Earth Station. RDS 11, 13, 15, 17 and 18 transmitted by Cassini (blue line), do not appear to be received by Earth Station (red line) during the first round. On the second round, 17, 18 are sent again. 15 (RDS with CP) is sent due to a time out event. Only RDS 17 is received at the Earth Station. On the third round, 18 and 15 are sent again but only 15 is received. On the third round, RDS 18, 11 and 13 are sent. These last 2 RDS were lost by ES in the first round. ES Receives only 18 and 11 from these transmissions. In the last round, RDS 13 is resent and received successfully by the Earth Station, closing the last session of transmissions. The chart below (Fig. ) shows the exchange of the data for example mentioned.

Fig. 22.- CS RDS output vs ES RDS input vectors in LTP [Fernando C.]
Cassini Checkpoint Output vs. Earth Station Checkpoint Input

In the chart below (Fig. 24) we are sending CP’s 79-82 according the number of blocks in our example. CP-80 is lost during the transmission from CS to ES. On the second round, CP-80 is retransmitted and CP-83 is generated due to the retransmissions of RDS 17 and 18 (See Fig. 23). Both CP’s are lost during transmission. CP-80 and CP-83 are sent again due to the timer expiration on Cassini and only CP-80 is received during the third round. In the fourth round, CP-83 is sent again and CP-84 is created to resend RDS 13. This time, CP-84 is lost by ES. In the fifth round CP-84 is received successfully by the Earth Station, closing the last session among engines.

Fig. 23. - CS Checkpoint Output vs ES Checkpoint Input vectors in LTP [Fernando C.]

Earth Station Report Segment vs Cassini Report Acknowledgment

In the next chart (Fig. 25) we can see the relation between Report Segment received by Cassini (square dots) and the Report Acknowledgment received by Earth Station (round
dots). In this example we see that RS-18, RS-19, RS-20 are received at Cassini from Earth Station. After this, Cassini responds with RA (RS-18), RA (RS-19) and RA (RS-20). As is shown, RS-19 is the only segment that arrives at Earth Station. In the second round, Timers on ES trigger RS-18 and RS-20 and are sent back to Cassini, which responds with RA (RS-18) and RA (RS-20). This time only RA (RS-18) arrives at Earth Station. In the third round, a time out event triggers RS-20. On this round, RS-21 is generated as a respond to CP-80 (not shown here) that was received in RDS-15. Cassini receives these RS and sends back RA (RS-20) and RA (RS-21) to Earth Station. This time, both RA are successfully received at ES. In the fourth round RS-22 is generated as a respond to CP-83 and in the fifth round, RS-23 does the same due to the reception of CP-84. Both RS are received at Cassini, generating RA for each of them and received successfully at Earth Station, closing the sessions and ending the transmission.

![Diagram](image_url)

**Fig. 24.- CS Report Segment Input vs ES Report Ack. Input vectors in LTP** [Fernando C.]
Conclusion

With the modeling of the Licklider Transmission Protocol, using the OMNeT++ open source tool for with this purpose, we are now able to have a clear idea of how a real transmission can operate in outer space. Delay, data rate and error factors are used here to obtain a model that can offer us an approach of real transmission using this new protocol for Delay Tolerant Networks (DTN). With the information extracted from the data using different values of parameters mainly in the Data Error Rate, we see the importance of the use of transmission forecasting to avoid long delays and retransmissions of segments.

An important implementation of the LTP protocol is the inclusion of Timers for Checkpoints, Report Segments and Cancellation. During the programming of the LTP simulation model, the hardest part was the implementation of these timers by using array pointers and my introduction to the C++ world, starting with the use of a new concept for me called “Classes” in C++. The program for this model was initially thought as we would be transmitting “real” data over deep space, coding a decoding 32 bit data stream as the core structure segment of the model. This way of thinking was an enjoyable mental exercise but the resulting implementation was quite large. C++ and OMNeT allow us to improve this version by the use of classes and functions that can be added in a later version of this model.

This design, based on the OMNeT discrete event simulation framework can be used, with some modifications, to model other space network applications in the middle distance range, like inter-satellite communications or earth-to-moon data transmission.
Bibliography:


Addendum

LTP Code Files

Transmitter_ES.cpp

// This file is part of Licklider Transmission Protocol Simulation based on OMNet++.
//
// Copyright (C) 2007 Fernando Carrasco
//
// This file is distributed WITHOUT ANY WARRANTY.
//
#include <string.h>
#include <omnetpp.h>
#include <iostream>
#include <string>
using namespace std;

class Transmitter_ES : public cSimpleModule {
private:
    int value1, value2, value3, value4, Core, flags;  //Declaring values of Core segment
    int Report_Segment_ES; // This is the last value of the RS (value1) before be sent it by the Earth Station. To be followed by the 'Watch' function
    int CheckPoint_ES; // This is the last value of the CP (value2) before be sent it by the Earth Station. To be followed by the 'Watch' function

    // Setting Histogram and Vectors for statistical purposes
    cLongHistogram Out_Report_Segment_ES_Stats;
    cOutVector Out_Report_Segment_ES_Vector;
    cLongHistogram Out_CheckPoint_ES_Stats;
cOutVector Out_CheckPoint_ES_Vector;

public:
    int Red_data_segments;
    int Green_data_segments;

protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
};

Define_Module(Transmitter_ES);

void Transmitter_ES::initialize()
{
    Out_Report_Segment_ES_Stats.setName("Out_ES_Report_Segment_ES_Stats");
    Out_Report_Segment_ES_Vector.setName("Out_ES_Report_Segment_ES_Vector");
    Out_CheckPoint_ES_Stats.setName("Out_ES_CheckPoint_ES_Stats");
    Out_CheckPoint_ES_Vector.setName("Out_ES_CheckPoint_ES_Vector");
}

void Transmitter_ES::handleMessage(cMessage *msg)
{
    unsigned int Core = msg->kind();
    ev << "Core --> " << Core << " \n\n";
    msg->senderModuleId();

    Core = Core & 0xFFFFFFFF;
    flags = Core & (~0x07FFFFFF);  // mask out all flags
    flags = (flags >> 27) & 0x0F;  // Shift the bits 28 bits to the right (to the Less Significant Nibble)
    and clear

    value1 = Core & (~0xFFFFFF00);  // mask out the value 1
    value2 = Core & (~0xFFFF00FF);  // mask out the value 2
    value2 = value2 >> 8;
    value3 = Core & (~0xFF00FFFF);  // mask out the value 3
    value3 = value3 >> 16;
    value4 = Core & (~0xF8FFFFFF);  // mask out the value 4
value4 = value4 >> 24;

ev << "*********************************************************************************************************
 Flags --> " << flags << " || Value 4 --> " << value4 << " || Value 3 --> " << value3 << " || Value 2 --> " << value2 << " || Value 1 --> " << value1 << "
*********************************************************************************************************

switch ( flags )
{
    case 0x08: //  8 ----> Report Segment (RS)
        
        
        Report_Segment_ES = value1;
        CheckPoint_ES = value2;
        Out_Report_Segment_ES_Vector.record(Report_Segment_ES); // Updating Statistics of the number of Report Segments Sent
        Out_Report_Segment_ES_Stats.collect(Report_Segment_ES); // Updating Statistics of the number of Report Segments Sent
        
        break;

    case (0x08 | 0x04): // 12 ----> Cancel Segment from Block Sender (CS)
        
        Red_Data_Segment_CS = value3;
        Out_CheckPoint_ES_Vector.record(CheckPoint_ES); // Updating Statistics of the number of CheckPoints Sent
        Out_CheckPoint_ES_Stats.collect(CheckPoint_ES); // Updating Statistics of the number of CheckPoints Sent
        
        break;
}
Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent

Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent

case (0x08 | 0x05):   // 13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)
{
    CANCEL ACKNOWLEDGMENT TO BLOCK SENDER (CAS)
    ev << "Cancel Acknowledgment Segment to Block Sender (CAS) \n";
    //Red_Data_Segment_CS = value3;
    Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
}
break;

case (0x08 | 0x06):   // 14 ----> Cancel Segment from Block Receiver (CR)
{
    CANCEL SEGMENT FROM BLOCK RECEIVER (CR)
    ev << " Cancel Segment from Block Receiver (CR) \n";
    //Red_Data_Segment_CS = value3;
    Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
}
break;

case (0x08 | 0x07):   // 15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)
{
    CANCEL ACKNOWLEDGMENT SEGMENT TO BLOCK RECEIVER (CAR)
    ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
    //Red_Data_Segment_CS = value3;
    Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent
/Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red Data Segments Sent

default:
    ev << " UNDEFINED!! \n";
break;

//

DEFAULT_SENDER_CS:

Sending  " << msg << " from Earth Station to Cassini \n";
send(msg, "out");

Receiver_ES.cpp

// This file is part of Licklider Trasmission Protocol Simulation based on OMNeT++.
//
// Copyright (C) 2007 Fernando Carrasco
//
// This file is distributed WITHOUT ANY WARRANTY.
//

#include <string.h>
#include <omnetpp.h>
#include <iostream>
#include <string>
using namespace std;

/////////////////////////////////////////////////////////////////////////////////////////
class Receiver_ES : public cSimpleModule
{
private:
    int i,d,j,k,l,m,o,p,w,q,y; // Different variables used by counters and loops
    unsigned int Core;        // Core Segment Value
    unsigned int RS_Backup_Core;
    int CPBlocks[256];        // Keeps in memory the number of Checkpoint Blocks
    int RS_backup[256];       // Keeps in memory the number of Report Segments
    int flags;
    int value1,value2,value3,value4;
    bool ds_error;
    int segment_lost[2048];   // Keeps in memory the number of segment lost
    int CP_counter;
    int Data_Segment_Error;
    int DS_Counter;
    int First_RS;
    int RS_counter;
    bool Last_DS_Received;   // to mark the data segment with flag 0x03 as "last data segment received"
    int Rep_Ack_Counter;
    int Core_temp;
    unsigned int RS_ReTx;
    int Last_CP_to_resent;
    int DS_Error_Counter;
    int Report_Segment_ES;   // This is the last value of the RS (value1) before be sent it by
    int CheckPoint_ES;        // This is the last value of the CP (value2) before be sent it by
    int Error_Byte_ES;        // This is the last value of the Error_Byte (value3) before be sent it by
    int Reserved_Value_ES;    // This is the last value of the Reserved_Value (value4) before be sent it by
    int Red_Data_Segment_ES;  // This is the last value of the RDS (value4) before be sent it by
    int Green_Data_Segment_ES; // This is the last value of the GDS (value3) before be sent it by
    int Report_Acknowledgment_ES; // This is the last value of the RA (value1) before be sent it by
    int CP_count_Statist;
    int DS_from_Cassini;
}
double timeout;                          // timeout
cMessage *event;                       // pointer to the event object which we'll use for timing
cMessage *timeoutEvent_RS[256];

//cLong Histogram DS_Error_Stats;
//cOut Vector DS_Error_Vector;

////////////////// Setting Histogram and Vectors for statistical purposes //////////////////
cLong Histogram Red_Data_Segment_ES_Stats;
cOut Vector Red_Data_Segment_ES_Vector;
cLong Histogram Green_Data_Segment_ES_Stats;
cOut Vector Green_Data_Segment_ES_Vector;
cLong Histogram Report_Segment_ES_Stats;
cOut Vector Report_Segment_ES_Vector;
cLong Histogram CheckPoint_ES_Stats;
cOut Vector CheckPoint_ES_Vector;
cLong Histogram Report_Acknowledgment_ES_Stats;
cOut Vector Report_Acknowledgment_ES_Vector;

public:
int Core_ES;
int CP_ES;
int Red_data_segments;
int Red_data_segments_counter;
int Green_data_segments;
int Green_data_segments_counter;
int Error_probability;

Receiver_ES();
virtual ~Receiver_ES();

protected:
virtual void initialize();
virtual void handleMessage(cMessage *msg);
virtual void finish();

Define_Module(Receiver_ES);              // Defining the module
to work with OMNET++
Constructor and Destructor for the LTP Timer

Receiver_ES::Receiver_ES()
{
    timeoutEvent_RS[256] = NULL;
    event = NULL;
}

Receiver_ES::~Receiver_ES()
{
    cancelAndDelete(timeoutEvent_RS[256]);
    cancelAndDelete(event);
}

:: initialize!!

void Receiver_ES::initialize()
{
    Last_DS_Received = 0;
    Rep_Ack_Counter = 0;
    //timeout = 9324.0;          // Maximum timeout. RTT + 5% to know if we
    lost the CP. (74min * 2 * 60 = 8880+444 = 9324 seconds)
    DS_Error_Counter = 0;
    double timeout = par("timeout");
    // Maximum timeout. RTT + 5% to know if we
    WATCH(RS_counter);
    WATCH(Report_Segment_ES);
    WATCH(CheckPoint_ES);
    WATCH(Error_Byte_ES);
    //DS_Error_Stats.setName("DS_Error_Stats");
    //DS_Error_Stats.setRangeAutoUpper(0, 10, 1.5);
    //DS_Error_Vector.setName("DS_Error");
    //DS_Error_Stats.collect(DS_Error_Counter);
    //DS_Error_Vector.record(DS_Error_Counter);
    Red_Data_Segment_ES_Stats.setName("In_ES_Red_Data_Segment_ES_Stats");
    Red_Data_Segment_ES_Vector.setName("In_ES_Red_Data_Segment_ES_Vector");
    Green_Data_Segment_ES_Stats.setName("In_ES_Green_Data_Segment_ES_Stats");
    Green_Data_Segment_ES_Vector.setName("In_ES_Green_Data_Segment_ES_Vector");
    Report_Segment_ES_Stats.setName("In_ES_Report_Segment_ES_Stats");
Report_Segment_ES_Vector.setName("In_ES_Report_Segment_ES_Vector");
CheckPoint_ES_Stats.setName("In_ES_CheckPoint_ES_Stats");
CheckPoint_ES_Vector.setName("In_ES_CheckPoint_ES_Vector");
Report_Acknowledgment_ES_Stats.setName("In_ES_Report_Acknowledgment_ES_Stats");
Report_Acknowledgment_ES_Vector.setName("In_ES_Report_Acknowledgment_ES_Vector");
}

////////////////////////////////////////////////////////////////////////////////:: handleMessage!! /////////////////////////////////////////////////////////////////////////

void Receiver_ES::handleMessage(cMessage *msg)
{
    unsigned int Core = msg->kind();
    unsigned int Core_Timer = msg->kind();
    int msg_length = msg->byteLength();
    ev << " Core ---------------> " << Core << "  
    ev << " Message Length ----> " << msg_length << "..
    int ds_error = msg->hasBitError();
    double d = simTime()-msg->timestamp();
    int CS_id = msg->senderModuleId();
    int module_id = msg->arrivalModuleId();
    int j = 0;
    int k = 0;
    int p = 0;
    int w = 0;
    int q = 0;
    int y = 0;
    int lost_counter;
    int Error_byte[8];
    int DS_CP_block;
    int DS_bit;
    int RS_index;
    int Last_RS_sent;

    int Error_probability = par("Error_probability");
    int First_RS = par("First_RS");
    double timeout = par("timeout");

    if(DS_Counter==0)
        {  // initialize 'CP_counter' one time
            int CP_counter=0;  // during the program.
int DS_Counter = 0;
RS_Backup_Core = 0;
int Core_temp = 0;
int Core_ReTx = 0;
int RS_ReTx = 0;
int Last_CP_to_resent = 0;
int CP_count_Statist = 0;
}

if (msg-> isSelfMessage())     // If the Message received was generated by the same Module (Timing), this

section is triggered!!
{
    ev << " / / / / / / / / / / / / / /     S E L F - M E S S A G E   T R I G G E R E D    @    E A R T H   S T A T I O N    / /
    / / / / / / / / / / / / / / / /     

    ev << " Receiving ----> " << msg->name() << ".\n"
    //unsigned int Core = msg->kind();
    //unsigned int Core_Timer = msg->kind();
    msg->senderModuleId();
    Core = Core & 0xFFFFFFFF;     // mask out 32-bits
    flags = Core & (~0x07FFFFFF);    // mask out all flags
    flags = (flags >> 27) & 0x0F;    // Shift the bits 27 bits to the right (to the Less Significant Nibble)
    and clear
    // any remaining bits in the left of the
    nibble
    value1 =  Core & (~0xFFFFFF00);    // mask out the value 1
    value2 =  Core & (~0xFFFF00FF);    // mask out the value 2
    value3 =  Core & (~0xFF00FFFF);    // mask out the value 3
    value4 =  Core & (~0xF8FFFFFF);    // mask out the value 4
    
    ev << "*********************************************************************************************************
    Flags --> " << flags << "   ||   Value 4 --> " << value4 << "   ||   Value 3 --> " << value3 << "   ||   Value 2 --> " << value2 << "   ||   Value 1 --> " << value1 << "  
    *********************************************************************************************************

    //

    //

    //

    //
Internal Management Code - Designed to identify the type of Timer arriving

Checkpoint - Timer_RS Self-Message ---> Flags = 0x08
Cancel - Timer_CAX Self-Message ---> Flags = 0x04
Schedule - Timer_SC Self-Message ---> Flags = 0x01

switch ( flags )
{ case 0x08: // Self-Message for the Report Segment RS Timer
    for(q=First_RS;q<(First_RS+RS_counter);q++)
    {
        if (q == value1)
        {
            char msgname[32];
            sprintf( msgname, "Timer RS-%d",q);    //assigns "Timer RS-#" to
            unsigned int RS_ReTx = msg->kind();
            ev << RS_backup[RS_ReTx] = "<< RS_backup[RS_ReTx] << \"\n\n\";
            Core_temp = RS_backup[RS_ReTx];
            Core_ReTx = RS_backup[RS_ReTx];
            Core_ReTx = Core_ReTx & 0xFFFFFFFF;    // mask out 32-bits
            flags = Core_ReTx & (~0xFFFFFF00);   // mask out the value 1
            flags = (flags >> 8);
            value1 = Core_ReTx & (~0xFFFF00FF);   // mask out the value 2
            value1 = value1 >> 16;
            value2 = Core_ReTx & (~0xF8FFFFFF);   // mask out the value 3
            value2 = value2 >> 24;
            char Rep_Seg[32];

Page 54 of 144
sprintf( Rep_Seg, "Report Segment-%d, (CP-%d)",value1,value2);
cMessage *msg = new cMessage(Rep_Seg);

ev << "Sending " << Rep_Seg << " to Cassini \n\n";

//msg->setTimestamp(System_Clock++);   // Setting the time stamp for this DS
msg->setByteLength(1024);      // Setting 1 KB

message length (1024*8=8192 bits)

of current module

module ID of the module to send

msg->senderModuleId();  // Sets the module ID
msg->arrivalModuleId();      // Sets the
msg->setKind(Core_temp);
msg->setBitError(ds_error);

ev <<
Ev << "Retransmitting Lost Report Segment \n";
bubble("Retransmitting Lost Report Segment! ");
send(msg, "out");
ev << "Core ---> " << Core " ....\n";

Flags = 0x08 ---> means Report Segment Timer

Flags = 0x08 ---> means Report Segment Timer

value2 = 0;
value3 = 0;
value4 = 0;

char timermsg[32];
sprintf( timermsg, "Timer RS-%d",value1);

assigns "Data Segment-#" to buffer 'timermsg'
`endl` flush the buffer
message 'msg' from 'timermsg'
timeoutEvent_RS[value1] = new cMessage(timermsg); // We create a 
value1 = (value1 & 0xFFFFFFFF); //
value2 = value2 << 8; // Shifting
value2 = (value2 & 0xFFFFFFFF); //
value_to_send

Core = (flags << 27) | value4 | value3 | value2 | value1; // add the 4-bit flags , Value2 is zero, RA just have RS number
Core 1 ------>  Core

Core 2 ------>  Core

E N D   O F   T I M E R   S E T U P

break;

break;
case 0x04:  // Self-Messages for the Cancel CAX Timer
    ev << " Cancelling CX Timer ---> timeoutEvent_Cancel[ " << value2 << " ] \n\n";
break;

default:

    ev << "SELF - MESSAGE UNDEFINED!! \n";

    break;

    }  // Switch Brace

    // Delete the isSelfmessage after being received. Helps to avoid error messages in the command prompt

    }

else
{

   ////////////////////////////////////////////////
    // E A R T H   S T A T I O N   -   R E C E I V I N G    D A T A    S E G.   W I T H    E R R O R S
    //////////////////////////////////////////////////

    msg->senderModuleId();

    Core = Core & 0xFFFFFFFF;     // mask out 32-bits
    flags = Core & (~0x07FFFFFF);    // mask out all flags
    flags = (flags >> 27) & 0x0F;    // Shift the bits 27 bits to the right (to the Less Significant Nibble)
    and clear

    // any remaining bits in the left of the nibble
    value1 =  Core & (~0xFFFFFF00);    // mask out the value 1
    value2 =  Core & (~0xFFFF00FF);    // mask out the value 2
    value2 = value2 >> 8;
    value3 =  Core & (~0xFF00FFFF);    // mask out the value 3
    value3 = value3 >> 16;
    value4 =  Core & (~0xF8FFFFFF);    // mask out the value 4
    value4 = value4 >> 24;

    ev << "*********************************************************************************************************
    Flags --> " << flags << "     ||    Value 4 --> " << value4 << "      ||    Value 3 --> " << value3 << "     ||    Value 2 --> " << value2 << "      ||    Value 1 --> " << value1 << " 
    *********************************************************************************************************\n\n";
DS_from_Cassini = value3; // Assignin the value of Data Segment
DS_CP_block = value3 / 8; // to get the correspondent block for each check point
DS_bit = value3 % 8;     // to get the correspondent bit of each block

if(flags == 0x03)       // If the RDS is received as a CP, EORP, Core
    ev << " LAST DATA SEGMENT RECEIVED !!!! LAST DATA SEGMENT RECEIVED !!!! LAST DATA SEGMENT RECEIVED !!!!\n\n";

/////////////////////////////////////////////////////////////////////////////////////////////////
//
// This loop is to detect the block were Data Segment must request retransmission
//
//<--|  DS 13 Here! |-->    
//
//| BLOCK 0 | BLOCK 1 | BLOCK 2 |
//
//| 0 1 2 3 4 5 6 7 8 | 91011121314151617181920212223 .........................
//|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|________|_____
```c
// if ((uniform(0,1) < (0.01 * Error_probability)) || (flags == 0x09 && value1==18))  // Here, I'll receive Data Segments with Error
if ((uniform(0,1) < (0.01 * Error_probability)) )  // Here, I'll receive Data Segments with Error
{
    bubble("Data Segment Received with Errors...!!");
    DS_Error_Counter++;
}
```
switch ( flags )
{
    ev << msg ->name() << " R e c e i v e d w i t h E r r o r ! \n\n";
    ev << " D a t a S e g m e n t w i t h C h e c k p o i n t ---> " << value2 << " R e c e i v e d w i t h E r r o r ! ! ! \n\n";
    case 0x00:       // 0 0 0
        0   0 ----> Red Data, NOT(CP, EORP, EOB)
        {  
            /////////////// R E D   D A T A ,   N O T ( C P ,   E O R P ,   E O B )
            /////////////////
            ev << msg ->name() << " R e c e i v e d w i t h E r r o r ! \n\n";
            ev << " D a t a S e g m e n t w i t h C h e c k p o i n t ---> " << value2 << " R e c e i v e d w i t h E r r o r ! ! ! \n\n"
            " << value2 << " R e c e i v e d w i t h E r r o r ! ! ! \n\n";
            for (p=0;p<8;p++)
                {  
                    if(p==DS_bit)
                        {
                            segment_lost[DS_from_Cassini]= (0x01 << p);
                        }
                    p=0;
                }
        break;
    case 0x01 :
        {  
            /////////////// R E D   D A T A ,   C P ,   N O T ( E O R P   O R   E O B ) ...... W I T
            H E R R O R ! ! ! /////////////////
        }
ev << msg ->name() << " Received with Error ! \n\n";
            ev << " Data Segment -------> " << value3 << " with Checkpoint --->
" << value2 << " Received with Error ! ! ! \n\n";
            for (p=0;p<8;p++)
            {
                if(p==DS_bit)
                {
                    segment_lost[DS_from_Cassini]=(0x01 << p);
                }
            }
            p=0;
            flags = ( 0x00 );
            // Reseting the Flag
            Core=0;
            // I set to zero Core,
            CP_Blocks[CP_counter] = value2;
            CP_counter++;
            value3 = 0;
            value4 = 0;
            // value 4 to zero.... this is empty now.
            break;
            case 0x02 :
            {
                // RED DATA, CP, EORP, NOT EOB ...... WITH ERR
                ev << msg ->name() << " Received with Error ! \n\n";
                ev << " Data Segment -------> " << value3 << " with Checkpoint --->
" << value2 << " Received with Error ! ! ! \n\n";
                for (p=0;p<8;p++)
                {
                    if(p==DS_bit)
                    {
                        segment_lost[DS_from_Cassini]=(0x01 << p);
                    }
                }
                p=0;
flags = ( 0x00 ); // Reseting the Flag
Core=0; // I set to zero Core,
CP_Blocks[CP_counter] = value2; // Assign the value of check point to the array

CP_Blocks[]

CP_counter++;
value3 = 0;

value4 = 0; // value 4 to zero.... this is empty
now.
break;
}
case 0x03 :
{

////////
ev << msg ->name() << " R e c e i v e d w i t h   E r r o r ! \n\n";
ev << " D a t a   S e g m e n t     -------> " << value3 << "   w i t h   C h e c k p o i n t   --->
" << value2 << "   R e c e i v e d   w i t h   E r r o r   ! ! !  \n\n";

for (p=0;p<8;p++)
{
    if(p==DS_bit)
    {
        segment_lost[DS_from_Cassini]=(0x01 << p);
    }
}
p=0;

flags = ( 0x00 ); // Reseting the Flag
Core=0; // I set to zero Core,
CP_Blocks[CP_counter] = value2; // Assign the value of check point to the array

CP_Blocks[]

CP_counter++;
value3 = 0;

value4 = 1; // Setting value4 as 1. Last RS to send
break;

    case 0x04:       // 0 1 0 0
4 ----> Green Data, NOT EOB
    {
        //////////////////////////////////////////////////////
        ev << msg ->name() << " Received with Error! \n\n";
        ev << " Data Segment -------> " << value3 << " with Checkpoint --->
        " " << value2 << " Received with Error ! ! ! \n\n";
        ev << "Green Data Received, NOT EOB --- WITH ERROR \n";
    }
break;

    case 0x07:   // 0 1 1 1
7 ----> Green Data, EOB
    {
        //////////////////////////////////////////////////////
        ev << msg ->name() << " Received with Error! \n\n";
        ev << " Data Segment -------> " << value3 << " with Checkpoint --->
        " " << value2 << " Received with Error ! ! ! \n\n";
        ev << "Green Data Received, EOB --- WITH ERROR \n";
    }
break;

    case (0x08 | 0x01):     // 1 0 0 1
9 ----> Report Acknowledgment (RA)
    {
        //////////////////////////////////////////////////////
        Core=0;
        ev << msg ->name() << " Received with Error! \n\n";
        ev << " Data Segment -------> " << value3 << " with Checkpoint --->
        " " << value2 << " Received with Error ! ! ! \n\n";
        ev << " End of Transmission, Session Closed ! \n";
    }
break;

    default:
    {
        ev << " ERROR UNDEFINED ! ! \n";
    }
break;

}
Earth Station – End of Receiving Data Seg. With Errors

else

Earth Station – Receiving Data Seg. Without Errors

SEGMENT TYPE CODES

0 ----> Red Data, NOT(CP, EORP, EOB) ----> For testing purposes!!

1 ----> Red Data, CP, NOT (EORP, EOB)

2 ----> Red Data, CP, EORP, NOT EOB

3 ----> Red Data, CP, EORP, EOB ----> For testing purposes!!

4 ----> Green Data, NOT EOB

7 ----> Green Data, EOB

8 ----> Report Segment (RS)

9 ----> Report Acknowledgment (RA)

12 ----> Cancel Segment from Block Sender (CS)
// 13 ----> Cancel Acknowledgment Segment to Block Sender (CAS) //
// 14 ----> Cancel Segment from Block Receiver (CR) //
// 15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR) //

// 1 bit 4 bits 3 bits 8 bits 8 bits 8 bits ---> 32 bits
//
// __________ ___________ __________ __________ __________ __________
//| Reserved |__flags___|__value4__|__value3__|__value2__|__value1__|
//    |     |             \
//    \
// Future Use |DS Number |CP Number |RS Number |

// Simulations

// Statistics Collection

Page 65 of 144
switch ( flags )
{
    case 0x00: // 0 ----> Red Data, NOT(CP, EORP, EOB)
    {
        // Red Data, NOT(CP, EORP, EOB)
        //ev << "Red Data Received and Registered \n\n";
        //CheckPoint_ES = value2;
        Red_Data_Segment_ES = value3;
        //Updating Statistics of the number of Red Data Segments Sent
        Red_Data_Segment_ES_Vector.record(Red_Data_Segment_ES);
        Red_Data_Segment_ES_Stats.collect(Red_Data_Segment_ES);
        //Updating Statistics of the number of CheckPoint Sent
        CheckPoint_ES_Vector.record(CheckPoint_ES);
        CheckPoint_ES_Stats.collect(CheckPoint_ES);
    }
    break;
    case 0x01: // 1 ----> Red Data, CP, NOT(EORP, EOB)
    {
        // Red Data, CP, NOT(EORP, EOB)
        //ev << "Red Data Received and Registered \n\n";
        CheckPoint_ES = value2;
        Red_Data_Segment_ES = value3;
        //Updating Statistics of the number of Red Data Segments Sent
        Red_Data_Segment_ES_Vector.record(Red_Data_Segment_ES);
        Red_Data_Segment_ES_Stats.collect(Red_Data_Segment_ES);
        //Updating Statistics of the number of CheckPoint Sent
        CheckPoint_ES_Vector.record(CheckPoint_ES);
        CheckPoint_ES_Stats.collect(CheckPoint_ES);
    }
    break;
}
case 0x02: // 2 ----> Red Data, CP, EORP, NOT EOB
{
    
    /////////////////////////////////////////////////////////
    RED DATA, CP, EORP (NOT EOB)
    
    //////////////////////
    //ev << "RED DS RECEIVED AND REGISTERED \n\n";
    CheckPoint_ES = value2;
    Red_Data_Segment_ES = value3;
    
    /////////////////////////////////////////////////////////////////////////////
    Red_Data_Segment_ES_Vector.record(Red_Data_Segment_ES); // Updating Statistics of the number
    of Red Data Segments Sent
    Red_Data_Segment_ES_Stats.collect(Red_Data_Segment_ES); // Updating Statistics of the number
    of Red Data Segments Sent
    CheckPoint_ES_Vector.record(CheckPoint_ES); // Updating Statistics of the
    number of CheckPoint Sent
    CheckPoint_ES_Stats.collect(CheckPoint_ES); // Updating Statistics of the
    number of CheckPoint Sent
    /////////////////////////////////////////////////////////////////////////////
}
break;  

case 0x03: // 3 ----> Red Data, CP, EORP, EOB
{
    
    /////////////////////////////////////////////////////////
    RED DATA, CP, EORP, EOB /////////////////////////////////////////////////////////
    //ev << "RED DS RECEIVED AND REGISTERED \n\n";
    CheckPoint_ES = value2;
    Red_Data_Segment_ES = value3;
    
    /////////////////////////////////////////////////////////////////////////////
    Red_Data_Segment_ES_Vector.record(Red_Data_Segment_ES); // Updating Statistics of the number
    of Red Data Segments Sent
    Red_Data_Segment_ES_Stats.collect(Red_Data_Segment_ES); // Updating Statistics of the number
    of Red Data Segments Sent
    CheckPoint_ES_Vector.record(CheckPoint_ES); // Updating Statistics of the
    number of CheckPoint Sent
    CheckPoint_ES_Stats.collect(CheckPoint_ES); // Updating Statistics of the
    number of CheckPoint Sent
    /////////////////////////////////////////////////////////////////////////////
}
break;  

case 0x04: // 4 ----> Green Data, NOT EOB
{
    
    /////////////////////////////////////////////////////////
    GREEN DATA, NOT EOB /////////////////////////////////////////////////////////
Simulation of LPT in OMNet++

```cpp
    //ev << "GREEN DATA RECEIVED AND REGISTERED \n\n";
    Green_Data_Segment_ES = value3;

    // Updating Statistics of the number of Red Data Segments Sent
    Green_Data_Segment_ES_Vector.record(Green_Data_Segment_ES);
    Green_Data_Segment_ES_Stats.collect(Green_Data_Segment_ES);

    case 0x07:   //  7 ----> Green Data, EOB
    {
      //ev << "GREEN DATA, EOB \n\n";
      Green_Data_Segment_ES = value3;

      // Updating Statistics of the number of Red Data Segments Sent
      Green_Data_Segment_ES_Vector.record(Green_Data_Segment_ES);
      Green_Data_Segment_ES_Stats.collect(Green_Data_Segment_ES);
    }
    break;

    case 0x08:   //  8 ----> Report Segment (RS)
    {
      //ev << "REPORT SEGMENT \n\n";
      // Report_Segment_ES = value1;

      // Updating Statistics of the number of Red Data Segments Sent
      Report_Segment_ES_Vector.record(Report_Segment_ES);
      Report_Segment_ES_Stats.collect(Report_Segment_ES);
    }
    break;

    case 0x09:   //  9 ----> Report Acknowledgment (RA)
    
```
{  
    // REPORT ACKNOWLEDGMENT RECEIVED AND REGISTERED
    ev << "REPORT ACKNOWLEDGMENT RECEIVED AND REGISTERED
";
    Report_Acknowledgment_ES = value1;
    Report_Acknowledgment_ES_Vector.record(Report_Acknowledgment_ES);  // Updating Statistics of the number of Red Data Segments Sent
    Report_Acknowledgment_ES_Stats.collect(Report_Acknowledgment_ES);  // Updating Statistics of the number of Red Data Segments Sent

    break;
}

case (0x08 | 0x04):     // 12 ----> Cancel Segment from Block Sender (CS)
{
    // CAN C E L   S E G M E N T   F R O M   B L O C K   S E N D E R   (CS)
    //ev << "Cancel Segment from Block Sender (CS) 
";
    //Red_Data_Segment_CS = value3;
    //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
    //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent

    break;
}

case (0x08 | 0x05):   // 13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)
{
    //ev << "Cancel Acknowledgment Segment to Block Sender (CAS) 
";
    //Red_Data_Segment_CS = value3;
    //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
    //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent

    break;
}
break;

case (0x08 | 0x06):    // 14 ----> Cancel Segment from Block Receiver (CR)
    {
        // CANCEL SEGMENT FROM BLOCK RECEIVER (CR)
        //ev << " Cancel Segment from Block Receiver (CR) \n";
        //Red_Data_Segment_CS = value3;
        //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
        //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
    }
    break;

case (0x08 | 0x07):    // 15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)
    {
        // CANCEL ACKNOWLEDGMENT SEGMENT TO BLOCK RECEIVER (CAR)
        //ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
        //Red_Data_Segment_CS = value3;
        //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
        //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
    }
    break;

default:
    ev << " UNDEFINED!! \n";
    break;

}
switch (flags)
{
    case 0x00:    // 0  0  0
        0   0 ----> Red Data, NOT(CP, EORP, EOB)
        Red_data_segments_counter++;
";
        segment_lost[DS_from_Cassini]=(0x00);
        break;

    case 0x01:    // 0  0  0
        1   1 ----> Red Data, CP, NOT (EORP, EOB)
        Red_data_segments_counter++;
        CP_count_Statist++;
        flags = (0x00);    // Reseting the Flag
        ev << "R E D    D S   R E C E I V E D  ,   N O T, (C P,   E O R P ,    E O B ) \n\n"
        segment_lost[DS_from_Cassini]=(0x00);
        CP_Blocks[CP_counter] = value2;   // Save the value of Checkpoint Number in
        CP_counter++;
        value3 = 0;
        value4 = 0;    // value 4 to zero.... this
        is empty now.
value1 = First_RS;      // Getting the Initial Report Segment
value1 = value1 + RS_counter;   // Assigning the value of the initial Report Segment number

RS_counter. This is to increase the RS number (18,19,20,21, ....~)

cchar Rep_Seg[32];
sprintf( Rep_Seg, "Report Segment-%d, (CP-%d)",value1,value2);    // RS Random number... just testing
cMessage *msg = new cMessage(Rep_Seg);
ev << "Sending " << Rep_Seg << " to Cassini \n\n";

ev << "*********************************************************************************************************

Flags --> " << flags << "     ||    Value 4 --> " << value4 << "     ||    Error Byte (Value3)--> " << value3 << "     ||    CheckPoint (Value2) --> " << value2 << "     ||    Report Segment (Value1) --> " << value1 << "

*********************************************************************************************************\n\n";

l=m;
m=m+8;

for(k=l;k<m;k++)
{
    value3 = value3 | segment_lost[k];
    if (segment_lost[k]!=0)
        ev << "C P ----> " << value2 << " Data Segment : " << k << " ---> received with error!! \n\n";
    else
        ev << "C P ----> " << value2 << " Data Segment : " << k << " ---> OK ! \n\n";
}

ev << " Value to request retransmission ----------> " << value3 << " \n\n";

value4 = 0;

value1 = (value1 & 0xFFFFFFFF);      // value1 to send
value2 = value2 << 8;        // Shifting bits to the left
value2 = (value2 & 0xFFFFFFFF);      // value2 to send
value3 = value3 << 16;        // Shifting bits to the left
value3 = (value3 & 0xFFFFFFFF);      // value3 to send
value4 = value4 << 24;  // Shifting bits to the left
value4 = (value4 & 0xFFFFFFFF);  // value4 to send
flags = ( 0x08 );  // Flag - Report Segment

Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
if((value2>>8)>=Last_CP_to_resent) // If the CP to send is greater than the CP sent, then the RS is not a Retransmission
    Last_CP_to_resent = value2>>8;  // This is to follow the CP sent by the Earth Station. Needed in the "isSelfmessage"

RS_Backup_Core = Core;
msg->setKind(Core);
msg->setByteLength(1024);  // Setting 1 KB message length (1024*8=8192 bits)
bubble("Sending RS to Cassini");
send(msg, "out");

RS_backup[value1] = RS_Backup_Core;
flags = ( 0x08 );  // set 'flags' to 0x80 (Internal Management Code. Flags - 0x08 ---> means Report Segment Timer

value2 = 0;
value3 = 0;
value4 = 0;
char timermsg[32];
sprintf( timermsg, "Timer RS-%d",value1);  // assigns "Timer RS-#" to buffer 'msgname'
ev << "Generating " << timermsg << endl;
'endl' flush the buffer
timeoutEvent_RS[value1] = new cMessage(timermsg); // We create a message 'msg' from 'msgname'

ev << "\n";

************************************************************************************************************

ev << "Flags --> " << flags << " || Value 4 --> " << value4 << " || Value 3 --> " << value3 << " || Value 2 --> " << (value2>>8) << " || Value 1 --> " << value1 << " \n";

************************************************************************************************************

Core = (flags << 27) | value4 | value3 | value2 | value1; // add the 4-bit flags, Value2 is zero, RA just have RS number

timeoutEvent_RS[value1]->setKind(Core);

scheduleAt(simTime()+ timeout, timeoutEvent_RS[value1]);

if(timeoutEvent_RS[value1]->isScheduled())
  ev << "TimeoutEvent_RS [" << timeoutEvent_RS[value1]->name() << "] is Scheduled!! \n\n";
else
  ev << "TimeoutEvent_RS [" << timeoutEvent_RS[value1]->name() << "] NOT Scheduled!! \n\n";

RS_counter++;

value3 = 0;

break;

case 0x02: // 0 0 1
  Red Data, CP, EORP, NOT EOB
  {
    RED DATA, CP, EORP (NOT EOB)
    Red_data_segments_counter++;
  }
CP_count_Statist++;  
flags = ( 0x00 );  // Reseting the Flag
ev << "R E D    D S   R E C E I V E D  ,  C P,  E O R P, ( N O T   E O B ) \n\n";
segment_lost[DS_from_Cassini] = (0x00);

value3 = 0;
value4 = 0;  // value 4 to zero.... this is empty now.
value1 = First_RS;
value1 = value1 + RS_counter;

random number... just testing

char Rep_Seg[32];
sprintf( Rep_Seg, "Report Segment-%d, (CP-%d)",value1,value2);
// RS
cMessage *msg = new cMessage(Rep_Seg);

l=m;
for(k=l;k<m;k++)
{
    value3 = value3 | segment_lost[k];
    if (segment_lost[k]!-0)
        ev << "C P ----> " << value2 << "  D a t a   S e g m e n t : " << k << "  --->  r e c e i v e d    w i t h    e r r o r!!  \n\n";
    else
        ev << "C P ----> " << value2 << "  V a l u e  3 --> " << value3 << " \n\n";
CP_Blocks[CP_counter] = value2;
CP_counter++;
```cpp
value1 = (value1 & 0xFFFFFFFF);      // value_to_send
value2 = value2 << 8;        // Shifting bits to the left
value2 = (value2 & 0xFFFFFFFF);      // value_to_send
value3 = value3 << 16;        // Shifting bits to the left
value3 = (value3 & 0xFFFFFFFF);      // value_to_send
value4 = value4 << 24;        // Shifting bits to the left
value4 = (value4 & 0xFFFFFFFF);      // value_to_send
flags = ( 0x08 );         // Flag = Report Segment
Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
if((value2>>8)>=Last_CP_to_resent)     // If the CP to send is greater than the CP sent, then the RS is not a Retransmission
    Last_CP_to_resent = value2>>8;     // This is to follow the CP sent by the Earth Station. Needed in the "IsSelfMessage"
RS_Backup_Core = Core;
msg->setKind(Core);
msg->setByteLength(1024);       // Setting 1 KB message length (1024*8=8192 bits)
bubble("Sending RS to Cassini");
send(msg, "out");
```

---

**Simulation of LPT in OMNet++**

```cpp
T I M E R   S E T U P
```

//

Page 76 of 144
RS_backup[value1] = RS_Backup_Core;

flags = ( 0x08 ); // set 'flags' to 0x80 (Internal Management Code.

// Flags = 0x08 ---> means Report Segment Timer

value2 = 0;
value3 = 0;
value4 = 0;

char timermsg[32];
sprintf( timermsg, "Timer RS-%d", value1); // assigns "Timer RS-#" to buffer 'msgname'

ev << "Generating " << timermsg << endl; // 'endl' flush the buffer

timeoutEvent_RS[value1] = new cMessage(timermsg); // We create a message 'msg' from 'msgname'

ev << "\n";

ev << " " << timeoutEvent_RS[value1]->name() << " was Setup!! \n\n";

"************************************************************************************************************

"Flags --> " << flags << "     ||    Value 4 --> " << value4 << "     ||    Value 3 --> " << value3 << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << "  

"************************************************************************************************************

Core = (flags << 27) | value4 | value3 | value2 | value1 ; // add the 4-bit flags , Value2 is zero, RA just have RS number

timeoutEvent_RS[value1]->setKind(Core);

scheduleAt(simTime()+ timeout, timeoutEvent_RS[value1]); // ev << "Core --------> " << Core << " \n\n";

if(timeoutEvent_RS[value1]->isScheduled())
>name() << " is Scheduled !! \n\n";
    else
>name() << " NOT Scheduled !! \n\n";
    RS_counter++;
    value3 = 0;
    }
    break;
    case (0x02 | 0x01): //  0  0  1  1
        3 ----> Red Data, CP, EORP, EOB
        {
            ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            //     Sending Report Segments with the error bits to resend in value3
            ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

            Red_data_segments_counter++;
            CP_counter++;
            flags = ( 0x00 ); // Reseting the Flag
            ev << " R E D   D S   R E C E I V E D   ,   C P,   E O R P,   E O B  ) \n\n";

            Core=0; // I
            set to zero Core,

            CP_Blocks[CP_counter] = value2; // Assign the value of check point to the array CP_Blocks[]

            segment_lost[DS_from_Cassini]=(0x00); // Set the segment_lost[] array to ZERO

            CP_counter++;
            value3 = 0;
            value4 = 1; // Setting value4 as 1. Last RS to send

            ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
            // Sending Report Segments with the error bits to resend in value3
            ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
value1 = First_RS;
value1 = value1 + RS_counter;

char Rep_Seg[32];
sprintf( Rep_Seg, "Report Segment-%d, (CP-%d)\n", value1, value2);
cMessage *msg = new cMessage(Rep_Seg);
ev << "Sending " << Rep_Seg << " to Cassini \n\n";
l=m;
m=m+8;

for(k=l;k<m;k++)
{
    value3 = value3 | segment_lost[k];
    if (segment_lost[k]! = 0)
        ev << "C P ----> " << value2 << "  Data Segment : " << k << " received with error!! \n\n";
    else
        ev << "C P ----> " << value2 << "  Data Segment : " << k << " OK ! \n\n";
}

ev << " Value to request retransmission ----------> " << value3 << " \n\n";

value1 = (value1 & 0xFFFFFFFF);      //
value_to_send

value2 = value2 << 8;        // Shifting
bits to the left

value2 = (value2 & 0xFFFFFFFF);      //

value3 = value3 << 16;
// Shifting
bits to the left

value3 = (value3 & 0xFFFFFFFF);      //

value4 = value4 << 24;
// Shifting
bits to the left

value4 = (value4 & 0xFFFFFFFF);      //

value_to_send
flags = (0x08); //

Flag = Report Segment

Core = (flags << 27) | value4 | value3 | value2 | value1; // add the 4-bit flags

if((value2>>8)>=Last_CP_to_resent) // If the CP to send
is greater than the CP sent, then the RS is not a Retransmission
Last_CP_to_resent = value2>>8; // This is to
follow the CP sent by the Earth Station. Needed in the "isSelfmessage"

RS_Backup_Core = Core;
msg->setKind(Core);
msg->setByteLength(1024); // Setting 1 KB message length (1024*8=8192 bits)

bubble("Sending RS to Cassini");
send(msg, "out");

////////////////////////////////////////////////////////////
//
// RS_backup[value1] = RS_Backup_Core;
//
flags = (0x08); // set 'flags' to 0x80 (Internal Management Code. Flags = 0x08 ---> means Report Segment Timer

0x08 ---> means Report Segment Timer

value2 = 0;
value3 = 0;
value4 = 0;

char timermsg[32];
sprintf(timermsg, "Timer RS-%d", value1); // assigns "Timer RS-#" to buffer 'msgname'
ev << "Generating " << timermsg << endl; // 'endl' flush the buffer

timeoutEvent_RS[value1] = new cMessage(timermsg); // We create a
message 'msg' from 'msgname'
ev << "\n";
ev << " " << timeoutEvent_RS[value1]->name() << " was Setup!! \n\n";

ev << "************************************************************************************************************
" << flags << "     ||    Value 4 --> " << value4 << "     ||    Value 3 --> " << value3 << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << " \n\n"

Core = (flags << 27) | value4 | value3 | value2 | value1; // add the 4-bit flags , Value2 is zero, RA just have RS number

timeoutEvent_RS[value1]->setKind(Core);

scheduleAt(simTime()+ timeout, timeoutEvent_RS[value1]);

ev << "Core ------> " << Core << " \n\n";

if(timeoutEvent_RS[value1]->isScheduled())
    ev << "TimeoutEvent_RS [ " << value1 << " ] ----------> " << timeoutEvent_RS[value1]->name() << " is Scheduled !! \n\n";
else
    ev << "timeoutEvent_RS [ " << value1 << " ] ----------> " << timeoutEvent_RS[value1]->name() << " NOT Scheduled !! \n\n";

RS_counter++;
value3 = 0;
break;

case 0x04:       // 0  1  0
    4 ----> Green Data, NOT EOB
    
    {  
        // G R E E N       D A T A ,       N O T       E O B
        ev << "Green Data Received , NOT EOB \n\n";
        Green_data_segments_counter++;
    }
break;
```c++
    case (0x04 | 0x02 | 0x01):
        // 0 1 1 1 1
        7 ----> Green Data, EOB
        {
            /////////// GREEN DATA, EOB
            ////////////////
            ev << "Green Data Received, EOB \n";
            Green_data_segments_counter++;
        }
        break;

    case 0x08:       // 1 0 0
        0   8 ----> Report Segment (RS)
        ev << "Report Segment (RS) \n";
        break;

    case (0x08 | 0x01):     // 1 0 0 1
        9 ----> Report Acknowledgment (RA)
        {
            /////////// REPORT ACKNOWLEDGEMENT //////////////
            // E C E I V E D AT E A R T H S T A T I O N
            ev << " / / / / / / / / / / / / / /              REPORT ACKNOWLEDGEMENT RECEIVED   AT   E A R T H  S T A T I O N   / / / / / / / / / / / / / / / / / /     
            Rep_Ack_Counter++;
            ev << "Report Acknowledgement Received ----> " << msg->name() << "..\n";
            ev << "Value1 ----> " << value1 << "..\n";
            ev << "Timer ----> " << timeoutEvent_RS[value1]->name() << "..Cancelled !!..\n";
            cancelAndDelete(timeoutEvent_RS[value1]);
            Core=0;
            ev << msg ->name() << " received!\n";
            ev << " End of Transmission, Session Closed !\n";
            bubble("End of Transmission, Session Closed !!");
            // Showing the end of the transmission
        }
        break;

    case (0x08 | 0x04):       // 1 1 0 0
        12 ----> Cancel Segment from Block Sender (CS)
        {
            //cancelAndDelete(timeoutEvent_RS[value1]);
            Core = 0;
            ev << "Cancel Segment from Block Sender (CS) \n";
```
value1 = 0;
value3 = 0;
value4 = 0;

char RLEXC[32];
sprintf(RLEXC, "Cancel Segment-Acknowledgment to Block Sender ", value3);
cMessage *msg = new cMessage(RLEXC);
ev << "Sending " << RLEXC << " from Earth Station to Cassini 

";

value1 = (value1 & 0xFFFFFFFF);      // value_to_send
value2 = value2 << 8;        // Shifting bits to
the left
value2 = (value2 & 0xFFFFFFFF);      // value_to_send
value3 = value3 << 16;        // Shifting bits to
the left
value3 = (value3 & 0xFFFFFFFF);      // value_to_send
value4 = value4 << 24;        // Shifting bits to
the left
value4 = (value4 & 0xFFFFFFFF);      // value_to_send

flags = ( 0x08 | 0x05 );       // Setting Flag as
Cancel-Acknowledgement Segment from Block Sender
Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  //  add the 4-bit flags
msg->setByteLength(1024);       // Setting 1 KB
message length (1024*8=8192 bits)

msg->setKind(Core);
ev << "Sending Cancel-Acknowledgment Segment to Block Sender (CAS) 

";
bubble("Sending Cancel-Acknowledgement Segment to Cassini");
send(msg, "out");

break;

13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)
ev << "Cancel Acknowledgment Segment to Block Sender (CAS) 

";
bubble("Cancel Acknowledgment Segment to Cassini");
send(msg, "out");

break;
case (0x08 | 0x04 | 0x02): // 1 1 1 0
14 ----> Cancel Segment from Block Receiver (CR)
ev << " Cancel Segment from Block Receiver (CR) \n";
break;

case (0x08 | 0x04 | 0x02 | 0x01): // 1 1 1 1
15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)
ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
break;

default:
    ev << " UNDEFINED!! \n";
brea;
Simulation of LPT in OMNet++

```cpp
recordScalar("CP_count_Statist", CP_count_Statist);
recordScalar("Rep_Ack.Counter", Rep_Ack.Counter);
recordScalar("RS_counter", RS_counter);

ev << "Earth Station --> Total Usage Link Time: " << t << " Seconds (" << t/60/60 << " Hours)" << endl;
Green data segments counter+CP_count_Statist+Rep_Ack.Counter)<<" Minutes " << endl;
ev << "Earth Station --> Red Data Segments Received: " << Red_data_segments_counter << endl;
ev << "Earth Station --> Green Data Segments Received: " << Green_data_segments_counter << endl;
ev << "Earth Station --> Checkpoints Received: " << CP_count_Statist << endl;
ev << "Earth Station --> Report Segments sent: " << RS_counter << endl;
ev << "Earth Station --> Report Acknowledgments Received: " << Rep_Ack.Counter << endl;

recordScalar("#Red_data_segments_counter", Red_data_segments_counter);
recordScalar("#Green_data_segments_counter", Green_data_segments_counter);
```

Transmitter_CS.cpp

```cpp
//
// This file is part of Licklider Transmission Protocol Simulation based on OMNet++.
//
// Copyright (C) 2007 Fernando Carrasco
//
// This file is distributed WITHOUT ANY WARRANTY.
//
#include "string.h"
#include "omnetpp.h"
#include "iostream"
#include "string"

using namespace std;

Jerry DOS: ~
```

Page 85 of 144
private:
    int value1, value2, value3, value4, Core, flags;
    int CheckPoint_CS;       // This is the last value of the CP (value2) before be sent
    int Red_Data_Segment_CS;     // This is the last value of the RDS (value4) before be sent it
    int Green_Data_Segment_CS;     // This is the last value of the GDS (value3) before be sent it
    int Report_Acknowledgment_CS;    // This is the last value of the RA (value1) before be sent it

Public:
    Transmitter_CS();
    virtual ~Transmitter_CS();

Protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();

Define_Module(Transmitter_CS);

///////////////////////////////////////////////////////////// Constructor and Destructor for the LTP Timer/////////////////////////////////////////////////////////////
Transmitter_CS::~Transmitter_CS()
{
}

/////////////////////////////////////////////////////////////// :: initialize!! ///////////////////////////////////////////////////////////////////////////
void Transmitter_CS::initialize()
{
    Out_Red_Data_Segment_CS_Stats.setName("Out_CS_Red_Data_Segment_CS_Stats");
    Out_Red_Data_Segment_CS_Vector.setName("Out_CS_Red_Data_Segment_CS_Vector");
    Out_Green_Data_Segment_CS_Stats.setName("Out_CS_Green_Data_Segment_CS_Stats");
    Out_Green_Data_Segment_CS_Vector.setName("Out_CS_Green_Data_Segment_CS_Vector");
    Out_CheckPoint_CS_Stats.setName("Out_CS_CheckPoint_CS_Stats");
    Out_CheckPoint_CS_Vector.setName("Out_CS_CheckPoint_CS_Vector");
    Out_Report_Acknowledgment_CS_Stats.setName("Out_CS_Report_Acknowledgment_CS_Stats");
    Out_Report_Acknowledgment_CS_Vector.setName("Out_CS_Report_Acknowledgment_CS_Vector");
}

/////////////////////////////////////////////////////////////// :: handleMessage!! ////////////////////////////////////////////////////////////////////////
void Transmitter_CS::handleMessage(cMessage *msg)
{
    unsigned int Core = msg->kind();
    ev << "Core --> " << Core << " \n\n";
    msg->senderModuleId();
    Core = Core & 0xFFFFFFFF;     // mask out 32-bits
    flags = Core & (~0x07FFFFFF);    // mask out all flags
    flags = (flags >> 27) & 0x0F;    // Shift the bits 27 bits to the right (to the Less Significant Nibble)
    and clear
    // any remaining bits in the left of the
    nibble
    value1 = Core & (~0xFFFFFFFF00); // mask out the value 1
    value2 = Core & (~0xFFFFFFFF00); // mask out the value 2
    value2 = value2 >> 8;
    value3 = Core & (~0xFFFF0000); // mask out the value 3
    value3 = value3 >> 16;
    value4 = Core & (~0xFFFF0000); // mask out the value 4
    value4 = value4 >> 24;


ev << "*********************************************************************************************************
";

ev << "T R A N S M I T T E R  C A S S I N I 
";

ev << "*********************************************************************************************************
";

ev << "Flags -- " << flags << "     ||    Value 4 -- " << value4 << "      ||    Value 3 -- " << value3 << "     ||    Value 2 -- " << value2 << "     ||    Value 1 -- " << value1 << "  
";

ev << "*********************************************************************************************************

switch ( flags )
{
    case 0x00:   // 0 ----> Red Data, NOT(CP, EORP, EOB)
        
        Red_Data_Segment_CS = value3;

        Out_Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
        Out_Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent

        break;

    case 0x01:   // 1 ----> Red Data, CP, NOT(EORP, EOB)
        
        Red_Data_Segment_CS = value3;
        CheckPoint_CS = value2;

        Out_Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
        Out_Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
        Out_CheckPoint_CS_Vector.record(CheckPoint_CS);     // Updating Statistics of the number of Checkpoints Sent
        Out_CheckPoint_CS_Stats.collect(CheckPoint_CS);     // Updating Statistics of the number of Checkpoints Sent

        break;

    case 0x02:   // 2 ----> Red Data, CP, EORP, NOT EOB
        

Simulation of LPT in OMNet++

    ////////////////////////////////////////////////// RED DATA, CP, EORP (NOT EOB)     //////////////////////////////////////////////////
ev << "RED DS IN TRANSMISSION \n\n";
Red_Data_Segment_CS = value3;
CheckPoint_CS = value2;
    /////////////////////////////////////////////////// // Updating Statistics of the number of Red
Data Segments Sent
Out_Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red
Data Segments Sent
Out_Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red
number of Checkpoints Sent
Out_CheckPoint_CS_Vector.record(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
Out_CheckPoint_CS_Stats.collect(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
    ////////////////////////////////////////////////////
}
break;

    case 0x03:  // 3 ----> Red Data, CP, EORP, EOB
    {
    ////////////////////////////////////////////////// RED DATA, CP, EORP, EOB     //////////////////////////////////////////////////
ev << "RED DS IN TRANSMISSION \n\n";
Red_Data_Segment_CS = value3;
CheckPoint_CS = value2;
    //////////////////////////////////////////////////// // Updating Statistics of the number of Red
Data Segments Sent
Out_Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red
Data Segments Sent
Out_Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red
number of Checkpoints Sent
Out_CheckPoint_CS_Vector.record(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
Out_CheckPoint_CS_Stats.collect(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
    ////////////////////////////////////////////////////
}
break;

    case 0x04:   //  4 ----> Green Data, NOT EOB
    {
    ////////////////////////////////////////////////// GREEN DATA, NOT EOB     //////////////////////////////////////////////////
ev << "GREEN DS IN TRANSMISSION \n\n";
Green_Data_Segment_CS = value3;
    //////////////////////////////////////////////////// // Updating Statistics of the number of Red
Data Segments Sent
Out_Green_Data_Segment_CS_Vector.record(Green_Data_Segment_CS);  // Updating Statistics of the number of Red
Data Segments Sent
Out_Green_Data_Segment_CS_Stats.collect(Green_Data_Segment_CS);  // Updating Statistics of the number of Red
number of Checkpoints Sent
Out_CheckPoint_CS_Vector.record(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
Out_CheckPoint_CS_Stats.collect(CheckPoint_CS);     // Updating Statistics of the
number of Checkpoints Sent
    ////////////////////////////////////////////////////
}
break;
case 0x07:   //  7 ----> Green Data, EOB

    ///////////         G R E E N       D A T A ,             E O B    /////////////
    ev << "Green Data Received, EOB
    Green_Data_Segment_CS = value3;
    //Updating Statistics of the number of Red Data Segments Sent
    Out_Green_Data_Segment_CS_STATS Vector.record(Green_Data_Segment_CS);
    //Updating Statistics of the number of Red Data Segments Sent
    Out_Green_Data_Segment_CS_Stats.collect(Green_Data_Segment_CS);

    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent

break;
case 0x09:   //  9 ----> Report Acknowledgment (RA)

    ///////////         R E P O R T     S E G M E N T    /////////////
    ev << "Report Segment (RS) \n"
    Report_Acknowledgment_CS = value1;
    //Updating Statistics of the number of Red Data Segments Sent
    Out_Report_Acknowledgment_CS_Vector.record(Report_Acknowledgment_CS);
    //Updating Statistics of the number of Red Data Segments Sent
    Out_Report_Acknowledgment_CS_Stats.collect(Report_Acknowledgment_CS);

    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent

break;
case (0x08 | 0x04):     // 12 ----> Cancel Segment from Block Sender (CS)

    ///////////         C A N C E L    S E G M E N T    F R O M     B L O C K   S E N D E R   (CS)
    ev << "Cancel Segment from Block Sender (CS) \n"
    //Red_Data_Segment_CS = value3;
    //Updating Statistics of the number of Red Data Segments Sent
    //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);
    //Updating Statistics of the number of Red Data Segments Sent
    //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);

    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent
    //Updating Statistics of the number of Red Data Segments Sent

break;
case (0x08 | 0x05):   // 13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)
Simulation of LPT in OMNet++

////////////////////////////////////////////////////////
Cancel Acknowledgment to Block Sender (CAS)
////////////////////////////////////////////////////////
ev << "Cancel Acknowledgment Segment to Block Sender (CAS) \n";
//Red_Data_Segment_CS = value3;
/////////////////////////////////////////////////////////////////////
//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
/////////////////////////////////////////////////////////////////////
break;

case (0x08 | 0x06): // 14 ----> Cancel Segment from Block Receiver (CR)
{
////////////////////////////////////////////////////////
Cancel Segment from Block Receiver (CR)
////////////////////////////////////////////////////////
ev << " Cancel Segment from Block Receiver (CR) \n";
//Red_Data_Segment_CS = value3;
/////////////////////////////////////////////////////////////////////
//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
/////////////////////////////////////////////////////////////////////
break;

case (0x08 | 0x07): // 15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)
{
////////////////////////////////////////////////////////
Cancel Acknowledgment Segment to Block Receiver (CAR)
////////////////////////////////////////////////////////
ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
//Red_Data_Segment_CS = value3;
/////////////////////////////////////////////////////////////////////
//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS); // Updating Statistics of the number of Red
Data Segments Sent
/////////////////////////////////////////////////////////////////////
break;
default:
ev << " UNDEFINED!! \n";
brea
ev << "Sending " << msg->name() <<" from Cassini\n\n";
send(msg, "out");
}

void Transmitter_CS::finish()
{
    // This function is called by OMNeT++ at the end of the simulation.
    double t=simTime();
    if (t==0) return;
}

//
// This file is part of Licklider Trasmission Protocol Simulation based on OMNeT++.
//
// Copyright (C) 2007 Fernando Carrasco
//
// This file is distributed WITHOUT ANY WARRANTY.
//
#include <string.h>
#include <omnetpp.h>
#include <iostream>
#include <string>
using namespace std;

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//
class Receiver_CS : public cSimpleModule
{
  private:
    #define ON  1
    #define OFF  0

    int counter;
    int i,j,k,CP,x,EORP,flags,w,z;
    unsigned int Core;       // Core Segment
    int ds_backup[2048];      // Keeps in memory the a copy of the Data Segments transmitted
    int CP_backup[256];       // To keep in memory the CP transmitted......
    int CP_index_backup[256]; // To keep the CP_index and use as reference in retransmissions
    int CP_index_counter;      // To count the number of CP's to be assigned inside the array.
    int number_of_DS;       // number of data segments
    int tx_request;
    int first_block;
    int last_block;
    int block;
    int block_lock;
    int value1, value2, value3;
    int value4, value_to_send;
    bool ds_error;
    int Red_block_counter;
    int Green_block_counter;
    int rotate;
    int byte_to_rotate;
    int error_bit;
    int CP_index;        // Index of between Checkpoints
    int DS_to_resend;       // After the detection of Erroneous Bits, Data Segment
    int last_DS_error_CP;      // Counter for the loop to detect the last DS to resend
    int Red_block_counter;      // Counter to detect the last DS (when we are resending DS) and
    int RDS_number;        // Assigning value1 to RS_number temporary to skip few
    int New_CP_counter;       // Checkpoint counter of the data segments that are resend
    int last_DS_with_CP;
    int last_DS_error_CP;
    int counting_error_bit;
    int Timer_CP_counter;

    // Counter to Stop the LTF simulation
    // Core Segment
    // Keeps in memory the a copy of the Data Segments transmitted
    // To keep in memory the CP transmitted......
    // To keep the CP_index and use as reference in retransmissions
    // To count the number of CP's to be assigned inside the array.
    // number of data segments
    // First Block or Checkpoint of the Session
    // Last Block to transmitt by Cassini
    // Total number of blocks
    // Counts the number of RDS arriving to form blocks
    // Counts the number of GDS arriving to form blocks
    // Rotates Bits
    // Index of between Checkpoints
    // After the detection of Erroneous Bits, Data Segment
    // Assigning value1 to RS_number temporary to skip few
    // Checkpoint counter of the data segments that are resend
    // Counter to detect the last DS (when we are resending DS) and
    // Counter for the loop to detect the last DS to resend
    // Counting error bits in the loop that was made to resend
    // Counter for the CP timer. This is for the CP_backup[] array.....

    // Byte on the Report Segment
    int CP_index;
    int DS_to_resend;

    // requested from memory for retransmission
    int RDS_number;

    // lines
    // due to errors...

    // mmmhhh let's check it out..
int RS_received_counter;     // Counter of RS received. This is for the Timers ON/OFF process.
int Total_CP_number;      // Total number of Checkpoints before start transmission
int Zero_value;        // Value zero: to replace value 3 with Zero_value
just to send the RA to Earth_Station
int CP_Array_counter;      // Counting each CP sent from cassini. This Counter is applied to
the for the CP array
//int CP_Array[256];
int System_Clock;
int RDS_iniCounter;
int GDS_iniCounter;
int Green_Loop;
int Green_Section_Lock;
// at the beginning of the loop
int CP_Counter_to_ReTx_Last_DS;
int CP_Backup_Core;
int Core_ReTx;
int Core_temp;
int Session_Closed_CP[256];
int RS_Resend_Counter[128];
int Error_Counter_RS;
int CX_value2;
// processes
int Report_Segment_CS;
Cassini. To be followed by the 'Watch' function
int CheckPoint_CS;
// it by Cassini. To be followed by the 'Watch' function
int RDS_Segment_CS;
Cassini. To be followed by the 'Watch' function
int Reserved_Value_CS;
sent it by Cassini. To be followed by the 'Watch' function
int Green_Segment_CS;
Cassini. To be followed by the 'Watch' function
int Report_Acknowledgment_CS;
To be followed by the 'Watch' function
int Last_RS_Received;
// Counts the last Report Segment received from Earth Station
double timeout;
int msgLength;        // Length of Messages
// Setting Histogram and Vectors for statistical purposes
//---------
cLongHistogram Red_Data_Segment_CS_Stats;
cOutVector Red_Data_Segment_CS_Vector;
//cLongHistogram Green_Data_Segment_CS_Stats;
//cOutVector Green_Data_Segment_CS_Vector;
cLongHistogram Report_Segment_CS_Stats;
cOutVector Report_Segment_CS_Vector;
cLongHistogram CheckPoint_CS_Stats;
cOutVector CheckPoint_CS_Vector;

/////////////////////////////////////////////////////////////////////////////////////////////////

//double timeout;        // timeout

cMessage *event;       // pointer to the event object which we'll use for timing

cMessage *timeoutEvent_CP[256];
cMessage *timeoutEvent_Cancel[128];

public:
    int Red_data_segments;
    int Red_data_segments_counter;
    int Green_data_segments;
    int Green_data_segments_counter;
    int CP_initial;        // First Random CheckPoint to initiate the
    // First Random CheckPoint to initiate the
transmission. From Omnetpp.ini
    //int msgLength;
    Receiver_CS();
    virtual ~Receiver_CS();

protected:
    virtual void initialize();
    virtual void handleMessage(cMessage *msg);
    virtual void finish();      // Called by OMNeT at the end of the simulation

Define_Module(Receiver_CS);

//////////////////////////////////////////////  Constructor and Destructor for the LTP Timer //////////////////////////////////

Receiver_CS::Receiver_CS()
{
    timeoutEvent_CP[256] = NULL;
    timeoutEvent_Cancel[128] = NULL;
    event = NULL;
}

Receiver_CS::~Receiver_CS()
{
    cancelAndDelete(timeoutEvent_CP[256]);
    cancelAndDelete(timeoutEvent_Cancel[128]);
    cancelAndDelete(event);
}

//////////////////////////////////////////////////// //:: initialize!! ///////////////////////////////////////////////////////

void Receiver_CS::initialize()
{
    i,j,k,x,w,z=0;
    Core = 0;
    Core_ReTx = 0;
    Core_temp = 0;
    CP_Backup_Core = 0;
    counter = 0;
    Red_data_segments_counter=0;
    Green_data_segments_counter=0;
    last_block = 0;
    block =0;

    Red_block_counter =0;
    // Reseting – This is a counter of RDS arriving to form blocks –
    counts RDS !
    Green_block_counter =0;
    CP_Array_counter = 0;
    System_Clock=0;
    Green_Loop = OFF;
    Green_Section_Lock =OFF;
    Error_Counter_RS =0;
    CX_value2 = 0;

    int RS_Resend_Counter[128]={0};

    CP_index_counter = 0;
    CP_Counter_to_ReTx_Last_DS = 0;

    RDS_ini_Counter = 0;
    GDS_ini_Counter = 0;

    Last_RS_Received = 0;

    block_lock = ON;

    flags=0;
    value1=0;
    value2=0;
    value3=0;
value4 = 0;

int Red_data_segments = par("Red_data_segments");
int Green_data_segments = par("Green_data_segments");
int CP_initial = par("CP_initial");
double timeout = par("timeout");
//int msgLength = par("msgLength");
//msgLength = &par("msgLength");
//int CP = 1 + intrand(50); // Result of Check Point Random Number in the range 0..1999
//WATCH(RDS_ini_Counter);
//WATCH(GDS_ini_Counter);
//WATCH(Timer_CP_counter);
//WATCH(Report_Segment_CS);
//WATCH(CheckPoint_CS);
//WATCH(Red_Data_Segment_CS);
//WATCH(Green_Data_Segment_CS);
//WATCH(Reserved_Value_CS);
//WATCH(timeoutEvent_CP[]);
//WATCH(flags);
//WATCH(Core);

//WATCH(msg);

Report_Segment_CS_Stats.setName("In_CS_Report_Segment_CS_Stats");
Report_Segment_CS_Vector.setName("In_CS_Report_Segment_CS_Vector");

CheckPoint_CS_Stats.setName("In_CS_CheckPoint_CS_Stats");
CheckPoint_CS_Vector.setName("In_CS_CheckPoint_CS_Vector");

//CP_initial = 79;
first_block = CP_initial;
value2 = CP_initial;

k = 0;
x = 0;

event = new cMessage("event");

//timeout = 9324.0; // Maximum timeout. RTT + 5% to know if we
lost the CP. (74min * 2 * 60 = 8880+444 = 9324 seconds)
scheduleAt(0.0, event);
void Receiver_CS::handleMessage(cMessage *msg)
{
  int Red_data_segments = par("Red_data_segments");
  int Green_data_segments = par("Green_data_segments");
  double timeout = par("timeout"); // First Random CheckPoint to
  int CP_initial = par("CP_initial");
  initiate the transmission. From Omnetpp.ini //int msgLength = par("msgLength");

  //ev << "Transmitted by Module ID: " << ES_id << " .... \n"; // Module ID number from Transmitter
  //ev << "This Module ID: " << module_id << " .... \n\n"; // Module ID number of this module

  int msg_length = msg->byteLength();
  ev << " Message Length ----> " << msg_length << "..\n\n";

  if(RDS_ini_Counter==0)
  {
    value1 = (value1 & 0xFFFFFFFF); // value_to_send
    value2 = value2 << 8; // Shifting bits to left
    value2 = (value2 & 0xFFFFFFFF); // value_to_send
    value3 = value3 << 16;
    value3 = (value3 & 0xFFFFFFFF); // value_to_send
    value4 = value4 << 24;
    value4 = (value4 & 0xFFFFFFFF); // value_to_send
    i=0;
    bool ds_error=0;
    block = Red_data_segments / 8; // to know the number of blocks
    last_block = Red_data_segments % 8; // To know the reminder of the division

    if (last_block > 0) // if reminder is higher than zero, add 1
    {
      block = block +1;
    }

    Total_CP_number = block; // Number of CP before starting first transmission
  }
}
if ( msg == event )
{
    if(Green_Section_Lock == ON)      // Coming from the Last Red Data Segment sent. This is to
activate the Green Section
        Green_Loop = ON;
    if(RDS_ini_Counter<Red_data_segments)
    {
        Red_block_counter++;       // Counting the number of RDS to form blocks of 8
    RDS
        if(RDS_ini_Counter==Red_data_segments-1)  // if Last data segment, add send it as a Red Data, CP, EORP, EOB
        {
            flags = ( 0x02 | 0x01 );     // 3 ----> Red Data, CP, EORP, EOB
            ev << "Last Data segment 

";
            Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
            CP_Backup_Core = Core;
            value3 = value3 >> 16;       // Keeping a copy of the DS before be sent
            ds_backup[value3]= Core;
            value3 = value3 << 16;
            CP_index_backup[value2]= CP_index_counter;
            value2 = value2 >> 8;
            char msgname[32];
            sprintf( msgname, " Last Red Data Segment-%d, CP-%d, & EORP",(value3 >> 16),(value2 >> 8));
            //assigns "Data Segment-#" to buffer 'msgname'
            ev << "Sending " << msgname << " .... \n"
;
            cMessage *msg = new cMessage(msgname);  //We create a message 'msg' from 'msgname'
            //msg->setLength( (long) *msgLength );  //Sets the length of the message
//msg->setTimestamp(--x); //Sets the time stamp to the current Data Segment.

msg->setTimestamp(System_Clock++); // Setting the time stamp for this DS
msg->setByteLength(1024); // Setting 1 KB message length (1024*8=8192 bits)
msg->senderModuleId(); // Sets the module ID of current module
msg->arrivalModuleId(); // Sets the module ID of the module to send
msg->setKind(Core);
msg->setBitError(ds_error);

ev << "*********************************************************************************************************
---------- Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "     ||    Value 3 --> " << (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << "  
*********************************************************************************************************

ev << "Last Red Data Segment with CP and EOB Flags 

send(msg, "out");

***************************************************** T I M E R   S E T U P *****************************************************

value2 = value2 >> 8;
CP_backup[value2] = CP_Backup_Core;
value2 = value2 << 8;
flags = ( 0x08 ); // set 'flags' to 0x80 (Internal Management Code. Flags = 0x08 ---> means

CheckPoint Timer

value1 = 0;
value4 = 0;

timeMsg[32];
sprintf( timermsg, "Timer CP-%d",value2>>8); // assigns "Timer CP-#" to buffer 'msgname'
ev << "Generating " << timermsg << endl; // 'endl' flush the buffer

timeoutEvent_CP[value2>>8]= new cMessage(timermsg);

value1 = (value1 & 0xFFFFFFFF);  // value_to_send
value4 = value4 << 24; // Shifting bits to left
value4 = (value4 & 0xFFFFFFFF); // value_to_send

"*********************************************************************************************************\n' ev << "Flags --> " << flags << " || Value 4 --> " << (value4>>24) << " || Value 3 --> " << (value3>>16) << " || Value 2 --> " << (value2>>8) << " || Value 1 --> " << value1 << "\n"; ev << "*********************************************************************************************************\n
Core = (flags << 27) | value4 | value3 | value2 | value1; // add the 4-bit flags, Value2 is zero, RA just have RS number

timeoutEvent_CP[value2>>8]->setKind(Core);
scheduleAt(simTime()+ timeout, timeoutEvent_CP[value2>>8]);

"Event Scheduled ------> " << timeoutEvent_CP[value2>>8] << "\n\n"; ev << "Core ------> " << Core << "\n\n";

\n\n\nEND OF TIMER SETUP

Green_Section_Lock = ON;
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;

value2 = value2 >> 8; // Incrementing Value 2 which is the CP
value2++; // in the Data Segment

value2 = value2 << 8; //

Timer_CP_counter++;

CP_index_counter++;

} else {

}
if (Red_block_counter%8==0 && RDS_ini_Counter!=Red_data_segments-1)
{
    ev << "end of block \\
    \n"
    flags = ( 0x01 );
    // 1 ----> Red Data, CP, NOT (EORP, EOB)
    Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
    CP_Backup_Core = Core;
    value3 = value3 >> 16;
    ds_backup[value3]= Core;
    value3 = value3 << 16;
    value2 = value2 >> 8;
    CP_index_backup[value2]= CP_index_counter;
    value2 = value2 << 8;
    char msgname[32];
    sprintf( msgname, " Red Data Segment-%d, CP-%d ",
             (value3 >> 16), (value2 >> 8));
    cMessage *msg = new cMessage(msgname); //We create a message 'msg'
    //msg->setLength( (long) *msgLength ); //Sets the length of the
    //msg->setTimestamp(--x);        //Sets the time stamp
    msg->setTimestamp(System_Clock++); // Setting the time stamp for this DS
    msg->setByteLength(1024);       // Setting 1 KB message
    msg->senderModuleId();        //Sets the module ID of
    current module
    msg->arrivalModuleId();       //Sets the module ID
    of the module to send
    msg->setKind(Core);
    msg->setBitError(ds_error);
    //bubble("End of Block, Check Point Request...");
ev <<
"*********************************************************************************************************
Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "     ||    Value 3 --> " << (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << "\n"
"; send(msg, "out");

CheckedPoint Timer Setup

value2 = value2 >> 8;

CP_backup[value2] = CP_Backup_Core;
value2 = value2 << 8;

flags = (0x08); // set 'flags' to 0x80 (Internal Management Code. Flags = 0x08 -->

> means CheckPoint Timer

value1 = 0;
value4 = 0;

char timermsg[32];
sprintf(timermsg, "Timer CP-\%d", value2>>8);  

\'msgname' ev << "Generating " << timermsg << endl;  

\'endl' flush the buffer
timeoutEvent_CP[value2>>8] = new cMessage(timermsg);

value1 = (value1 & 0xFFFFFFFF);   // value_to_send
value4 = value4 << 24;     // Shifting bits to left
value4 = (value4 & 0xFFFFFFFF);   // value_to_send
value1 = (value1 & 0xFFFFFFFF);   // value_to_send

ev <<
"*********************************************************************************************************
Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "     ||    Value 3 --> " << (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << "\n"
"; Core = (flags << 27) | value4 | value3 | value2 | value1;  

// add the 4-bit flags to the core
timeoutEvent_CP[value2>>8]->setKind(Core);
scheduleAt(simTime()+ timeout, timeoutEvent_CP[value2>>8]);
ev << "Event Scheduled -------> " << timeoutEvent_CP[value2>>8] << " \n\n"

ev << "Core -------> " << Core << " \n\n"

//
//
//

///////////////////////////////////    E N D    O F    T I M E R    S E T U P

block_lock = ON;
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
value2 = value2 >> 8;
value2++;
value2 = value2 << 8;

Timer_CP_counter++;
CP_index_counter++;
}
else {

flags = ( 0x00 );

// 0 ------ Red Data, NOT(CP, EORP, EOB)
Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags

value3 = value3 >> 16;
ds_backup[value3]= Core;
value3 = value3 << 16;

char msgname[32];
sprintf( msgname, " Red Data Segment-%d", (value3 >> 16));  // assigns "Red Data Segment-#" to
buffer 'msgname'

ev << "\n";
ev << "Sending " << msgname << " .... \n\n";

cMessage *msg = new cMessage(msgname);       // We create a
message 'msg' from 'msgname'

//msg->setLength( (long) *msgLength );       // Sets the
length of the message

//msg->setTimestamp(-x);          // Sets
the time stamp to the current Data Segment.

msg->setTimestamp(System_Clock++);        // Setting the
time stamp for this DS

msg->setByteLength(1024);          // Setting 1 KB message length (1024*8=8192 bits)

msg->setKind(Core);
msg->setBitError(ds_error);

ev <<

************************************************************************************************************

ev << "Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "      ||    Value 3 --> "<<
(value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "      ||    Value 1 --> "<< value1 << " \n";

************************************************************************************************************

send(msg, "out");

value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}

scheduleAt(simTime()+0.0, event);
RDS_ini_Counter++;
}
// brace of "if RDS_ini_Counter"

///////////////////////////////////////////////////////////////////////////////////////////

// END OF RED DATA SEGMENT LOOP
///////////////////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////////////////

Page 105 of 144
if(GDS_ini_Counter<Green_data_segments && Green_Loop == ON )
{
    Green_block_counter++;
    value1 = 1;            // Setting the "Green" color to Green Data segments
    if(GDS_ini_Counter==Green_data_segments-1)    // if Last green data segment, add send it as a
        // if Last green data segment, add send it as a Green Data, EOB
    {
        flags = ( 0x04 | 0x02 | 0x01 );      //  7 ----> Green Data, EOB
        Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
        char msgname[32];
        sprintf( msgname, " Last Green Data Segment-%d & EOB",(value3 >> 16)); //assigns "Last Green Data Segment-#"
        to buffer 'msgname'
        ev << "\n";
        ev << "Sending " << msgname << " .... \n\n";
        cMessage *msg = new cMessage(msgname);    //We create a message 'msg' from 'msgname'
        //msg->setLength( (long) *msgLength );    //Sets the length of the message
        //msg->setTimestamp(--x);       //Sets the time stamp to the current
        Data Segment.
        msg->setTimestamp(System_Clock++);     // Setting the time stamp for this DS
        msg->setByteLength(1024);       // Setting 1 KB message length
        (1024*8=8192 bits)
        msg->senderModuleId();            //Sets the module ID of current
        msg->arrivalModuleId();         //Sets the module ID of the
        module to send
        msg->setKind(Core);
        msg->setBitError(ds_error);
ev << "Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "     ||    Value 3 --> " << (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "     ||    Value 1 --> " << value1 << " \n"
;
ev << "Last Green Data Segment with EOB Flag \n"
;
send(msg, "out"); // Sending Green Data
Message. Green Data Does not need backup!!
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
Green_Section_Lock = OFF;
Green_Loop = OFF;
} else {

if (Green_block_counter%8==0 && GDS_ini_Counter!=Green_data_segments-1) {

ev << "end of Green Data Block \n\n"
;
flags = ( 0x04 | 0x02 | 0x01 );
//  7 ----> Green Data, EOB

Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags
char msname[32];
//assigns "Data Segment-#" to buffer 'msname'
sprintf( msname, " Green Data Segment-%d & EOB ", (value3 >> 16));
ev << "\n";

ev << "Sending " << msname << " .... \n\n";
cMessage *msg = new cMessage(msname); //We create a message 'msg' from
'msname'
//msg->setLength( (long) *msgLength ); //Sets the length of the message

}
 Simulation of LPT in OMNet++

 current Data Segment.

 //msg->setTimestamp(--x);             //Sets the time stamp to the
 msg->setTimestamp(System_Clock++);   // Setting the time stamp for this DS
 msg->setByteLength(1024);           // Setting 1 KB message

 length (1024*8-8192 bits)

 msg->senderModuleId();               //Sets the module ID of
 msg->arrivalModuleId();              // the module to send
 of the module to send

 msg->setKind(Core);
 msg->setBitError(ds_error);

 ev <<
 "************************************************************************************************************
 ev << "Flags --> " << flags << " || Value 4 --> " << (value4>>24) << " || Value 3 --> " <<
 (value3>>16) << " || Value 2 --> " << (value2>>8) << " || Value 1 --> " << value1 << " \n"
 ev <<
 "************************************************************************************************************

 /*
 /////////////////////////////////////////////////////// Values to be followed by the 'Watch' function
 ///////////////////////////////////////////////////////

 //Report_Segment_CS = value1;
 CheckPoint_CS = value2>>8;
 Green_Data_Segment_CS = value3>>16;
 //Reserved_Value_CS = value4>>24;

 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////
 Green_Data_Segment_CS_Vector.record(Green_Data_Segment_CS);   // Updating Statistics of the number
 of Green Data Segments Sent
 Green_Data_Segment_CS_Stats.collect(Green_Data_Segment_CS);    // Updating Statistics of the number
 of Green Data Segments Sent
 CheckPoint_CS_Vector.record(CheckPoint_CS);                    // Updating Statistics
 of the number of CheckPoint Sent
 CheckPoint_CS_Stats.collect(CheckPoint_CS);                    // Updating Statistics
 of the number of CheckPoint Sent

 "*/

 send(msg, "out");
 scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;

}  else
{
    flags = ( 0x04 );
    // 4 ----> Green Data, NOT EOB

    Core = (flags << 27 )| value4 | value3 | value2 | value1 ;  // add the 4-bit flags

    char msgname[32];
sprintf( msgname, " Green Data Segment-%d", (value3 >> 16));  //assigns "Green Data
Segment-#" to buffer 'msgname'
    ev << "Generating " << msgname << endl;
    //'endl' flush the buffer
    ev << "\n";
    cMessage *msg = new cMessage(msgname);  //We create a message 'msg' from
    //msg->setLength( (long) *msgLength );  //Sets the length of the message
    //msg->setTimestamp(--x);       //Sets the time stamp to the
    current Data Segment.
    msg->setTimestamp(System_Clock++);
    msg->setByteLength(1024);       // Setting the time stamp for this DS
    msg->setKind(Core);
    msg->setBitError(ds_error);
    ev << "************************************************************************************************************
    Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "      ||    Value 3 --> " <<
    (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "      ||    Value 1 --> " << value1 << "  
    ************************************************************************************************************
    ";

    //Report_Segment_CS = value1;
    CheckPoint_CS = value2>>8;
    Green_Data_Segment_CS = value3>>16;
    //Reserved_Value_CS = value4>>24;
Green_Data_Segment_CS_Vector.record(Green_Data_Segment_CS);  // Updating Statistics of the number of Green Data Segments Sent
Green_Data_Segment_CS_Stats.collect(Green_Data_Segment_CS);  // Updating Statistics of the number of Green Data Segments Sent
CheckPoint_CS_Vector.record(CheckPoint_CS);      // Updating Statistics of the number of CheckPoint Sent
CheckPoint_CS_Stats.collect(CheckPoint_CS);      // Updating Statistics of the number of CheckPoint Sent

send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;
}

Send(msg, "out");
scheduleAt(simTime()+0.0, event);
value3 = value3 >> 16;
value3++;
value3 = value3 << 16;
}
}
GDS_ini_Counter++;

else
{
  unsigned int Core = msg->kind();
  unsigned int Core_Timer = msg->kind();
  int ES_id = msg->senderModuleId();
  int module_id = msg->arrivalModuleId();
}
int DS_to_resend;

ev << "Core -----------> " << Core << " .... \n";

ev << "Transmitted by Module ID: " << ES_id << " .... \n"; // Module ID number from Transmitter

ev << "This Module ID: " << module_id << " .... \n"; // Module ID number of this module

Core = Core & 0xFFFFFFFF; // mask out 32-bits
flags = Core & (~0x07FFFFFF); // mask out all flags
flags = (flags >> 27) & 0x0F; // Shift the bits 27 bits to the right (to the Less Significant Nibble) and clear

value1 =  Core & (~0xFFFFFF00); // mask out the value 1
value2 =  Core & (~0xFFFF00FF); // mask out the value 2
value2 = value2 >> 8;
value3 =  Core & (~0xFF00FFFF); // mask out the value 3
value3 = value3 >> 16;
value4 =  Core & (~0xF8FFFFFF); // mask out the value 4
value4 = value4 >> 24;

RS_number = value1; // Assigning the value1 to RS_number to jump directly to the

if (value4 == 1)
    Last_RS_Received = 1;
else
    Last_RS_Received = 0;

if (msg->isSelfMessage()) // We will receive our own messages (Timeout Messages) in this section.
{
    ev << "SELF - MESSAGE TRIGGERED @ CASSINI \n\n";
    ev << "Receiving ----> " << msg->name() << "..\n";

    Core_Timer = Core_Timer & 0xFFFFFFFF; // mask out 32-bits
flags = Core_Timer & (~0x07FFFFFF);      // mask out all flags
flags = (flags >> 27) & 0x0F;         // Shift the bits 27 bits to the right (to the Less
Significant Nibble) and clear

left of the nibble
value1 = Core_Timer & (~0xFFFFFFFF00);  // mask out the value 1
value2 = Core_Timer & (~0xFFFF00FF);    // mask out the value 2
value2 = value2 >> 8;                   // mask out the value 3
value3 = Core_Timer & (~0xFF00FFFF);    // mask out the value 3
value4 = Core_Timer & (~0xF8FFFFFF);    // mask out the value 3
value4 = value4 >> 24;

////////////////////////////////////////////~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~///

Internal Management Code - Designed to identify the type of Timer arriving

Checkpoint - Timer_CP Self-Message ----> Flags = 0x08

Cancel - Timer_CX Self-Message ----> Flags = 0x04

Schedule - Timer_SC Self-Message ----> Flags = 0x01

////////////////////////////////////////////~/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~///

switch ( flags ) {
    case 0x08:   // Self-Message for the Checkpoint CP Timer
        {
            for(w=CP_initial;w<(CP_initial+Timer_CP_counter);w++)
            {
                char msgname[32];
                sprintf( msgname, "Timer CP-%d",w);     //assigns "Timer CP-#" to
                buffer 'msgname'
                if (strcmp(msgname, msg->name()) == 0)
                {
                    Core_temp = CP_backup[value2];     // This is the
                    original Core and is use to for setKind(Core_temp)
Simulation of LPT in OMNet++

Core_ReTx = CP_backup[value2];  // This is a Core to be shred...

Core_ReTx = Core_ReTx & 0xFFFFFFFF;  // mask out 32-bits
flags = Core_ReTx & (~0x07FFFFFF);  // mask out all flags
flags = (flags >> 27) & 0x0F;      // Shift the bits 27 bits to the right (to the Less Significant Nibble) and clear

// any remaining bits in the left of the nibble
value1 = Core_ReTx & (~0xFFFFFFFF00);  // mask out the value 1
value2 = Core_ReTx & (~0xFFFFFFFF0FF); // mask out the value 2
value2 = value2 >> 8;
value3 = Core_ReTx & (~0xFFFFFFFFFF); // mask out the value 3
value3 = value3 >> 16;
value4 = Core_ReTx & (~0xFFFFFFFFFFFF); // mask out the value 4
value4 = value4 >> 24;

char msgname[32];
sprintf(msgname, "Red Data Segment-%d, CP-%d", value3, value2);

//assigns "Red Data Segment-#" to buffer 'msgname'

msgname[32];
sprintf(msgname, " Red Data Segment-%d, CP-%d", value3, value2);

// assigns "Red Data Segment=#" to buffer 'msgname'

// We create a message 'msg' from 'msgname'

ev << "Generating " << msgname << endl;  // 'endl' flush the buffer

from 'msgname'

ev << "\n";
cMessage *msg = new cMessage(msgname);  // We create a message 'msg'

// We create a message 'msg'

msg->setTimestamp(System_Clock++);  // Setting the time stamp for this DS

msg->setByteLength(1024);  // Setting 1 KB message length (1024*8=8192 bits)

msg->senderModuleId();  // Sets the module ID of current module

msg->arrivalModuleId();  // Sets the module ID of the module to send

msg->setKind(Core_temp);
msg->setBitError(ds_error);

************************************************************************************************************

Flags --> 0x0F     ||    Value 4 --> 0xFFFFFFFFFFFF     ||    Value 3 --> 0xFFFFFFFFFF     ||    Value 2 --> 0xFFFFFFFF0FF     ||    Value 1 --> 0xFFFFFFFF0000

************************************************************************************************************

**"*******************************************************************************\n"**"*******************************************************************************\n"
ev << " Red Data Segment with CP and EOB Flags \n";
bubble("Retransmitting Lost CheckPoint!"); send(msg, "out");

 Tic Mer S E T U P

Flags = 0x08 ---> means CheckPoint Timer
value1 = 0;
value3 = 0;
value4 = 0;

char timermsg[32];
sprintf( timermsg, "Timer CP-\d",value2); // assigns
"Timer CP-#" to buffer 'msgname'
flush the buffer

value2 = value2 << 8;
// Shifting 8 bits to left
value2 = (value2 & 0xFFFFFFFF);

Core = (flags << 27) | value4 | value3 | value2 | value1;  // add the 4-bit flags , Value2 is zero, RA just have RS number

timeoutEvent_CP[value2>>8]->setKind(Core);
scheduleAt(simTime()+ timeout, timeoutEvent_CP[value2>>8]);
Timer_CP_counter++;  
// Timer_CP_counter - This counts every CP (value2) generated by Casinni

ev << "Core -------> " << Core << " \n\n";  
//
//
//
///////////////////////////////////   E N D   O F    T I M E R   S E T U P
////////////////////////////////////////////////////

break;
case 0x04:   // Self-Messages for the Cancel CX Timer
{  
  ev << " Cancelling  CX Timer ---> timeoutEvent_Cancel[ " << value2 << " ] \n\n";  
  //cancelAndDelete(timeoutEvent_Cancel[value2]);
  
  break;
}
default:
  
  ev << " S E L F - M E S S A G E U N D E F I N E D !! \n";
  break;
} // Switch Brace

delete msg;  // Delete the isSelfmessage after being received. Helps to avoid error messages in the command
promt

else
{
  ev << "Received: " << msg->name() << "\n";
  ev << "CP_Backup_Core --> "]<< value2 << " \n\n";
switch (flags) {
   case 0x08:   //  8 ----> Report Segment (RS)
      {
         ///////////////////////////////////////////////////////////////////////
         REPORT SEGMENT
         ///////////////////////////////////////////////////////////////////////
         ev << "REPORT SEGMENT RECEIVED AND REGISTERED \n\n";
         Report_Segment_CS = value1;
         CheckPoint_CS = value2;
         Report_Segment_CS_Vector.record(Report_Segment_CS);  // Updating Statistics of the number of Report Segment Sent
         Report_Segment_CS_Stats.collect(Report_Segment_CS);  // Updating Statistics of the number of Report Segment Sent
         CheckPoint_CS_Vector.record(CheckPoint_CS);    // Updating Statistics of the number of CheckPoints Sent
         CheckPoint_CS_Stats.collect(CheckPoint_CS);    // Updating Statistics of the number of CheckPoints Sent
         ///////////////////////////////////////////////////////////////////////
      }
   break;
   case (0x08 | 0x04):     // 12 ----> Cancel Segment from Block Sender (CS)
   {
      ///////////////////////////////////////////////////////////////////////
      CANCEL SEGMENT FROM BLOCK SENDER (CS)
      ///////////////////////////////////////////////////////////////////////
      ev << "Cancel Segment from Block Sender (CS) \n";
      //Red_Data_Segment_CS = value3;
      //Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
      //Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
      }
//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

} break; 
case (0x08 | 0x05): // 13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)
{
////////////////////////////////////////////
////////////////////////////////////////////
ev << "Cancel Acknowledgment Segment to Block Sender (CAS) \n";
//Red_Data_Segment_CS = value3;

//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

} break; 
case (0x08 | 0x06): // 14 ----> Cancel Segment from Block Receiver (CR)
{
////////////////////////////////////////////
////////////////////////////////////////////
ev << " Cancel Segment from Block Receiver (CR) \n";
//Red_Data_Segment_CS = value3;

//Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

//Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number
of Red Data Segments Sent

} break; 
case (0x08 | 0x07): // 15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)
{
////////////////////////////////////////////
////////////////////////////////////////////
ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
//Red_Data_Segment_CS = value3;
//Red_Data_Segment_CS<Vector>.record(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent
//Red_Data_Segment_CS Stats.collect(Red_Data_Segment_CS);  // Updating Statistics of the number of Red Data Segments Sent

switch ( flags )
{
    case 0x00:       // 0  0 0
        0   0 ----> Red Data, NOT(CP, EORP, EOB)
        ev << "Red Data Segment Received, NOT(CP, EORP, EOB) " Core << " " flags <<"\n";
        break;
    case 0x01:       // 0  0 0
        1   1 ----> Red Data, CP, NOT(EORP, EOB)
        ev << "Red Data, CP, NOT(EORP, EOB) \n";
        break;
    case 0x02:       // 0  0 1
        2   2 ----> Red Data, CP, EORP, NOT EOB
        ev << "Red Data, CP, EORP, NOT EOB \n";
break;

case (0x02 | 0x01):
     //  0  0  1  1
    3 ----> Red Data, CP, EORP, EOB
    {
        ev << "Red Data Segment Received, CP, EORP, EOB \n";
        char Rep_Seg[32];
        //sprintf( Rep_Seg, "Report Segment=%d",RS); // RS random number... just testing
        cMessage *msg = new cMessage(Rep_Seg);
        ev << "Sending " << Rep_Seg << " from Cassini to Earth Station \n";
        msg->setKind(Core);
        //ev << "Report Segment " << Rep_Seg <<" from Earth Station\n";
        bubble("Sending RS to Earth Station");
        send(msg, "out");
    }
break;

case 0x04:       //  0  1  0
     0   4 ----> Green Data, NOT EOB
        ev << "Green Data, NOT EOB \n";
break;

case (0x04 | 0x02 | 0x01):   //  0  1  1  1
    7 ----> Green Data, EOB
        ev << "Green Data, EOB \n";
break;

case 0x08:       //  1  0  0
     0   8 ----> Report Segment (RS)
    {
        /////////////////////////////////////////////////// REPORT SEGMENT
            ev << " / / / / / / / / / / / / REPORT SEGMENT RECEIVED @ CASSINI / / / / / / / / / / / / \n\n";
        Core=0;
        CX_value2 = value2;
    }
ev << "Value2 ----> " << value2 << ".\n"

This section is to verify if I already received the report segment
It cancels the timer of the first RS received and starts counting the RA sent by
Cassini

This section is to verify if I already received the report segment
It cancels the timer of the first RS received and starts counting the RA sent by
Cassini

if (Session_Closed_CP[value2]==value2) // if Session_Closed_CP[] is equal to
value2 in the message....
{
    ev << " " << msg->name() << " was previously received and their session was closed !!
Sending RA again or this !! !! ..\n"
Error.Counter_RS = RS_Resend.Counter[value2]; // Get the RS error counting for the
specific value2
if (Error.Counter_RS>2) // If the RS received
has been received more than 3 times....
{
    ev << "MAXIMUM NUMBER OF RETRANSMISSION HAS
BEEN REACHED!! THE EVENT WILL BE CLOSED! \n\n";

Sending the Cancel Segment from Block Sender

Sending the Cancel Segment from Block Sender

Sending the Cancel Segment from Block Sender

flags = ( 0x08 | 0x04 ); // set 'flags' to 12 for Cancel
Segment from Block Sender
```c
value2 = 0;
value3 = 2;      //  Value3 =
value4 = 0;

delete msg;

char RLEXC[32];
sprintf( RLEXC, "Cancel Segment from Block Sender (Code-%d)",value3);
cMessage *msg = new cMessage(RLEXC);
ev << "Sending " << RLEXC << " from Cassini to Earth Station \n\n";

value_to_send
value2 = value2 << 8;        //  Shifting 8 bits to left
value2 = (value2 & 0xFFFFFFFF);      //  value_to_send

value3 = value3 << 16;
value3 = (value3 & 0xFFFFFFFF);      //  value_to_send

value4 = value4 << 24;
value4 = (value4 & 0xFFFFFFFF);      //  value_to_send

Core = (flags << 27) | value4 | value3 | value2 | value1 ;  //  add the 4-bit flags , Value2 is zero, RA just have RS number
msg->setKind(Core);
bubble("Sending Cancel Segment to Earth Station");
send(msg, "out");
ev << "Core -------> " << Core << " \n\n";
```

Reason code -----> "Retransmission limit exceeded"

---

Page 121 of 144
flags = ( 0x04 ); // set 'flags' to 0x04 (Internal Management Code. Flags = 0x04 ---> means 'Cancel Timer' )
value1 = 0;
value2 = CX_value2;
value3 = 0; // Value3 = 0
value4 = 0;
char timer_Cancel[32];
sprintf( timer_Cancel, "Timer CX-%d",value2);
// assigns "Timer CX-%d" to buffer 'msgname'
'endl' flush the buffer

value_to_send
value2 = value2 << 8; // Shifting bits to left
value2 = (value2 & 0xFFFFFFFF); // value_to_send

value1 = (value1 & 0xFFFFFFFF); // value_to_send
value2 = value2 << 8; // value_to_send
value4 = 0;

// T I M E R S
TIMERS

ET UP

flags = ( 0x04 ); // set 'flags' to 0x04 (Internal Management Code. Flags = 0x04 ---> means 'Cancel Timer' )
value1 = 0;
value2 = CX_value2;
value3 = 0; // Value3 = 0
value4 = 0;
char timer_Cancel[32];
sprintf( timer_Cancel, "Timer CX-%d",value2);
// assigns "Timer CX-%d" to buffer 'msgname'
'endl' flush the buffer

value_to_send
value2 = value2 << 8; // Shifting bits to left
value2 = (value2 & 0xFFFFFFFF); // value_to_send

value1 = (value1 & 0xFFFFFFFF); // value_to_send
value2 = value2 << 8; // value_to_send
value4 = 0;
value3 = value3 << 16;  // Shifting bits to left
value3 = (value3 & 0xFFFFFFFF); // value_to_send
value4 = value4 << 24;  // Shifting bits to left
value4 = (value4 & 0xFFFFFFFF); // value_to_send

ev << "************************************************************************************************************
|
|  Value 3 --> "<< (value3>>16) << "  ||  Value 2 --> "<< (value2>>8) << "  ||  Value 1 --> "<< value1 << "  \n"
|
|  "---------------------------------------------------------------\n
Core = (flags << 27) | value4 | value3 | value2 | value1;  // add the 4-bit flags, Value2 is zero, RA just have RS number
timeoutEvent_Cancel[value2>>8]->setKind(Core);
scheduleAt(simTime()+ timeout, timeoutEvent_Cancel[value2>>8]);

ev << "Core ------> " << Core << " \n\n";

break;  // getting out of this case
else  // else, count the RS received
{  
    Error_Counter_RS++;  
    memset(RS_Resend_Counter, '\0', value2);  
    // cleaning the RS_Resend_Counter[] array!
    RS_Resend_Counter[value2] = Error_Counter_RS;  
    // sets the RS error counter to memory
    ev << " RS_Resend_Counter[ " << value2 << " ] ----> " << RS_Resend_Counter[value2] << ".. to be Cancelled !!..\n";
}
else     // First Report Segment received.
{
    if(timeoutEvent_CP[value2]->isScheduled())
    {
        Session_Closed_CP[value2] = value2;  
        // Sets the RS as closed!
        ev << " Session_Closed_CP[ " << value2 << " ] ----> " << Session_Closed_CP[value2] << " was setup !!..\n";
        ev << " " << timeoutEvent_CP[value2]->name() << ".. to be Cancelled !!..\n";
        cancelAndDelete(timeoutEvent_CP[value2]);  
        // Cancels the CP timer
    }
    else
    {  
        ev << "timeoutEvent_CP[value2] NOT Scheduled!!! \n\n";
    }
    ///
    ///
    ///
    ///
    ///
    ///

    ev << msg->name() <<" Received @ Cassini \n";
    delete msg;
    CP_index = CP_index_backup[value2];
}
Sending the Report Acknowledgement for the Data Segments !!!!

flags = ( 0x08 | 0x01 );    // set flags 1 & 8 for Report Acknowledgement
Zero_value = 0;       // I'm replacing value 3 with Zero_value just to send the RA to Earth_Station
cMessage *msg = new cMessage(Rep_Ack);
ev << "Sending " << Rep_Ack << " from Cassini to Earth Station \n";

value1 = (value1 & 0xFFFFFFFF);      // value_to_send
value2 = value2 << 8;        // Shifting bits to right
value2 = (value2 & 0xFFFFFFFF);      // value_to_send
Zero_value = Zero_value << 16;
Zero_value = (Zero_value & 0xFFFFFFFF);

value4 = value4 << 24;          // Shifting bits to right
value4 = (value4 & 0xFFFFFFFF);       // value_to_send

"************************************************************************************************************
 evacuation << "Flags --> " << flags << " || Value 4 --> " << (value4>>24) << " || Value 3 --> " << (value3>>16) << " || Value 2 --> " << (value2>>8) << " || Value 1 --> " << value1 << " \n"

Core = (flags << 27) | value4 | Zero_value | 0 | value1 ;    // add the 4-bit flags, Value2 is zero, RA just have RS number
msg->setKind(Core);
bubble("Sending RS to Earth Station");
send(msg, "out");
#if
#endif
#if
#endif

if (value3 != 0)  // If value3 (Byte containing Data Segments
Errors ) comes with an Error bit.....
{
    // ....start sending the Data
    // TESTING THE FIRST RA !!
    value1 = Core & (~0xFFFFFFFF00);  // mask out value 1
    value2 = Core & (~0xFFFFFFFFFF);  // mask out value 2
    value2 = value2 >> 8;
    value4 = Core & (~0xF8FFFFFFFF);  // mask out value 3
    value4 = value4 >> 24;
#endif
#endif

// Values for the first Loop
//
last_DS_error_CP = value3;
   // Assigning value3 to last_DS_error_CP to know what's
   // the last DS and include a Checkpoint flag
   last_DS_error_CP = last_DS_error_CP & 0x00FF;  // Cleaning
   // any bit after bit 7
   
   ///////////////////////////////////////////////////////////////////////////////////////////////
   ///////////////////////////////////////////////////////////////////////////////////////////////
   // Values for the first Loop
   ///////////////////////////////////////////////////////////////////////////////////////////////
   ///////////////////////////////////////////////////////////////////////////////////////////////
   
   byte_to_rotate = value3;  // Assigning value3 to rotate
   byte_to_rotate = byte_to_rotate & 0x00FF;  // Cleaning any bit after bit 7
   
   ///////////////////////////////////////////////////////////////////////////////////////////////
   ///////////////////////////////////////////////////////////////////////////////////////////////
   
   ev << " value2 --->  " << value2 << " CP --->  " << CP << " CP_index --->  " << CP_index << " 

   for (rotate=0; rotate<8; rotate++)
   {
      error_bit= last_DS_error_CP & (~0xFE);    // Masking Bit 0 to know if it has error
//ev << "Bit " << rotate << " ---> " << error_bit << "\n";

if(error_bit == 1)
{
    last_DS_with_CP++;  // last DS with CP
    last_DS_with_CP = (last_DS_with_CP >> 1);  // shift bit
    //ev << "last Red Data Segment with CP --->  " << last_DS_with_CP

    error_bit = 0;     // Reseting error_bit from previous loop
    rotate = 0;       // Reseting rotate from previous loop

    // Rotating the 8 bits of value 3 (Error Byte) to know which Data Segments we need to reset

    for (rotate=0; rotate<8; rotate++)
    {
        error_bit = byte_to_rotate & (~0xFE);           // Masking Bit 0 to know if it has error
    }
}

error_bit = 0;     // Reseting error_bit from previous loop
rotate = 0;       // Reseting rotate from previous loop
if(error_bit == 1)
{
    counting_error_bit++;    // Counting error bits in the loop

    DS_to_resend = (CP_index*8) + rotate;
    ev << "Red Data Segment to resend ---> " << DS_to_resend << "\n";

    Core = ds_backup[DS_to_resend];    // Getting the Core requested from memory

    Core = Core & 0xFFFFFFFF;    // mask out 32-bits

    flags = Core & (~0x07FFFFFF);    // mask out all flags
    flags = (flags >> 27) & 0x0F;    // Shift the bits 28 bits to the right (to the Less Significant Nibble) and clear

    value1 =  Core & (~0xFFFFFF00);    // mask out the value 1
    value2 =  Core & (~0xFFFF00FF);    // mask out the value 2
    value3 =  Core & (~0xFF00FFFF);    // mask out the value 3
    value4 =  Core & (~0xF8FFFFFF);    // mask out the value 4

    value1 = RS_number;    // As arrived from Earth Station
    value2 = first_block + block;    // Initial block number plus the block number
    value2 = value2 + New_CP_counter;    // value2 plus number of new CP or block numbers
    value3 = DS_to_resend;    // value3
    value4 = 0;

    if (counting_error_bit == last_DS_with_CP)
    {
        CP_Counter_to_ReTx_Last_DS++;    // Counter to know when to send RDS, CP, EORP, EOB
    }
if(Last_RS_Received==1)
{
    flags = (0x02 | 0x01); // set flag to 3 (last Red Data Segment as RDS, CP, EORP, EOB)
    CP_index_backup[value2]= CP_index;
    Last_RS_Received=0;
}
else
{
    flags = (0x01);
    CP_index_backup[value2]= CP_index;
}
else
{
    flags = (0x00);
    // set flag to 0 (Red Data Segment) again to resend it
    // TEST THE NEXT R A !!
}

value1 = (value1 & 0xFFFFFFFF);
// value_to_send
value2 = value2 << 8;
// Shifting bits to the right
value2 = (value2 & 0xFFFFFFFF);
// value_to_send
value3 = value3 << 16;
// Shifting bits to the right
value3 = (value3 & 0xFFFFFFFF);
// value_to_send
value4 = value4 << 24;
// Shifting bits to the right
value4 = (value4 & 0xFFFFFFFF);

char Rep_Ack[32];
if (counting_error_bit == last_DS_with_CP)  //This "IF" is used to send Last DS with CP and the rest without it...
sprintf( Rep_Ack, " Red Data Segment-%d, CP-%d, RS-%d,

                           // RS random number... just testing
                         
                         cMessage *msg = new cMessage(Rep_Ack);
                         
                         ev << "Sending " << Rep_Ack << " from Cassini to Earth 

                         Core = (flags << 27) | value4 | value3 | value2 | value1;
                         
                         /*
                           // Values to be followed by the 'Watch' function ///////////////
                           
                           Report_Segment_CS = value1;
                           CheckPoint_CS = value2>>8;
                           Red_Data_Segment_CS = value3>>16;
                           //Reserved_Value_CS = value4>>24;
                           
                           Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);
                           Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);
                           CheckPoint_CS_Vector.record(CheckPoint_CS);
                           CheckPoint_CS_Stats.collect(CheckPoint_CS);
                           Report_Segment_CS_Vector.record(Report_Segment_CS);
                           Report_Segment_CS_Stats.collect(Report_Segment_CS);
                           */
                         
                         msg->setKind(Core);
                         
                         bubble("Resending Red Data Segment Lost to Earth Station
                         
                         send(msg, "out");
                         
                         
                         p  ///////////////////////////////////////////////////////////////////////////////////////////
// Simulation of LPT in OMNet++

 value2 = value2 >> 8;
 CP_backup[value2] = Core;
 value2 = value2 << 8;

 flags = ( 0x08 );   // set 'flags' to 0x80

 value1 = 0;
 value4 = 0;

 char timermsg[32];
 sprintf(timermsg, "Timer CP-%d",value2>>8);

 ev << "Generating " << timermsg << endl;

 timeoutEvent_CP[value2>>8]= new cMessage(timermsg);
 ev << " " << timeoutEvent_CP[value2>>8]->name() << " was Setup!! \n\n";

 value1 = (value1 & 0xFFFFFFFF);

 value4 = value4 << 24;

 value4 = (value4 & 0xFFFFFFFF);

 ev << "*********************************************************************************************************
" << "Flags --> " << flags << "     ||    Value 4 --> " << (value4>>24) << "      ||    Value 3 --> " << (value3>>16) << "     ||    Value 2 --> " << (value2>>8) << "      ||    Value 1 -" << value1 << "  
" << "*********************************************************************************************************

 Core = (flags << 27) | value4 | value3 | value2 | value1 ;

 timeoutEvent_CP[value2>>8]->setKind(Core);
scheduleAt(simTime()+ timeout, timeoutEvent_CP[value2>>8]);

ev << "Event Scheduled ------> " <<

//

//

///////////////////////////////////   E N D   O F   T I M E
R   S E T U P   //////////////////////////////////////////////////////////////

Timer_CP_counter++;

// Timer_CP_counter = This counts every CP (value2) generated by Casînni

else

    sprintf( Rep_Ack, " Red Data Segment-%d, RS-%d ",DS_to_resend, value1 );    // RS random number... just testing

    cMessage *msg = new cMessage(Rep_Ack);

    ev << "Sending " << Rep_Ack << " from Cassini to Earth 

    Core = (flags << 27) | value4 | value3 | value2 | value1 ;

    /*
     * // Updating Statistics of the number of Red Data Segments Sent
     * Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);
     */

    // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);

    // Reserve the 'Watch' function //////////////////////////////////////////////////////////////

Station \n";

    // add the 4-bit flags

    Core = (flags << 27) | value4 | value3 | value2 | value1 ;

    Report_Segment_CS = value1;
    CheckPoint_CS = value2>>8;
    Red_Data_Segment_CS = value3>>16;
    //Reserved_Value_CS = value4>>24;

    // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Vector.record(Red_Data_Segment_CS);

    // Updating Statistics of the number of Red Data Segments Sent
    Red_Data_Segment_CS_Stats.collect(Red_Data_Segment_CS);
// Updating Statistics of the number of CheckPoint Sent
CheckPoint_CS_Vector.record(CheckPoint_CS);
CheckPoint_CS_Stats.collect(CheckPoint_CS);
// Updating Statistics of the number of CheckPoint Sent
Report_Segment_CS_Vector.record(Report_Segment_CS);
Report_Segment_CS_Stats.collect(Report_Segment_CS);
// Updating Statistics of the number of Report Segment Sent
// Updating Statistics of the number of Report Segment Sent
//////////////////////////////////////////////////////////////////////////////////////////////////////////////
*/
msg->setKind(Core);
bubble("Resending Red Data Segment Lost to Earth Station
send(msg, "out");
}

byte_to_rotate = (byte_to_rotate >> 1);    // shift bit
to right, then check the bit again to know if it comes with error
}

byte_to_rotate = (byte_to_rotate >> 1);    // shift bit
to right, then check the bit again to know if it comes with error
}

counting_error_bit = 0;      // Reseting the
counter of the last Red Data Segment to resend with CP..
error_bit = 0;        // Reseting the
error_bit from the previous loop
rotate = 0;         // Reseting
the rotate variable
last_DS_with_CP = 0;      // Reseting the
last_DS_with_CP counter

Page 134 of 144
New_CP_counter++;       // Checkpoint counter of the DS that are resend

CP_Array_counter++;

break;

case (0x08 | 0x01):     // 1 0 0 1
9 ----> Report Acknowledgment (RA)

    ev << "Report Acknowledgment (RA) " << Core << " " << flags <<"\n";

break;

case (0x08 | 0x04):     // 1 1 0
0 12 ----> Cancel Segment from Block Sender (CS)

    ev << "Cancel Segment from Block Sender (CS) \n";

break;

case (0x08 | 0x04 | 0x01): // 1 1 0 1
13 ----> Cancel Acknowledgment Segment to Block Sender (CAS)

    cancelAndDelete(timeoutEvent_Cancel[value2]);  // Cancels the Cancellation timer
    ev << "Cancel Acknowledgment Segment Received from Earth Station CAx \n\n";
    ev << " Session Closed !! \n\n";
    delete msg;

break;

case (0x08 | 0x04 | 0x02): // 1 1 1 0
14 ----> Cancel Segment from Block Receiver (CR)

    ev << " Cancel Segment from Block Receiver (CR) \n";

break;

case (0x08 | 0x04 | 0x02 | 0x01): // 1 1 1 1
15 ----> Cancel Acknowledgment Segment to Block Receiver (CAR)

    ev << " Cancel Acknowledgment Segment to Block Receiver (CAR) \n";
break;
default:
    ev << " UNDEFINED!! \n";
break;
}
if ( CP_Array_counter == Total_CP_number)
    CP_Array_counter=0;

} // else timer brace
} // else brace

void Receiver_CS::finish()
{
    // This function is called by OMNeT++ at the end of the simulation.
    double t=simTime();
    if (t==0) return;

    ///////////////////////////////////////////////////////////////////////////
    Cassini Info //////////////////////////////////////////////////////////////////////////
    recordScalar("RDS_ini_Counter", RDS_ini_Counter);
    recordScalar("GDS_ini_Counter", GDS_ini_Counter);
    recordScalar("Timer_CP_counter", Timer_CP_counter);
    recordScalar("Report_Segment_CS", Report_Segment_CS);
    recordScalar("CheckPoint_CS", CheckPoint_CS);
    recordScalar("Red_Data_Segment_CS", Red_Data_Segment_CS);
    recordScalar("Green_Data_Segment_CS", Green_Data_Segment_CS);
}
Dummy_Cloud.cpp

// This file is part of Licklider Transmission Protocol Simulation based on OMNeT++.
// Copyright (C) 2007 Fernando Carrasco
// This file is distributed WITHOUT ANY WARRANTY.

#include <string.h>
#include <omnetpp.h>
#include <iostream>
#include <string>
using namespace std;

class Dummy_Cloud : public cSimpleModule
{
private:
    int Core;

protected:
    virtual void handleMessage(cMessage *msg);
    virtual void displayStatus(bool isBusy);
};

Define_Module(Dummy_Cloud);

void Dummy_Cloud::handleMessage(cMessage *msg)
{
    unsigned int Core = msg->kind();
}
ev << "Core --> " << Core << " \n\n";

if (ev.isGUI()) displayStatus(true);
ev << "Sending " << msg->name() <<" .....\n";
send(msg, "out" );
if (ev.isGUI()) displayStatus(false);
}

void Dummy_Cloud::displayStatus(bool isBusy)
{
    displayString().setTagArg("t",0, isBusy ? "RF Signal Traveling..." : " ");
    displayString().setTagArg("i",1, isBusy ? "blue" : "white" );
}

LTP.ned

//
// This file is part of Licklider Trasmission Protocol Simulation based on OMNeT++.
//
// Copyright (C) 2007 Fernando Carrasco
//
// This file is distributed WITHOUT ANY WARRANTY.
//

///////////////////////////////////////////////////////////////////////////////
/// DUMMY CLOUD - SIMPLE MODULE /////////////////////////////////////////////////////////////////////////////////

simple Dummy_Cloud
    gates:
        in: in;
        out: out;
endsimple

///////////////////////////////////////////////////////////////////////////////
/// CASSINI - TRANSMITTER - SIMPLE MODULE /////////////////////////////////////////////////////////////////////////////////

simple Transmitter_CS
    gates:
        in: in;
        out: out;
endsimple
simple Receiver_CS

parameters:
Red_data_segments: numeric,    // Number of Red Data Segments we want to send
Green_data_segments: numeric,  // Number of Green Data Segments we want to send
CP_initial: numeric,           // First Random CheckPoint to initiate the transmission. From Omnetpp.ini
timeout: numeric;

gates:
in: in;
out: out;

endsimple

simple Transmitter_ES

gates:
in: in;
out: out;

endsimple

simple Receiver_ES

parameters:
Error_probability: numeric,
First_RS: numeric,
timeout: numeric;

gates:
in: in;
out: out;

endsimple

 simples Receiver_ES

parameters:
Error_probability: numeric,
First_RS: numeric,
timeout: numeric;

gates:
in: in;
out: out;

endsimple

simple Transmitter_ES

gates:
in: in;
out: out;

endsimple

simple Receiver_ES

parameters:
Error_probability: numeric,
First_RS: numeric,
timeout: numeric;

gates:
in: in;
out: out;

endsimple

 simples Receiver_ES

parameters:
Error_probability: numeric,
First_RS: numeric,
timeout: numeric;

gates:
in: in;
out: out;

endsimple
Simulation of LPT in OMNet++

module Tx_Rx_block_ES

parameters:
    Error_probability: numeric;

gates:
    in: in;
    out: out;

submodules:
    Tx_ES: Transmitter_ES; //Tx_ES: TxRx_ES;
    display: "p=128,108;i=misc/square_l";
    Rx_ES: Receiver_ES; //Rx_ES: TxRx_ES;
    parameters:
        Error_probability = Error_probability;
    display: "p=208,108;i=misc/square_l";

connections:
    Tx_ES.out --> out;
    in --> Rx_ES.in;
    Rx_ES.out --> Tx_ES.in;

    display: "b=400,200";
endmodule

module Tx_Rx_block_CS

parameters:
    Red_data_segments: numeric,
    Green_data_segments: numeric;

gates:
    in: in;
    out: out;

submodules:
    Tx_CS: Transmitter_CS; //Tx_CS: TxRx_CS;
    parameters:
        display: "p=112,108;i=misc/square_l";
Simulation of LPT in OMNet++

Rx_CS: Receiver_CS; //Rx_CS: TxRx_CS;
   parameters:
       Red_data_segments = Red_data_segments,
       Green_data_segments = Green_data_segments;
   display: "p=280,108;i=misc/square_1";
connections:
   Tx_CS.out --> out;
in --> Rx_CS.in;
Rx_CS.out --> Tx_CS.in;
   display: "b=400,200";
endmodule

module ES_block
   parameters:
       Error_probability: numeric;
gates:
in: in;
out: out;
submodules:
   Transceiver_ES: Tx_Rx_block_ES;
   parameters:
       Error_probability = Error_probability;
   display: "p=92,104;i=block/network_1";
connections:
   Transceiver_ES.out --> out;
in --> Transceiver_ES.in;
   display: "b=170,200";
endmodule

module CS_block
parameters:
    Red_data_segments: numeric,
    Green_data_segments: numeric;

gates:
in: in;
    out: out;

submodules:
    Transceiver_CS: Tx_Rx_block_CS;
        parameters:
            Red_data_segments = Red_data_segments,
            Green_data_segments = Green_data_segments;
        display: "p=100,104;i=block/network_1";
    connections:
        Transceiver_CS.out --> out;
        in --> Transceiver_CS.in;
        display: "b=170,200";
endmodule

module LTP_Sim_1
    parameters:
        Red_data_segments: numeric,
        Green_data_segments: numeric,
        Error_probability: numeric;

submodules:
    Earth_Station: ES_block;
        parameters:
            Error_probability = Error_probability;
        display: "p=436,208;i=device/receiverdish_1";
    Cassini: CS_block;
        parameters:
            Red_data_segments = Red_data_segments,
            Green_data_segments = Green_data_segments;


display: "p=92,56;i=device/satellite_1";
Space_1: Dummy_Cloud;
parameters:
  display: "p=190,56;i=misc/cloud_1";
Space_2: Dummy_Cloud;
parameters:
  display: "p=342,208;i=misc/cloud_1";
connections:
  Earth_Station.out --> RF_UpLink --> Space_2.in;
  Space_2.out --> Cassini.in;
  Cassini.out --> RF_DownLink --> Space_1.in;
  Space_1.out --> Earth_Station.in;
display: "b=519,272";
endmodule

EMPLATE lc
S

cP

------------------------------
//GENERAL LTP NETWORK
------------------------------

network LTP : LTP_Sim_1
parameters:
  Red_data_segments = input(5, "Number of Red Data Segments to Send : "); // Initial window to request the number of Data segments
  Green_data_segments = input(5, "Number of Green Data Segments to Send : "); // Initial window to request the number of Data segments
  Error_probability = input(0, "Probability of Error in Data Segments Transmission (0-100): "); // Initial window to request the number of Data segments
endnetwork

------------------------------
//UPLINK AND DOWNLINK RF LINK CHANNEL PARAMETERS
------------------------------

channel RF_UpLink
  delay 74m; // min
  //error 1e-10;
  datarate 57344; // bit/sec (56kbps = 56000*1024-->57344)
endchannel

channel RF_DownLink
  delay 74m; // min

------------------------------


Simulation of LPT in OMNet++

```
//error 1e-10;
datarate 57344; // bit/sec (56kbps = 56000*1024-->57344)
endchannel

Omnetpp.ini

[General]
network = LTP

[Parameters]
LTP.Cassini.Transceiver_CS.Rx_CS.msgLength = intuniform(8,32768)

// First Random Checkpoint Number =
LTP.Cassini.Transceiver_CS.Rx_CS.CP_initial = 79

// First Random Report Segment Number =
LTP.Earth_Station.Transceiver_ES.Rx_ES.First_RS = 27

// Timeout of Timers in Cassini Receiver =
LTP.Cassini.Transceiver_CS.Rx_CS.timeout = 9324

// Timeout of Timers in Earth Station Receiver =
LTP.Earth_Station.Transceiver_ES.Rx_ES.timeout = 9324

// Cassini's Message Length =
LTP.Cassini.Transceiver_CS.Rx_CS.msgLength = 1024
```